Cross-Language Process Execution Benchmarking

Student: Butas Rafael-Dorian

Structure of Computer Systems Project

Technical University of Cluj-Napoca

Contents

1.Introduction

1.1 Context

In modern computing, software performance is influenced by how efficiently processes such as memory allocation, memory access, thread creation, context switching, and thread migration are managed. These processes can significantly affect system performance, particularly in multi-threaded environments. Programming languages like C, C++, and Java offer distinct runtime environments, memory management models, and threading mechanisms, which impact performance differently.

This project aims to measure and compare the performance of key processes (memory management, thread operations, etc.) across different programming languages: C, C++, and Java. By understanding these performance differences, developers can make more informed decisions regarding language selection for performance-critical applications.

1.2 Objectives

The primary goal of this project is to create a tool that measures and compares the execution times of processes in three programming languages (C, C++, Java). The specific objectives include:

- Measuring the execution of memory management strategies in each language, focusing on static and dynamic memory allocation.
- Measuring and comparing the efficiency of thread creation, context switching, and thread migration.
- Using high-precision time measurement techniques in each language.
- Storing measurement results in JSON files for easy data analysis.
- Generating visual graphs from the results, representing process performance across languages.
- Generating textual explanations of the graphs and recommendations.
- Designing a GUI that enables the user to run code written in different languages and visualize the comparison results.

1.3 Project Proposal

This project will develop a cross-language benchmarking tool that measures the execution time of critical processes in three selected programming languages (C, C++, Java). The tool will focus on evaluating the performance of key operations such as memory allocation (static and dynamic), memory access, thread creation, context switching, and thread migration.

To achieve this, the project will implement the following:

- **Measurement Methods:** In C/C++, the <chrono> library will be used for high-resolution time measurement, and thread operations will be handled using pthread_create. In Java, nanoTime() will be used for time measurements and the Java Thread API for threading operations.

- **Data Storage:** The results will be stored in JSON format using libraries such as cJSON (C), nlohmann/json (C++), and Jackson/JSON-simple (Java). Each JSON file will contain detailed information about the process being measured (e.g., memory management or threading), the execution time, and the programming language used.

- **Visualization and Graphs:** Using the JFreeChart library, performance data will be extracted from the JSON files and visualized in graphical form. Each graph will represent the performance of different processes across languages. For example, one graph may compare memory allocation times between C, C++, and Java, while another compares the time taken for thread creation and migration.

- **Explanations and Recommendations:** Alongside the graphs, textual explanations will be provided to help interpret the data. These explanations will:

    - **Analyze Trends:** Highlight patterns in the graphs, such as which language performed better in specific scenarios (e.g., faster memory access or more efficient thread handling).

    - **Provide Insights:** Explain why certain languages excel in particular processes, based on their runtime environments or memory/thread management models.

    - **Offer Recommendations:** Based on the results, recommendations will be made for different use cases. For instance, if C++ shows superior performance in dynamic memory allocation, the tool will suggest using C++ for applications requiring heavy memory management. Conversely, if Java handles thread migration more efficiently, it will be recommended for systems with high concurrency needs.

- **Cross-Language Execution & GUI:** A Java-based GUI will be developed to run the benchmarking tests across C, C++, and Java programs. The GUI will utilize the Java Native Interface (JNI) to interact with C/C++ code, enabling cross-language execution.

Users will be able to select specific tests to run and receive both visual (graphs) and textual feedback (explanations and recommendations) on the results.

The GUI will feature:

- **Test Selection Panel:** Users can choose which performance aspects (e.g., memory, threads) to benchmark.
- **Graphical and textual Output:** The GUI will display graphs with corresponding textual explanations and recommendations appearing next to the graphs.
- **Export Functionality:** Users will be able to export the graphs and the textual analysis for future reference or reporting.

2. Bibliographic Research

2.1 Memory Management

Memory management is the process of controlling and coordinating a computer's main memory. It ensures that blocks of memory space are properly managed and allocated so the operating system, applications and other running processes have the memory they need to carry out their operations. [1]

- What are the 3 areas of memory management?
  - **Memory management at the hardware level** which is concerned with the physical components that store data (RAM chips, memory caches). The management at this level is done by the MMU (memory management unit) whose role is to translate the logical addresses (used by running processes) to physical addresses on the memory devices. [1]
  - **Memory management at the OS level** which involves allocation of specific memory blocks to individual processes. The OS moves these processes between memory and storage devices (HDD or SSD), while tracking each memory location. When the computer runs out of physical memory space, the OS turns to virtual memory (allocated from the storage device), but this can impact performance because storage device memory is slower than the computer's main memory. [1]
  - **Memory management at the application level** which ensures the availability of adequate memory for the program's objects and data structures. This is achieved through allocation. This can be manual (done by the developer) or automatic (done using an allocator). When the developer no longer needs the memory space, that memory is released for reassignment (recycling). [1]
- Memory management in C/C++
  - In C/C++, memory management falls on **your shoulders**. This can be hard to do, but it gives you the means to optimize the performance of the program. Understanding how to release memory when it's no longer needed and using just enough for the task at hand is crucial. [2]
  - **Static allocation (on the stack)** refers to the process where the memory for variables is allocated at **compile time**. The amount of memory required is determined then the program is compiled and that specific amount is reserved for the entire duration of the execution. [2]

- **Dynamic allocation (on the heap)** refers to the process where the memory is allocated at **runtime**. This allows programs to use memory more flexibly and efficiently. Dynamic memory allocation is managed using functions loke malloc (new in C++) (allocates a specified number of bytes and returns a pointer) ,calloc (allocates memory of an array of elements, initializes them to zero, and returns a pointer) , realloc (changes the size of previously allocated memory) and free (delete in C++) (deallocates the memory). It gives you complete control over the amount of memory used at any time. It can only be accessed via **pointers**. [2]
- A **memory leak** occurs then a program fails to release an allocated memory which is no longer needed. If this memory leaks accumulate, they will consume resources leading to worse performance, system instability/crashes. [2]

- Memory management in Java
  - Java **itself** manages the memory and no special interventions of programmers in needed as the Garbage Collector has the role of clearing (free in C) the objects that are no longer used. But the Garbage collector doesn't guarantee the deallocation of memory that is still referenced, which can lead to memory leaks that can't be managed by the Java Virtual Machine (JVM). [3]
  - **JVM Memory Structure:** it has 3 areas: Method Area (for methods, classes, constructors, can be fixed or dynamic), Heap Area (shared runtime data for instances and arrays), Stack Area (store method specific values that have a short lifetime, can be fixed or dynamic). [3]
  - **Java Garbage Collection:** the process of collecting memory occupied by unreferenced objects. The Garbage Collector is controlled by JVM, which chooses when to run it (when in detects low availability of memory resources). [3]

2.2 Threads

A thread is, fundamentally, a subject or a topic. A thread may exist in human communication, such as an email exchange. In technology, a thread is typically the smallest set of instructions that a computer can manage and execute; it is the basic unit of processor utilization. There are four elements used to control a thread: ID, program counter, register set, stack allocated to the thread. A CPU core works on one thread at a time. To optimize execution performance, the CPU can order the thread's instructions for efficiency. [4]

- **Multithreading** is a CPU core design that lets two or more threads execute simultaneously. The threads share resources, even if they don't interact at all. We

need to avoid conflicts when multithreading, such as race conditions or deadlocks. Multithreading can be parallel (cores actually handle more threads) or concurrent (cores only handle one thread at a time but the processor handles more threads. [4]

- Thread Creation

  - In C/C++: You can use the **pthread_create** function to create a new thread. The **pthread.h** header file includes its signature definition along with other thread-related functions.

  - In Java: There are two ways to create a thread in java, one by extending **Thread class** (using the start() method) and the other by implementing **Runnable/Callable** Interface (using the run()/call() method of it).

- Thread Context Switch: A thread context switch refers to the process of saving the current execution context of a running thread and restoring the execution context of another thread to allow it to run. In a multithreaded environment, multiple threads within a single process can execute concurrently, and the operating system performs thread context switches to switch the execution between different threads. They are fast and have lower overhead compared to process context switching. The 4 elements of the thread are saved, it allows to continue from where it left off. There is no need to switch the memory address space because it is shared and directly accessed. In order to measure thread context switch time we will create two threads, use a synchronization mechanism (mutex, condition variable, semaphores, etc.) where thread 1 signals thread 2 to run and vice versa. [5]

- Thread Migration: Thread migration is the process of moving or migrating a thread from one processor to a remote processor. This provides dynamic load balancing and failure recovery in a multi-threaded environment. One of the most difficult problems while migrating the state of a thread is dealing with pointers in the migrant thread. If the pointers refer to data in the thread's stack, then they will only remain correct if the stack is placed in the same memory location on the destination processor as on source processor. A similar situation exists for pointers that reference data in the heap. In order to measure thread migration time, we need to pin a thread to one CPU core and then move it to another. In order to set CPU **affinity** we will use pthread_setaffinity_np to force a thread to run on a specific core. Doing it in Java is difficult because Java does not have direct APIs for setting thread affinity, but we can use **Java Native Interface** to call native C function or we can use external tool such as **taskset** (Linux) to control on which CPU's core the Java process can run on. We can also use **Windows Performance Analyzer** (WPA) to monitor thread migration time. [6]

3. Analysis

3.1 Project Analysis

Measuring the execution time of program or of parts of it is sometimes harder than it should be, as many methods are often not portable to other platforms. Choosing the right method will depend largely on your operating system, your compiler version, and also on what you mean by 'time'. For example, the methods that measure full seconds are not good for our scope because we need precision since our measurements will not take long. [7]

- **Wall time vs CPU time:** wall time (clock time) is the total time elapsed during the measurement (like a stopwatch). CPU time refers to the time the CPU was busy processing the instructions of the program. [7]
- **Best methods for our purposes**: the best and most portable way to measure wall time on C++ is using <chrono>. The <chrono> library has access to a few different clocks in your machine, each of them with different purposes and characteristics. The best one is high_resolution_clock, it uses the clock with the highest resolution available. The best (and most complicated) way to measure CPU time on windows is using <processthreadsapi.h> and GetProcessTimes(), or we can use <time.h> and clock() in Linux if we want a less complicated way to measure CPU time. In C, clock_gettime() is used for high-resolution time measurement, providing precise timestamps suitable for benchmarking tasks. It retrieves the current time with nanosecond precision, typically using either CLOCK_REALTIME (system-wide clock) or CLOCK_MONOTONIC (time since an unspecified starting point, which is unaffected by system clock changes). [7]
- **In Java:** Because we need high precision, the best method from java for measuring time is nanoTime(). This method can only be used to measure elapsed time and is not related to any other notion of system or wall-clock time. If we want to measure wall time in java, we need to use currentTimeMillis() but this method is not entirely accurate for time below one second. [7]

The GUI will be created using Java as it was discussed above. But how can it run C/C++ code? We do actually need to use code that's natively compiled for a specific architecture. To achieve this, the JDK introduces a bridge between the bytecode running in our JVM and the native code (usually written in C or C++). The tool is called **Java Native Interface**. [8]

- Java provides the *native* keyword that's used to indicate that the method implementation will be provided by a native code. Normally, when making a native executable program, we can choose to use static or shared libs. The latter is what makes sense for JNI as we can't mix bytecode and natively compiled code into the same binary file. Therefore, our shared lib will keep the native code

separately within its .dll file instead of being part of the java classes. [8]

- The native keyword transforms our method into a sort of abstract method with the main difference that instead of being implemented by another Java class, it will be implemented in a separated native shared library. [8]
  - **Components needed:** Java code, native code in C/C++, JNI header file for C/C++ (jni.h), C/C++ compiler. [8]
  - **C/C++ elements (defined within jni.h):** JNIEXPORT- marks the function into the shared lib as exportable so it will be included in the function table, and thus JNI can find it**.** JNICALL – combined with JNIEXPORT, it ensures that our methods are available for the JNI framework. JNIEnv – a structure containing methods that we can use our native code to access Java elements. JavaVM – a structure that lets us manipulate a running JVM (or even start a new one) adding threads to it, destroying it, etc. [8]

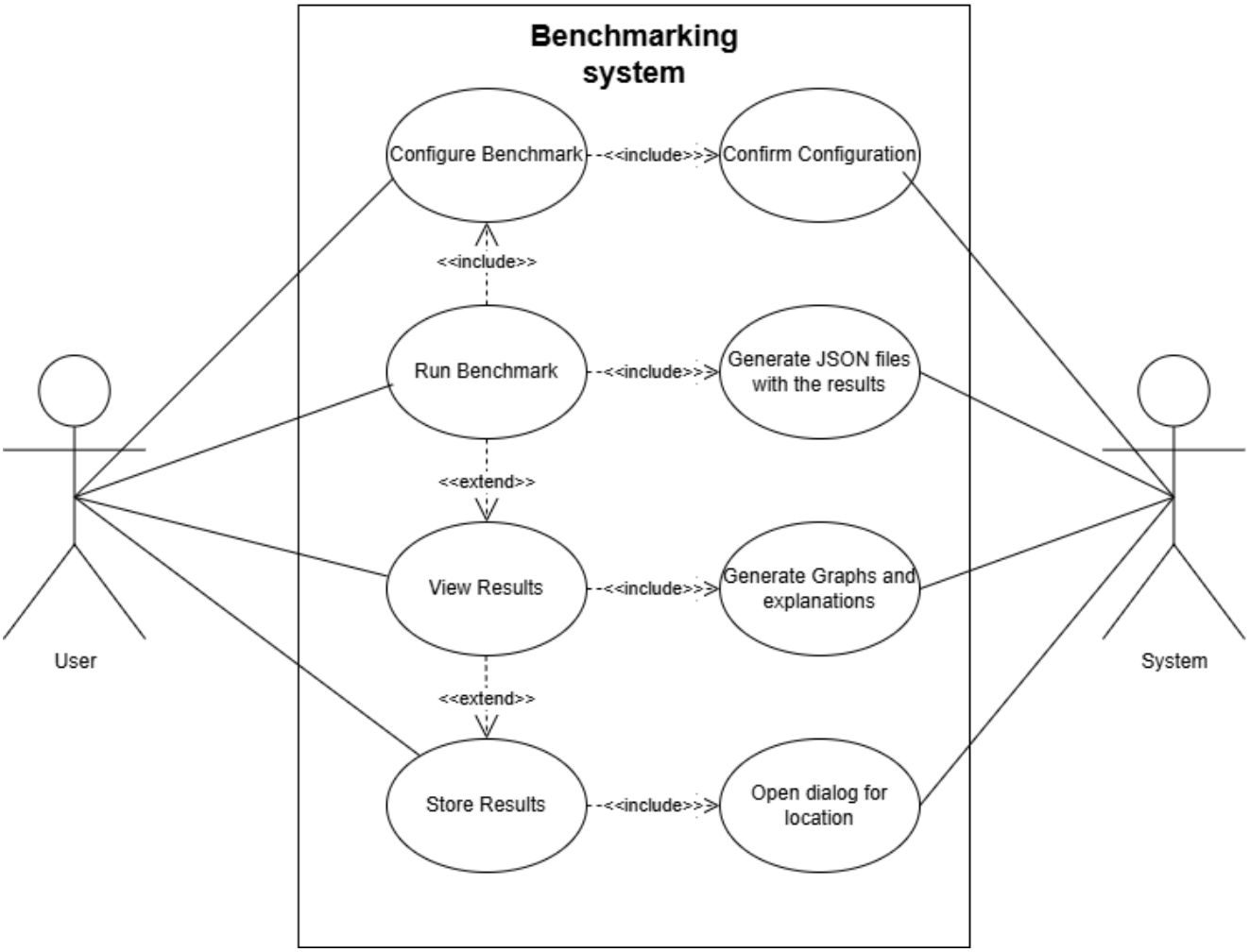
In performance benchmarking analyzing averages, standard deviations, and removing outliers is essential for obtaining meaningful, reliable insights. Here's how each metric and technique contributes to accurate measurement:

- **Averages**
  - **Mean (Arithmetic Average)**: The mean of multiple time measurements offers a straightforward central tendency, summarizing the performance of a particular operation. [9]
  - **Use in Benchmarking**: Calculating the average of multiple runs smooths out minor fluctuations due to unpredictable system behaviors, like background processes or I/O operations, which can briefly impact performance. [9]
- **Standard Deviation**
  - **Definition**: The standard deviation measures the spread of time measurements around the mean, indicating the variability or consistency in performance. A low standard deviation means the measurements are tightly clustered around the mean, suggesting consistent performance, while a high standard deviation shows variability. [9]
  - **Use in Benchmarking**: Inconsistent execution times may suggest underlying issues like memory allocation inefficiencies or unpredictable thread scheduling, which should be examined. [9]
- **Removing Outliers**
  - **Outliers**: Outliers are extreme measurements that deviate significantly from the rest, usually caused by unexpected interruptions or system anomalies. [9]
  - **Why Remove Outliers**: Outliers can skew the mean and inflate the standard deviation, leading to misleading results. For instance, a single long delay due to a system process can make the average appear slower than the true typical performance. We can remove outliers using the empirical rule. The empirical rule, or the 68-95-99.7 rule, tells you where most of the values lie in a normal distribution:
    - Around 68% of values are within 1 standard deviation of the mean.

- Around 95% of values are within 2 standard deviations of the mean.
- Around 99.7% of values are within 3 standard deviations of the mean [9]

3.2 Use cases

A use case diagram is a graphical representation of the interactions between users (actors) and a system, depicting the various use cases or functionalities of the system and how users interact with them. It provides a high-level view of the system's behavior from the perspective of its users. Here's a use case diagram representing the cross-language benchmarking project:



- **Configure Benchmark**
  - **Description**: The user selects a specific type of benchmark (e.g., Memory Allocation, Memory Access, Thread Migration) that they wish to analyze and a specific programming language in which that benchmark will run.
  - **Primary Actor**: User
  - **Preconditions**: The system presents available benchmark types and programming languages to the user.
  - **Basic Flow**:
    - User chooses a programming language from a list.

        - User chooses a benchmark type a list.
        - The system confirms the selection and prepares to run the benchmark.
    - **Postconditions**: The system is configured to execute the chosen benchmark type in the chosen programming language.

- **Run Benchmark**
    - **Description**: The user selects a programming language (C, C++, or Java) and a benchmarking type (e.g., memory access or thread creation). The system then runs the selected benchmark and records the execution time.
    - **Primary Actor**: User
    - **Preconditions**: The system is running, and the available languages and benchmark types are visible to the user.
    - **Basic Flow**:
        - User selects the programming language.
        - User chooses the benchmark type
        - User initiates the "Run Benchmark" action.
        - The system executes the benchmark, either directly in Java or through JNI for C/C++.
        - The system collects the benchmark data in JSON files.
    - **Postconditions**: The benchmark results are stored for further analysis.
    - **Alternate Flow**: If a selected benchmark or language fails to execute, an error message is shown to the user.

- **View Results**
    - **Description**: After a benchmark is completed, the user can view the results as a graph with corresponding textual explanations and recommendations appearing next to the graphs for easy analysis.
    - **Primary Actor**: User
    - **Preconditions**: A benchmark has been run and results are available for viewing.
    - **Basic Flow**:
        - User selects the "View Results" option.
        - The system generates a graph based on stored benchmark data.
        - The graph together with the explanations are presented to the user.
    - **Postconditions**: The user sees a visual and textual representation of benchmark results, aiding in performance comparison.
    - **Alternate Flow**: If no benchmark has been run, an error message informs the user that there are no results to view.

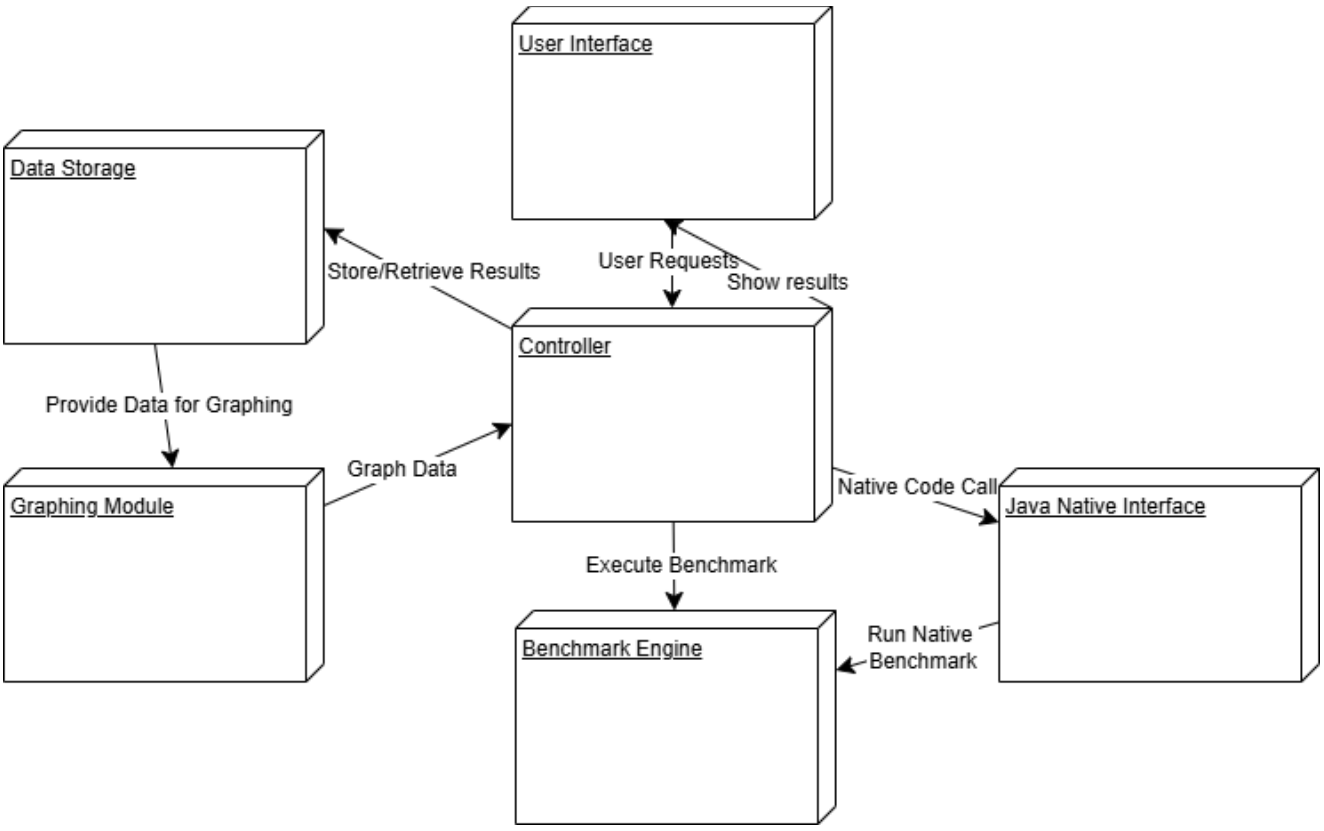
- **Export Results**
    - **Description**: The user exports the benchmark results as a JSON file and as photos with the graphs for documentation or further use.
    - **Primary Actor**: User
    - **Preconditions**: Benchmark results exist in the system.

- **Basic Flow**:
  - User selects "Export Results."
  - The system opens a dialog for the user to choose the file location.
  - Results are saved as a JSON file and as photos at the chosen location.
- **Postconditions**: The benchmark results are saved externally as a JSON file and as photos.
- **Alternate Flow**: If no results are available, an error message is shown to the user.

4. Design

4.1. Block Diagram

The block diagram shows the architectural flow of the system, detailing how components interact to accomplish the benchmarking tasks. It's more about the high-level, functional software architecture and dependencies that are easy to digest and understand:



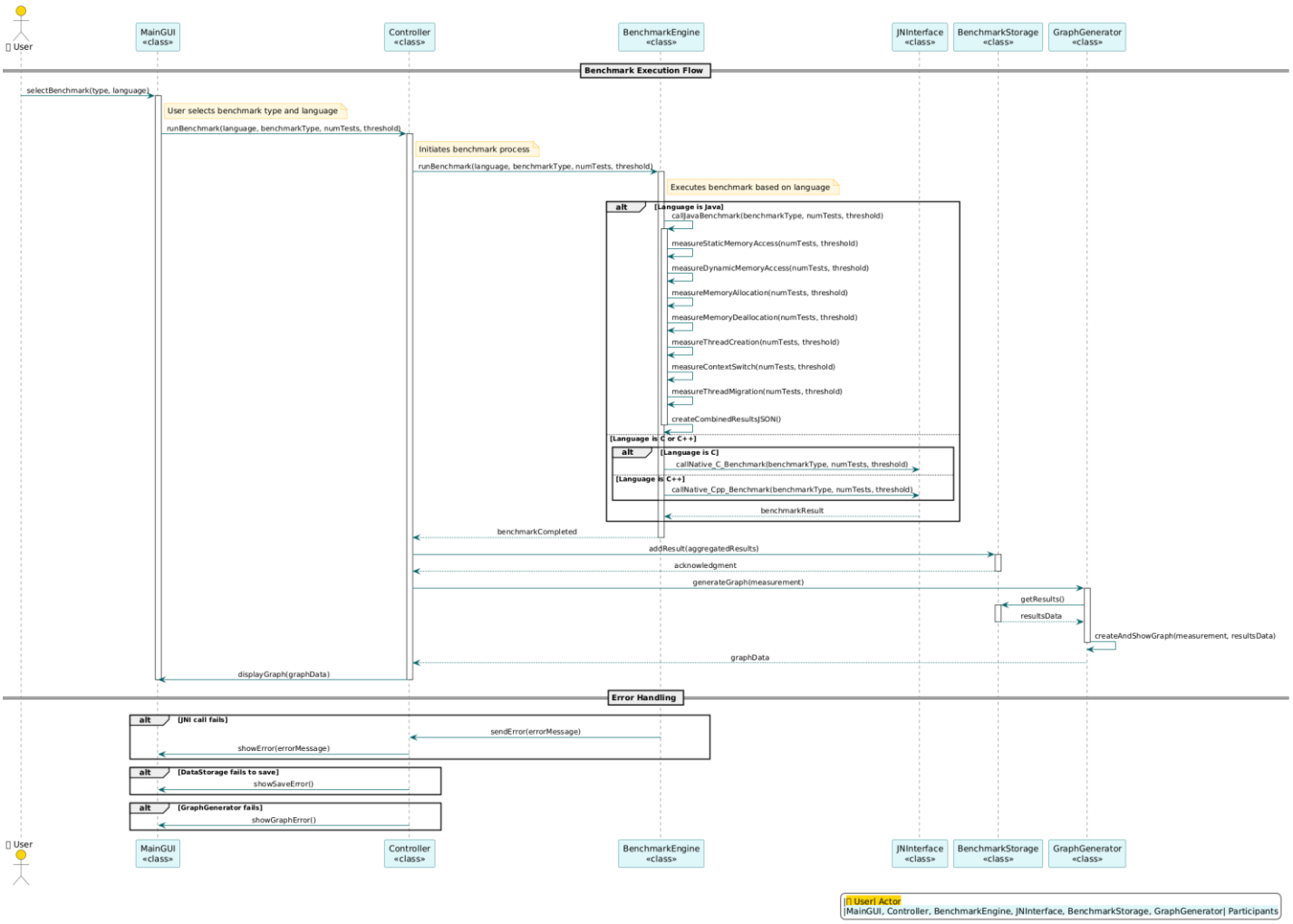
- **User Interface (UI)**: Represents the front-end where users interact with the system, select benchmarks, and view results.
- **Controller (MeasurementController)**: The main logic that coordinates benchmark selection, execution, and data retrieval.
- **Benchmark Engine**: This component houses the different benchmarking scripts or executables for each language (C, C++, Java). It receives instructions from the Controller and runs the specified benchmark.
- **JNI (Java Native Interface)**: Used to bridge Java and native code (C/C++). This layer

facilitates communication between Java GUI/Controller and native C/C++ benchmarks.

- **Data Storage**: Handles storing and retrieving results in a structured format like JSON.
- **Graphing Module**: Generates graphical and textual representations of benchmark results.

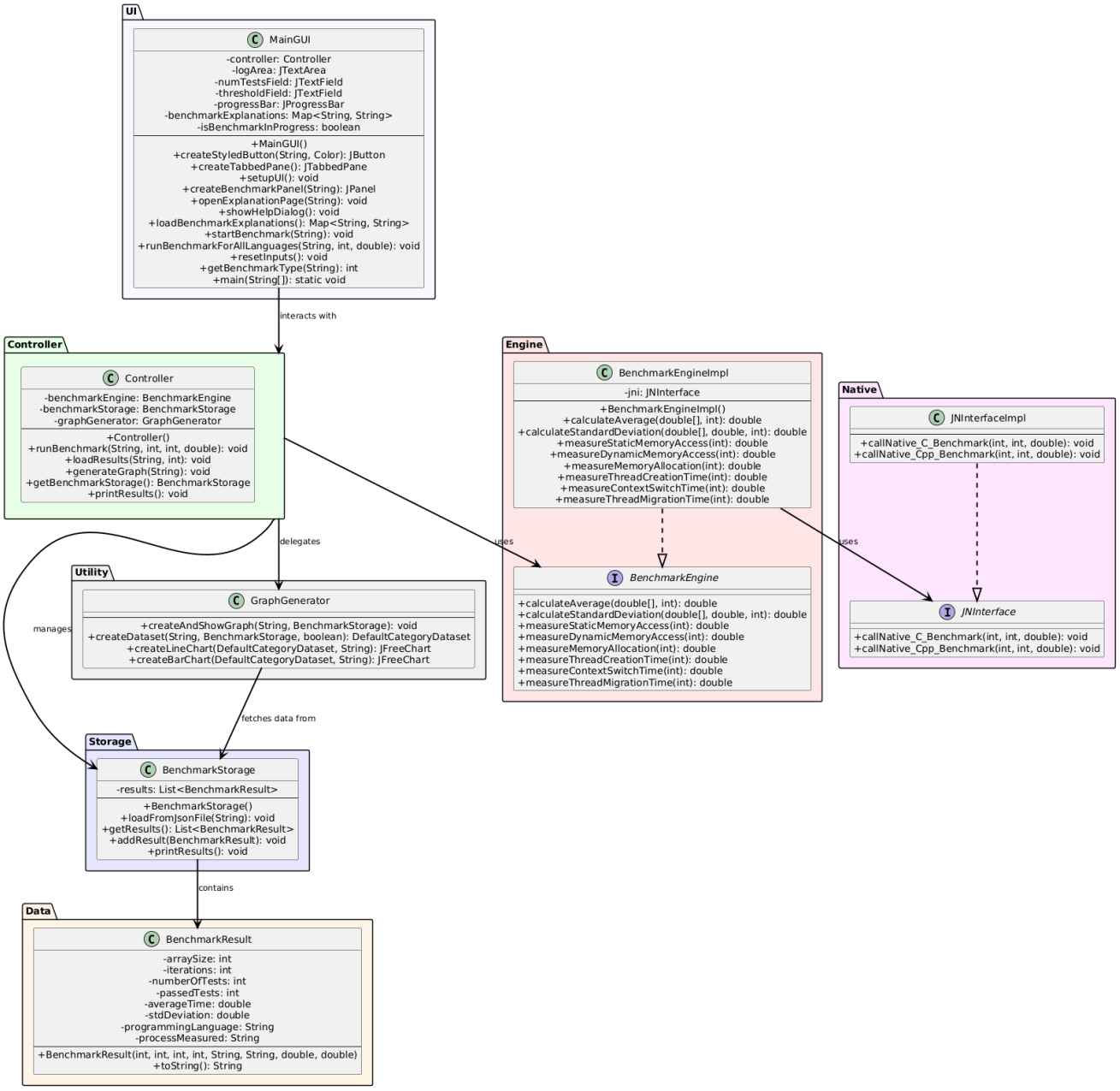
4.2. Sequence Diagram

The sequence diagram below illustrates the interaction between the primary components of the benchmarking system during the execution of a benchmark. It highlights how the user initiates a benchmark via the GUI, how the system processes the request through the Controller, and how results are generated and displayed. This dynamic representation provides insight into the flow of operations, from user input to result visualization.



4.3. Class Diagram

In software engineering, a class diagram in the Unified Modeling Language (UML) is **a type of static structure diagram** that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.



The class diagram above represents the main components and interactions in the Cross-Language Process Execution Benchmarking system. This system is designed to facilitate benchmarking across multiple programming languages, providing a graphical interface to view results and compare performance metrics. The key classes and their roles are as follows:

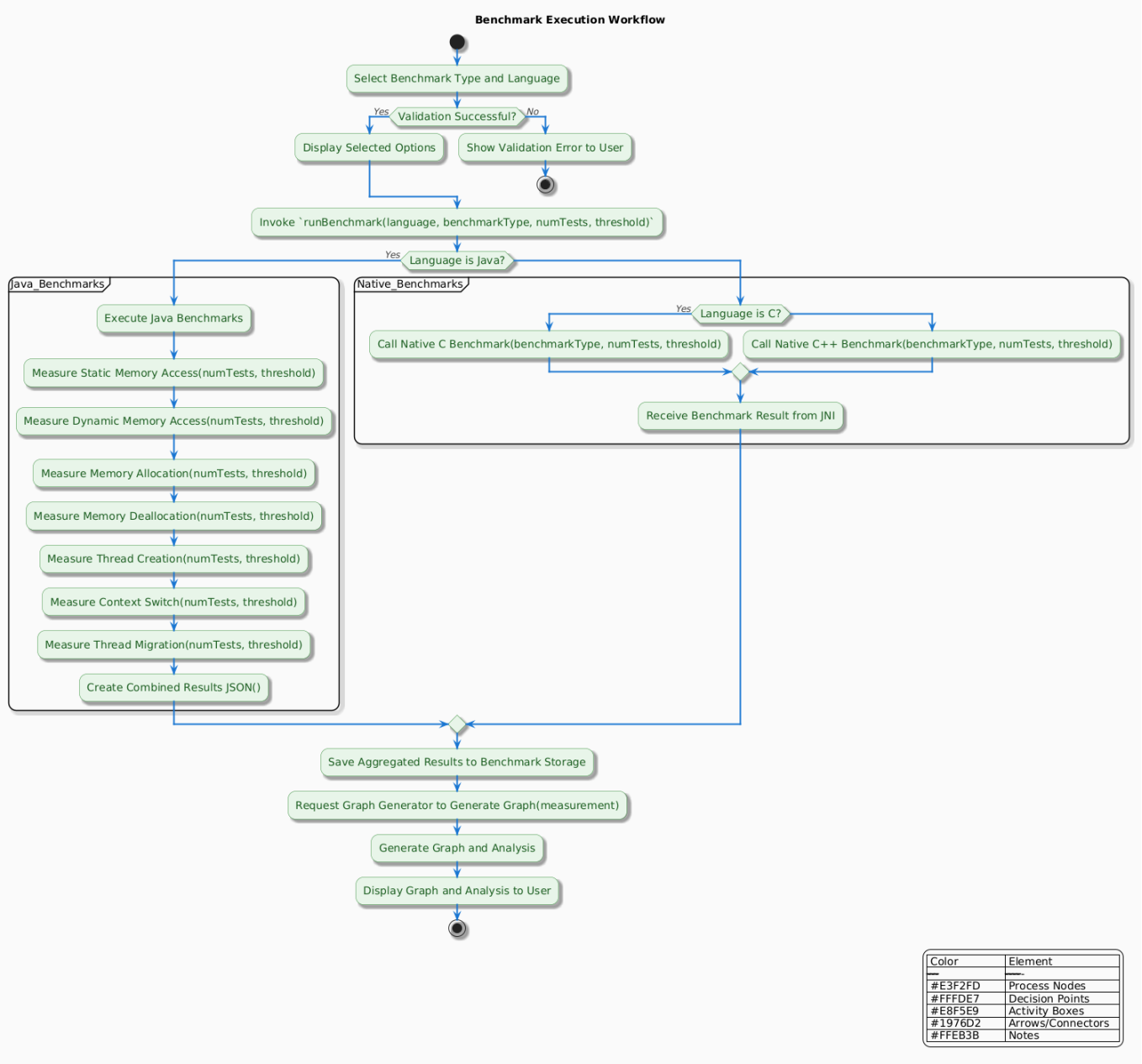
- **Controller**: Serves as the central coordinator, connecting the User Interface with the other components. The Controller initiates benchmarking through the

BenchmarkEngine and JNI Interface, manages data storage, and triggers the graph generation. Key methods in the Controller include runBenchmark(), which initiates the benchmarking process, and generateGraph(), which calls on the GraphingModule to visualize results.

- **BenchmarkEngine**: Executes the benchmarking process within the Java environment. It is responsible for running benchmarks based on selected languages and benchmark types, storing raw execution data for further processing. The Controller communicates with this class to obtain benchmark data and manage its execution.
- **JNI Interface**: Acts as a bridge to allow Java to interface with native C or C++ code, facilitating cross-language benchmarking. The Controller calls methods within this interface to execute benchmarks in other languages, with callNativeBenchmark() serving as the primary method for initiating native code benchmarks.
- **DataStorage**: Manages the persistent storage of benchmark results, including saving, loading, and retrieving results as needed. The Controller stores results in DataStorage for later retrieval and presentation.
- **GraphingModule**: Generates and displays graphs based on benchmark data. After retrieving benchmark data from DataStorage, the Controller calls generateGraph() to create visualizations. The displayGraph() method then presents these graphs to the User Interface.
- **User Interface**: The main entry point for users to interact with the application. It allows users to initiate benchmarks, view results, and see visualized performance comparisons. Methods in this class include initiateBenchmark() for starting the benchmarking process, and showResults() for displaying saved results.

4.4. Activity Diagram

The **Activity** diagram illustrates the step-by-step process of running benchmarks within the system. It starts with the user selecting the benchmark type and programming language, followed by validation of these inputs. If the validation is successful, the system proceeds to execute the benchmarks specific to the chosen language—either Java or a native language like C/C++. After executing the benchmarks, the results are aggregated and stored. The workflow then requests the generation of visual graphs and analyses, which are displayed to the user. Key decision points and distinct processing paths based on the programming language are clearly highlighted, providing a concise overview of the entire benchmark execution process.



5. Implementation

The Cross-Language Process Execution Benchmarking project comprises various functions for benchmarking system performance across C, C++, and Java. These functions measure key aspects of memory and thread management, including memory allocation, deallocation, memory access, thread creation, context switching, and thread migration. This section provides a detailed explanation of each function, its purpose, and how it was implemented.

5.1. Memory Benchmarking Functions

Memory benchmarking functions measure the efficiency of memory allocation, deallocation, and access. These metrics reveal how each language's memory model handles these processes.

5.1.1. Memory Allocation

- **Function Name**: `measureMemoryAllocation`
- **Purpose**: Measures the time required to allocate a block of memory of a specified size.
- **Implementation Details**:
    - **C Implementation**: In C, memory allocation is done using `malloc`, which dynamically allocates a specified number of bytes on the heap and returns a pointer to the start of this memory block. The `clock_gettime()` function, which provides nanosecond resolution, is used to record start and end times with `CLOCK_MONOTONIC` (to avoid inconsistencies due to system time changes). Timing begins immediately before the `malloc` call and stops right after it, and the allocation duration is calculated by subtracting the start time from the end time. This gives an accurate measurement of the allocation overhead.
    - **C++ Implementation**: Similar to C, the C++ version uses `new` to dynamically allocate memory. Timing in C++ is handled by `<chrono>`, specifically using `std::chrono::high_resolution_clock::now()`. The time difference between allocation (`new`) and release (`delete`) gives the allocation duration. `delete` is used after each allocation to prevent memory leaks. This setup allows an accurate, comparable allocation measurement between languages.
    - **Java Implementation**: In Java, memory allocation is managed by the JVM. Here, `System.nanoTime()` provides a high-precision timestamp immediately before and after creating an integer array. This function mimics C/C++ manual allocation by creating an array of integers. Unlike C/C++, where the developer has control over deallocation, Java relies on garbage collection. By isolating this allocation process, the function allows a direct comparison with C/C++.

5.1.2. Memory Deallocation

- **Function Name**: `measureMemoryDeallocation`
- **Purpose**: Measures the time taken to free or deallocate previously allocated memory.
- **Implementation Details**:
    - **C Implementation**: In C, `free` is used to release dynamically allocated memory. Timing starts immediately before calling `free` and stops immediately after, with `clock_gettime()` capturing the duration in nanoseconds. This duration represents the deallocation time, which can be directly compared to Java's garbage collection.
    - **C++ Implementation**: C++ uses `delete` to deallocate memory, following the same timing logic as in C. The `<chrono>` library calculates the deallocation time, providing a comparable duration by timing the difference between allocation and deallocation.
    - **Java Implementation**: Java does not allow explicit memory deallocation, as memory is automatically managed by the JVM's garbage collector. However, `System.gc()` is called to request garbage collection, though it may not trigger immediately. `System.nanoTime()` measures around the garbage collection request to approximate memory cleanup time in Java. This is not an exact deallocation time but provides a rough comparison with manual deallocation in C/C++.

5.1.3. Memory Access (Static and Dynamic)

- **Function Names**: `measureStaticMemoryAccess`, `measureDynamicMemoryAccess`
- **Purpose**: Measures the time taken to access elements in statically and dynamically allocated arrays.
- **Implementation Details**:
    - **Static Access (C/C++)**: Static arrays are allocated at compile-time. In the function, an array is accessed sequentially within a loop that sums up its elements. Timing is recorded around this loop using `clock_gettime()` (C) or `<chrono>` (C++), and the total time is divided by the number of accesses to calculate an average per-access time. For large static arrays, `ulimit -s` is used in Linux to increase stack size, ensuring the array can be stored without segmentation faults.
    - **Dynamic Access (C/C++)**: Dynamic arrays are allocated with `malloc` in C or `new` in C++. After allocation, a loop initializes the array, then a second loop accesses and sums its elements. Timing starts just before the access loop and stops immediately after. Using `clock_gettime()` in C and `<chrono>` in C++ captures the access time overhead specific to heap-allocated memory.
    - **Java Implementation**: Java arrays are managed within the heap, whether declared final (static) or not. Arrays are accessed in a loop where each element is added to a sum variable. Timing starts and stops around this loop with `System.nanoTime()`, yielding a comparable access time metric. The final modifier in Java prevents size changes at runtime, aligning a bit with C/C++ static arrays.

5.2. Thread Management Functions

Thread management functions measure the performance of thread creation, context switching, and thread migration, demonstrating each language's concurrent capabilities.

5.2.1. Thread Creation Time

- **Function Name**: `measureThreadCreationTime`
- **Purpose**: Measures the time required to create and join a thread.
- **Implementation Details**:
    - **C Implementation**: Threads in C are created with `pthread_create`. The function starts timing just before `pthread_create` and stops right before `pthread_join`, which blocks until the thread finishes. This timing includes the overhead of thread initialization and creation.
    - **C++ Implementation**: C++ uses the `<thread>` library to handle threading. A new `std::thread` object is instantiated, and timing is captured before calling `start()` and right before `join()`, which joins the thread with the main execution.
    - **Java Implementation**: In Java, a `Thread` object is created, and timing is captured with `System.nanoTime()` around `start()` and `join()`. This function measures Java's thread creation efficiency in comparison to C/C++.

5.2.2. Context Switch Time

- **Function Name**: `measureContextSwitchTime`
- **Purpose**: Measures the average time taken for a context switch between two threads.
- **Implementation Details**:

- **C Implementation**: Context switching is achieved by creating two threads synchronized using a mutex and condition variable. Each thread alternates control, signaling the other to run. A shared atomic counter increments with each switch, and timing is recorded between the first and last switch. The total time is divided by the number of switches to calculate an average switch time.
- **C++ Implementation**: Using `<mutex>` and `<condition_variable>`, context switching in C++ is synchronized similarly to C, using a shared boolean flag. An atomic counter records the number of switches, and the total duration is divided by this counter to yield the average context switch time.
- **Java Implementation**: In Java, `ReentrantLock` and `Condition` classes synchronize two threads for context switching. Timing is recorded at each switch using `System.nanoTime()`, and the final time is divided by the total switches to give an average switch time. Two functions are needed: one for each thread that alternately signals the other, evaluating Java's context switching in comparison to pthreads.

5.2.3. Thread Migration Time

- **Function Name**: `measureThreadMigrationTime`
- **Purpose**: Measures the time required to migrate a thread between CPU cores.
- **Implementation Details**:
  - **C Implementation**: Migration is implemented using `pthread_setaffinity_np`, which binds the thread to a specific CPU core. Timing starts right after setting the initial affinity and stops after switching to another core. This captures the migration time.
  - **Java Implementation (JNI Integration)**: Java lacks a built-in method for setting thread affinity. A native function is created with JNI to handle affinity setting in C, using `pthread_setaffinity_np`. The JNI function follows naming conventions (`Java_ClassName_MethodName`) and includes `JNIEXPORT` and `JNICALL` specifiers to ensure compatibility. A `.h` file is generated for the C library, and `.so` file is compiled for use in Java. The JNI function is invoked through Java using System.loadLibrary() to load the C library (e.g., .so or .dll (for windows) file) into the JVM.

5.3. Data Storage Functions

Data storage functions ensure benchmarking results are structured and saved in JSON format, making them accessible for further analysis.

5.3.1. Saving Results to JSON

- **Function Name**: `saveResultsToJSON`
- **Purpose**: Saves benchmark results (e.g., average time, standard deviation) in JSON files, including metadata.

Implementation Details:

- **Metadata:**
  - Metadata includes:
    - **process_measured**: Name or description of the process being benchmarked (e.g., memory allocation).

        - **number_of_tests**: Total number of tests conducted.
        - **passed_tests**: Number of tests that passed.
        - **outlier_threshold**: Threshold for outlier detection in the results.
        - **programming_language**: The language used for the benchmark (e.g., C, C++, Java).
        - **array_size**: Size of the array used in the benchmark (if applicable).
        - **average_time**: The average execution time across tests.
        - **std_deviation**: Standard deviation of the times measured.

- **C Implementation**:
    - The results (such as `average_time`, `std_deviation`, `number_of_tests`, `passed_tests`, etc.) are stored in a C JSON object.
    - Data is written to a file using `fprintf` to ensure that the JSON format is correctly structured. If the file already exists, the new results are appended; otherwise, a new file is created. If isFirstEntry is false, a comma is added before appending the next JSON object to the file. (the array format is hardcoded, in main function the [ ] are appended to the start and end of the file) .
- **C++ Implementation**:
    - The C++ implementation uses the `nlohmann::json` library (referred to as `ordered_json` in the provided code) to create structured JSON objects. It reads an existing JSON file (if available) and appends the new results in an ordered manner keeping the order of insertion and not the alphabetical order. The result object is added to the existing JSON array (json.push_back(result)), whether the file was previously empty or not.
    - The function checks if the JSON file exists. If it does, the data is read and parsed into a JSON object. If not, an empty array is initialized. Then, the new benchmarking data is appended to this JSON object (array), and the file is saved using `std::ofstream`.
- **Java Implementation**:
    - In Java, the implementation makes use of `org.json.JSONObject` and `org.json.JSONArray` to create structured JSON objects.
    - The function creates a `JSONObject` for each benchmark result, including the metadata.
    - Results are saved in individual JSON files (e.g., `static_access.json`, `dynamic_access.json`) using the `FileWriter` class. These files are organized by process type (static or dynamic memory access), and each file contains the benchmark results in a structured format that makes it easy to retrieve and analyze later.

5.3.2. Combining JSON Files

- **Function Name**: `combineJSONFiles`
- **Purpose**: Consolidates multiple JSON files into a single output file for unified analysis.

Implementation Details:

- **C Implementation**:
    - The C implementation would require reading each input JSON file (using a JSON parsing library like `cJSON`) and extracting the contents.

    - The contents from each file are read into a C data structure (an array of JSON objects).
    - After all files are processed, the combined results are written to an output file using `fprintf`. This file will contain all the results from the individual files in a unified JSON array.
- **C++ Implementation**:
    - The C++ version follows a similar approach to the C implementation but uses the `nlohmann::json` library for easier handling of JSON data.
    - It iterates through each input file (`filenames`), reads the contents into a `ordered_json` object, and appends the entries from each file into the `combinedResults` array.
    - The `ordered_json::array()` is used to store the combined results. After reading and appending all the entries from the files, the combined data is written into a new JSON file (`outputFilename`), formatted for easy readability using the `dump(4)` method.
- **Java Implementation**:
    - The Java implementation reads each individual JSON file using `FileReader` and `org.json.JSONTokener`.
    - For each file, it parses the content into a `JSONArray`. Each element of the array represents an individual result entry.
    - After reading all files, the entries from all the `JSONArray` objects are merged into a single `JSONArray` that holds all benchmark results.
    - The combined results are then saved into a new output file (`results.json`) using `FileWriter` and the `toString(4)` method to pretty-print the JSON with an indentation of 4 spaces.

How it combines the files:

- For each file in the list of input filenames:
    - The file is opened and read into a `JSONArray`.
    - Each entry (a `JSONObject`) from the input file is added to the `combinedResults` array.
    - After all files are processed, the combined array is dumped into a new output file.
- The resulting `results.json` file contains a complete, unified dataset, with each benchmark result from every input file included as individual entries in the array. This consolidated file is used by the `GraphingModule` to generate visual comparisons of the benchmark data.

5.4. Statistical Analysis Functions

Statistical functions calculate averages, standard deviations, and remove outliers, providing reliable data for performance comparisons.

5.4.1. Average Calculation

- **Function Name**: `calculateAverage`
- **Purpose**: Computes the mean time across multiple test iterations.
- **Implementation Details**:

- **Implementation in All Languages**: An array of recorded times is summed, then divided by the array's length to get the mean. This simple average smooths out minor fluctuations, yielding a stable metric for each process measured.

5.4.2. Standard Deviation Calculation

- **Function Name**: `calculateStandardDeviation`
- **Purpose**: Determines variability in time measurements, providing insight into performance consistency.
- **Implementation Details**:
  - **Implementation in All Languages**: Using the mean as a baseline, each value's deviation from the mean is squared and averaged. The square root of this average yields the standard deviation, indicating whether timing is consistent or varies widely due to factors like system load.

5.4.3. Outlier Removal

- **Function Name**: `removeOutliers`
- **Purpose**: Filters out extreme values to prevent skewing of results.
- **Implementation Details**:
  - **Implementation in All Languages**: After calculating the mean and standard deviation, values beyond a specified threshold from the mean are removed from the data set. This ensures anomalies like sudden system interruptions do not distort the benchmark results.

5.5.JNI Integration for Benchmark Invocation

To facilitate the integration of native C and C++ benchmarking modules with Java, **JNI (Java Native Interface)** is used. The following functions allow seamless communication between Java and native code to trigger specific benchmark operations based on user input.

5.5.1. C JNI Interface

- **Function Name:** Java_JNInterface_callNative_1C_1Benchmark
- **Purpose:** Provides a JNI function to invoke specific benchmarks in the native C library based on user-specified types.
- **Implementation Details:**

- **Parameters:**
  - benchmarkType: Integer identifier for the benchmark type (e.g., static access, dynamic access, allocation, etc.).
  - numTests: Number of test iterations to perform.
  - threshold: Threshold for outlier removal.
- **Behavior:**
  - Uses a switch statement to determine the benchmark type and calls the corresponding native function.

- Handles invalid benchmark types with an error message and terminates the process.

5.5.2. C++ JNI Interface

- **Function Name:** `Java_JNInterface_callNative_1Cpp_1Benchmark`
- **Purpose:** Provides a JNI function to invoke benchmarks in the native C++ library.
- **Implementation Details:**
  - Follows a similar approach to the C JNI function but uses `std::cerr` for error handling and `std::exit()` to terminate on invalid types.
  - Matches the integer identifiers for consistency across the project.

5.5.3. Java Benchmark Invocation

- **Function Name:** `callJavaBenchmark`
- **Purpose:** Acts as a dispatcher for Java-native benchmark functions, allowing execution of individual benchmarks or all tests collectively.
- **Implementation Details:**

- **Parameters:**
  - `benchmarkType`: Integer identifier for benchmark selection.
  - `numTests`: Number of iterations for the benchmark.
  - `threshold`: Outlier removal threshold.
- **Behavior:**
  - Uses a `switch` statement to determine and invoke the appropriate Java method for the specified benchmark.

5.5.4. Unified Benchmark Dispatcher

- The **`runBenchmark`** method orchestrates benchmark invocations across languages using the JNI interface and native Java functions.

- **Inputs:**
  - `language`: Specifies the programming language (`C`, `C++`, or `Java`).
  - `benchmarkType`: Identifies the specific benchmark (e.g., memory allocation, thread migration).
  - `numTests`: Number of iterations to perform.
  - `threshold`: Outlier removal threshold.
- **Output:** Executes the appropriate benchmarking function based on the language and type.

5.5.5. JNInterface Class

- The **JNInterface** class defines native methods and ensures the required libraries are loaded at runtime.

- The **System.loadLibrary** calls load the native C and C++ libraries.
- Native methods callNative_C_Benchmark and callNative_Cpp_Benchmark bridge Java with the corresponding native functions, ensuring consistency in input parameters and invocation logic.

5.6.Result processing

The implementation is designed to handle the storage, retrieval, and representation of benchmarking results in a structured manner/using graphs. It uses two primary classes:

- **BenchmarkResult**:
  - Represents a single benchmarking result with relevant details, such as array size, iterations, number of tests, and performance statistics.
  - Encapsulates the data through fields, a constructor, and getter methods.
  - Includes a toString method for human-readable representation.
- **BenchmarkStorage**:
  - Manages a collection of BenchmarkResult objects.
  - Handles loading benchmarking results from a JSON file into memory.
  - Provides methods to access and display the loaded results.

5.6.1. BenchmarkResult Class

- **Purpose**
  To model a single benchmark test result with details about the parameters and outcomes.
- **Key Components**

1. **Fields**:
   - **arraySize**: Size of the array used during benchmarking (relevant for memory-related tests).
   - **iterations**: Number of iterations used in the test.
   - **numberOfTests**: Total number of tests conducted.
   - **passedTests**: Number of tests that passed (within acceptable thresholds).
   - **outlierThreshold**: Threshold for determining outliers in test results.
   - **programmingLanguage**: Programming language used for the test.
   - **processMeasured**: The specific process being benchmarked (e.g., memory allocation, thread creation).
   - **averageTime**: Average execution time of the tests.
   - **stdDeviation**: Standard deviation of execution times.
2. **Constructor**:
   - Initializes all fields of the class with provided values.
3. **Getters**:
   - Provide access to individual fields.
4. **toString Method**:
   - Returns a formatted string representation of the benchmark result for easy display.

5.6.2. BenchmarkStorage Class

- **Purpose**

  To manage and operate on a collection of benchmarking results.

- **Key Components**

1. **Field**:
   - **results**: A list (ArrayList) to store multiple BenchmarkResult objects.
2. **Constructor**:
   - Initializes the results list.
3. **loadFromJsonFile Method**:
   - Loads benchmarking results from a JSON file and populates the results list.
   - Parses the JSON file using the org.json library.
   - Handles missing or optional fields using optInt and similar methods.
   - **Error Handling**:
   - Catches IOException for file reading errors and logs an error message.
4. **getResults Method**:
   - Returns the list of all loaded benchmark results.
5. **printResults Method**:
   - Iterates over the results list and prints each BenchmarkResult using its toString method.
6. **Error Handling**

   - If the specified JSON file path is incorrect, an error message is logged: *Error reading JSON file: <message>*.
   - Optional fields like array_size or iterations default to 0 if missing.

5.6.3. GraphGenerator

The GraphGenerator class generates visual representations (charts) of the benchmark results using the JFreeChart library. It supports line and bar charts with user interactivity and export functionality.

- **Key Features**

  - **Line and Bar Charts:** Generates two types of charts for performance comparison.
  - **Dynamic Filtering:** Allows users to filter the results by programming language or adjust the value range.
  - **Interactive GUI:** Includes components like combo boxes, checkboxes, sliders, and buttons for user interaction.
  - **Export Options:** Supports exporting charts as PNG images or datasets as CSV files.

- **Key Methods**

1. **createAndShowGraph(String measurement, BenchmarkStorage benchmarkStorage):**
   - Generates and displays the graphs for a specified performance measurement.
   - Integrates line and bar charts into an interactive GUI.

2. **createDataset(String measurement, BenchmarkStorage benchmarkStorage, boolean showStandardDeviation):**
    - Creates a dataset for the specified measurement.
    - Includes the ability to display either average time or standard deviation.
3. **createLineChart(DefaultCategoryDataset dataset, String measurement):**
    - Constructs a line chart using the provided dataset.
    - Highlights series points and adds interactivity through tooltips.
4. **createBarChart(DefaultCategoryDataset dataset, String measurement):**
    - Constructs a bar chart using the provided dataset.
    - Includes tooltips with detailed information for each bar.
5. **styleChart(JFreeChart chart, CategoryPlot plot):**
    - Styles the charts with custom fonts, gradient backgrounds, and gridline colors for better readability.
6. **displayChart(JFreeChart lineChart, JFreeChart barChart, DefaultCategoryDataset dataset, String measurement, BenchmarkStorage benchmarkStorage):**
    - Displays the charts in a JFrame with options to toggle between chart types, filter data, and adjust the range.
    - Includes export functionality for saving the charts or datasets.

- **Interactive Components**

    - **Chart Type Selector:** Allows toggling between line and bar charts.
    - **Metric Selector:** Switches between average time and standard deviation views.
    - **Language Filters:** Enables/disables specific programming languages for comparison.
    - **Range Slider:** Adjusts the visible range of the Y-axis.
    - **Export Button:** Saves the chart as a PNG or the dataset as a CSV file.

- **Libraries Used**

    - **JFreeChart:** For creating and customizing charts.
    - **Swing:** For building the interactive GUI.
    - **Java IO:** For reading/writing JSON, PNG, and CSV files.

- **Workflow Summary**

1. **Data Loading:**
    - JSON files containing benchmark results are loaded using BenchmarkStorage.loadFromJsonFile.
    - Results are stored as BenchmarkResult objects in a list.
2. **Dataset Preparation:**
    - GraphGenerator.createDataset processes the loaded results into a JFreeChart-compatible dataset.
3. **Chart Creation:**
    - Line and bar charts are created based on the dataset and styled for clarity.
4. **Visualization and Interaction:**
    - Charts are displayed in an interactive GUI with filters, sliders, and export options.
5. **Export Functionality:**
    - Users can save charts as PNG images or export datasets as CSV files for further analysis

5.7.Controller class

The Controller class acts as the central manager for the benchmarking application, integrating the execution of benchmarks, storage of results, and graph generation. It connects the core components (BenchmarkEngine, BenchmarkStorage, and GraphGenerator) to provide a cohesive user experience. Below is a detailed breakdown of the class:

Class Overview

- **Name:** `Controller`
- **Purpose:**
  - To orchestrate the execution of benchmarks.
  - To manage result loading and storage.
  - To handle graph generation for visualizing benchmark results.

Attributes

1. **`benchmarkEngine`**
   - **Type:** `BenchmarkEngine`
   - **Description:** Responsible for executing the benchmarks for different languages and measurement types.
2. **`benchmarkStorage`**
   - **Type:** `BenchmarkStorage`
   - **Description:** Handles storage, retrieval, and manipulation of benchmark results.
3. **`graphGenerator`**
   - **Type:** `GraphGenerator`
   - **Description:** Generates and displays graphs based on stored benchmark results.

Constructors

1. **`Controller()`**
   - **Description:** Initializes the controller by creating instances of `BenchmarkEngine`, `BenchmarkStorage`, and `GraphGenerator`.

Methods

1. **`runBenchmark(String language, int benchmarkType, int numTests, double threshold)`**
   - **Description:** Executes a benchmark for a specified language and measurement type.
   - **Parameters:**
     - `language` - The programming language to benchmark (e.g., "C", "Java").
     - `benchmarkType` - The type of benchmark (e.g., memory allocation, thread creation).
     - `numTests` - Number of iterations to execute.

        - `threshold` - Threshold for excluding outlier data points.
    - **Throws:** `InterruptedException` if the benchmarking process is interrupted.
2. **`loadResults(String language, int benchmarkType)`**
    - **Description:** Loads benchmark results from a JSON file into the `BenchmarkStorage` object.
    - **Parameters:**
        - `language` - The programming language whose results should be loaded.
        - `benchmarkType` - The type of benchmark whose results should be loaded.
    - **Behavior:**
        - Retrieves the appropriate file path using `getBenchmarkFilePath`.
        - Loads results into the `BenchmarkStorage`.
        - Outputs an error message if the benchmark type is invalid.
3. **`getBenchmarkFilePath(String language, int benchmarkType)`**
    - **Description:** Generates the file path for a given language and benchmark type.
    - **Parameters:**
        - `language` - The programming language of the benchmark.
        - `benchmarkType` - The benchmark type.
    - **Returns:** The file path as a `String`, or `null` if the benchmark type is invalid.
    - **Example Paths:**
        - `"Java_measurements/Java_results.json"`
        - `"C_measurements/C_thread_creation.json"`
4. **`generateGraph(String measurement)`**
    - **Description:** Generates and displays a graph for a specific measurement (e.g., memory allocation, context switching).
    - **Parameters:**
        - `measurement` - The name of the measurement to visualize (e.g., "Memory Allocation", "Thread Migration").
    - **Behavior:** Uses the `GraphGenerator` to create and display the graph.
5. **`getBenchmarkStorage()`**
    - **Description:** Provides access to the current `BenchmarkStorage` object.
    - **Returns:** An instance of `BenchmarkStorage`.
6. **`printResults()`**
    - **Description:** Prints the results stored in `BenchmarkStorage` to the console.
    - **Use Case:** For debugging or quick inspection of loaded benchmark data.

Integration with Other Components

- **`BenchmarkEngine`:** Handles the execution of benchmarks, allowing the controller to specify parameters such as language and measurement type.
- **`BenchmarkStorage`:** Stores and retrieves benchmark results in JSON format, enabling seamless integration with the graph generator.
- **`GraphGenerator`:** Creates visual representations of the stored data, providing meaningful insights into performance.

Advantages of the `Controller` Class

- **Centralized Management:** Provides a single entry point for all operations.

- **Extensibility:** Easy to add support for new languages or benchmarks by updating file paths or benchmark types.
- **Integration:** Smooth coordination between execution, storage, and visualization components.

This comprehensive design ensures that the benchmarking application is modular, reusable, and user-friendly.

5.8.GUI class

The MainGUI class is a graphical user interface (GUI) for managing and executing benchmarking operations. It interacts with the Controller class to execute benchmarks, manage results, and generate visualizations.

Class Overview

Name: `MainGUI`

Purpose:

- Provides a user-friendly interface for configuring and running benchmarks.
- Displays real-time logs and progress updates.
- Integrates with the `Controller` to execute benchmarks and load or visualize results.

Attributes:

1. **`controller`**: An instance of the `Controller` class to handle core operations.
2. **`logArea`**: A `JTextArea` to display logs and feedback during benchmarking.
3. **`numTestsField`**: A `JTextField` for inputting the number of test iterations.
4. **`thresholdField`**: A `JTextField` for inputting the threshold for outlier detection.
5. **`progressBar`**: A `JProgressBar` to indicate benchmarking progress visually.
6. **`benchmarkExplanations`**: A `Map<String, String>` storing HTML-based explanations for benchmarks.
7. **`isBenchmarkInProgress`**: A `boolean` flag to prevent multiple benchmarks from running simultaneously.

Key Methods

1. Constructor

```
public MainGUI()
```

- Initializes the `Controller` and loads explanations for benchmarks.
- Calls `setupUI()` to configure the GUI layout and components.

2. User Interface Components

```
private void setupUI()
```

- Sets up the main layout using a `BorderLayout`.

- Divides the GUI into:
    - **Input Panel:** Includes fields for the number of tests and threshold, along with reset and help buttons.
    - **Tabbed Pane:** Contains tabs for each benchmark type.
    - **Log Panel:** Displays logs and a progress bar for real-time updates.
- Applies modern styling with gradients and consistent fonts.

```
private JTabbedPane createTabbedPane()
```

- Creates a tabbed pane for navigating benchmarks.
- Adds hover effects for a dynamic user experience.

```
private JButton createStyledButton(String text, Color backgroundColor)
```

- Creates styled buttons with hover effects and a uniform design.

3. Benchmark Execution

```
private void startBenchmark(String benchmark)
```

- Validates that no other benchmarks are in progress.
- Uses a `SwingWorker` to execute the selected benchmark asynchronously.
- Calls `runBenchmarkForAllLanguages()` to execute benchmarks across all languages.

```
private void runBenchmarkForAllLanguages(String benchmark, int numTests, double
threshold)
```

- Runs the benchmark for `C`, `C++`, and `Java`.
- Updates progress dynamically and generates graphs once the benchmark completes.
- Handles both individual benchmarks and the "All Benchmarks" option.

```
private int getBenchmarkType(String benchmarkName)
```

- Maps benchmark names to corresponding integer identifiers.

4. Benchmark Explanations

```
private Map<String, String> loadBenchmarkExplanations()
```

- Loads HTML-formatted explanations for each benchmark type.
- Provides descriptions, comparisons, and key takeaways for users.

```
private void openExplanationPage(String benchmark)
```

- Displays a new window with detailed information about the selected benchmark.
- Includes notes and a "Start Benchmark" button for quick execution.

5. Utility Functions

```
private void resetInputs()
```

- Resets the input fields, log area, and progress bar to their default states.

`private void showHelpDialog()`

- Displays a help dialog explaining the input parameters (e.g., `numTests`, `threshold`).

Integration with `Controller`

- **Running Benchmarks:** Calls `controller.runBenchmark(language, benchmarkType, numTests, threshold)` to execute benchmarks.
- **Loading Results:** Uses `controller.loadResults(language, benchmarkType)` to retrieve benchmark data.
- **Graph Generation:** Calls `controller.generateGraph(measurement)` to display results visually.

5.9.Main Menu Pages

Main Menu Overview

The **Main Menu** of the benchmarking application provides users with an intuitive interface for configuring and executing benchmarks. As shown in the first image, the layout consists of the following key components:

- **Benchmark Settings Panel**: Allows users to input the number of tests and the threshold for outlier detection. It also includes reset and help buttons for additional guidance.
- **Tabbed Navigation**: Users can switch between different types of benchmarks, such as memory allocation, thread creation, and context switching, using the tabbed interface.
- **Logs Panel**: Displays real-time logs of benchmark execution and a progress bar to indicate task completion.
- **Main Display Area**: Displays the benchmark data or visualizations based on user selections.

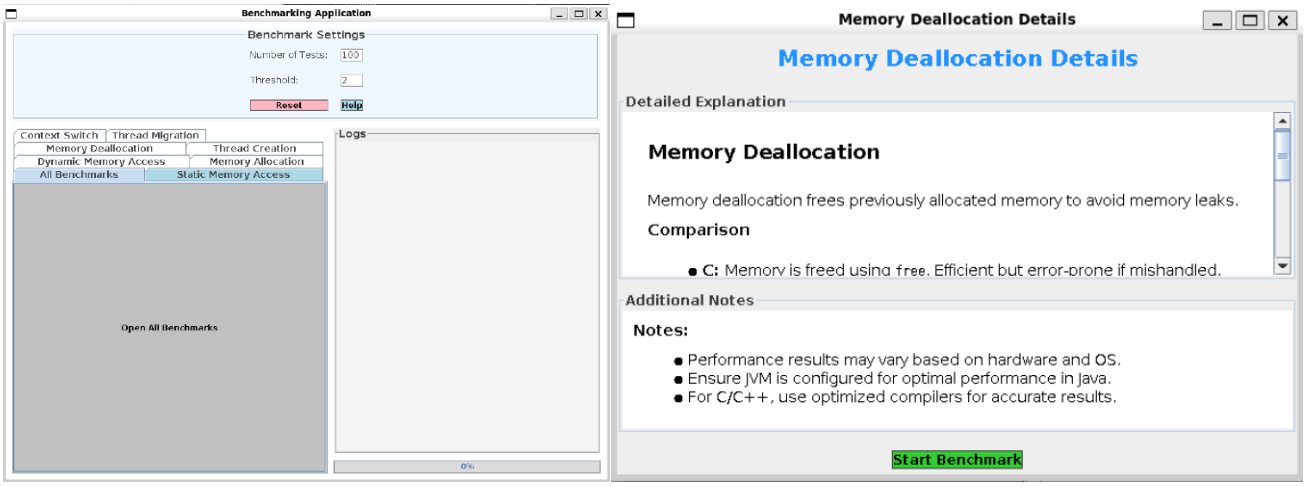
This layout ensures that all key functionalities are easily accessible and logically organized for efficient user interaction.

Benchmark Details

The second image highlights the **Benchmark Details** page, which provides a detailed explanation of a specific benchmark. Key features include:

- **Detailed Explanation Section**: Offers a concise description of the benchmark process, its purpose, and comparisons across programming languages.
- **Additional Notes Section**: Includes important tips and considerations, such as hardware and OS impact, JVM configuration for Java, and compiler optimizations for C/C++.
- **Action Button**: The "Start Benchmark" button allows users to quickly execute the selected benchmark directly from this view.

This page enhances user understanding by presenting detailed contextual information alongside the ability to perform specific actions.



6. Testing and Validation

6.1. Overview

The objective of testing and validation was to comprehensively evaluate the performance of critical computational processes across three programming languages: C, C++, and Java. These processes include memory management, thread creation, context switching, and thread migration. The tests covered various array sizes and thread counts to ensure accurate benchmarking and meaningful performance comparisons.

6.2. Test Cases

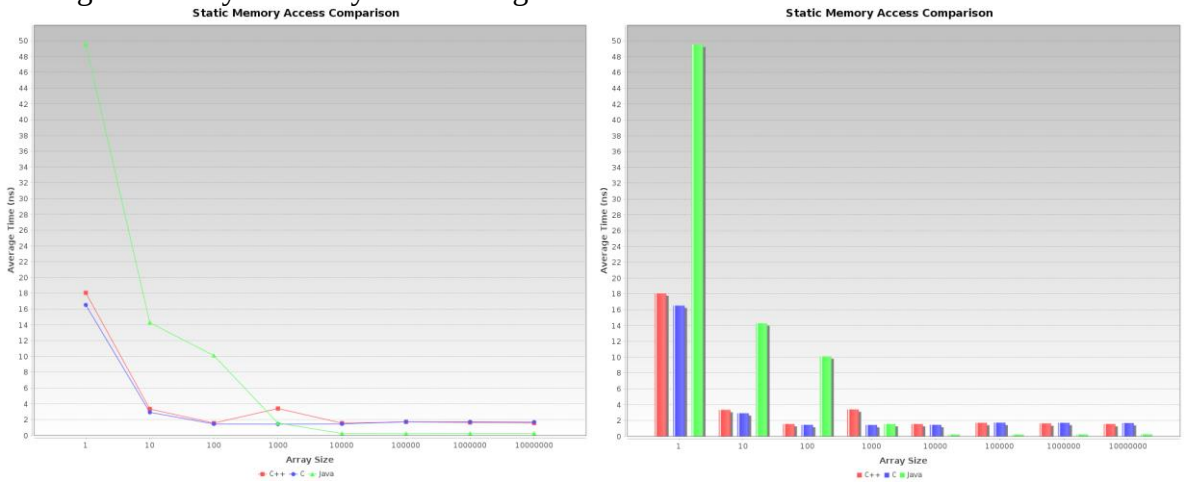
Memory Management

6.2.1. Static Memory Access

- **Objective:** Measure access times for statically allocated memory across different array sizes.
- **Expected Results:** Static memory access times should decrease as array size increases due to increased memory locality and optimized cache use.
- **Results Given by the System:**
  - **C:**
    - The average access time decreased from 16.70 ns for array size 1 to 1.8 ns for array size 10,000,000.
    - Standard deviations were minimal, indicating stable and predictable memory access performance.
    - Interpretation: The decreasing trend demonstrates that C efficiently leverages memory locality and caching mechanisms.
  - **C++:**
    - Access times ranged from 18 ns for size 1 to 1.5 ns for size 10,000,000.
    - Consistently low standard deviations showed a predictable memory access pattern.
    - Interpretation: C++'s performance closely mirrored that of C, confirming its efficient use of low-level memory management.

- **Java:**
  - The average access time started at 48 ns for size 1 and dropped significantly to 0.21 ns for size 10,000,000.
  - Standard deviations showed reduced variability as array sizes increased.
  - Interpretation: Java's initial high access times were due to JVM initialization and runtime checks. The improved performance at larger sizes suggests JIT compilation optimizations.

- **Summary:**
  Static memory access times in C and C++ demonstrated predictable scaling due to direct memory addressing. Java showed significant performance improvements after its JIT compiler optimized repeated access patterns. The minimal standard deviations across all languages highlighted stable memory access behavior, confirming that the implementations efficiently leveraged memory hierarchy and caching mechanisms.

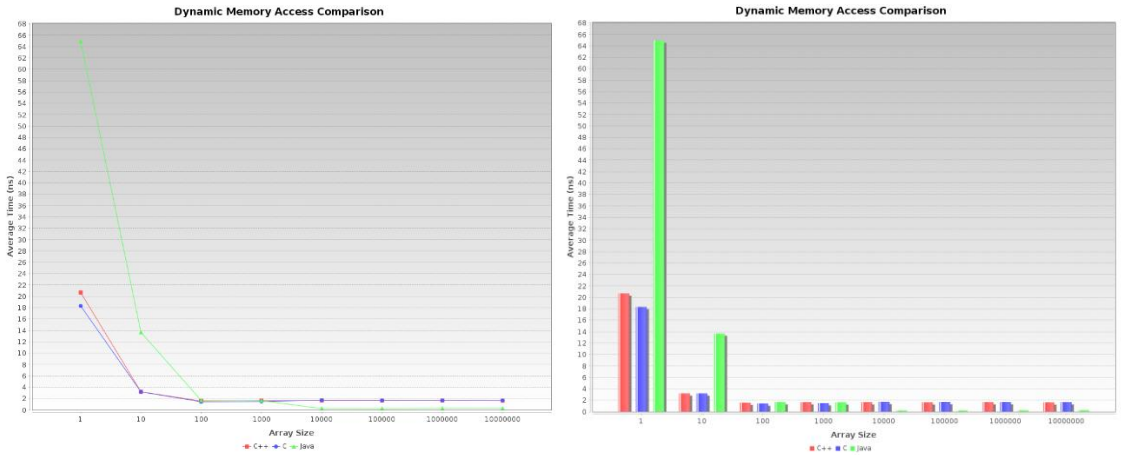


6.2.2. Dynamic Memory Access

- **Objective:** Measure access times for dynamically allocated memory across various array sizes.
- **Expected Results:** Dynamic memory access times should stabilize as array sizes increase due to reduced memory overhead and cache optimization.
- **Results Given by the System:**
  - **C:**
    - The average access time decreased from 18 ns for array size 1 to 1.8 ns for array size 10,000,000.
    - Standard deviations were low, indicating consistent memory access performance.
    - **Interpretation:** Initial overhead for smaller arrays reflects the cost of heap allocation. As array sizes increased, C efficiently leveraged memory caching and direct memory addressing.
  - **C++:**
    - Access times ranged from 21 ns for array size 1 to 1.7 ns for array size 10,000,000.
    - Standard deviations remained minimal, confirming predictable performance.
    - **Interpretation:** C++ showed comparable results to C, indicating effective use of dynamic memory management mechanisms provided by its standard library.
  - **Java:**

        - Average access times started at 65 ns for array size 1 and dropped to 0.22 ns for array size 10,000,000.
        - Standard deviations were slightly higher for smaller arrays but stabilized with larger arrays.
        - **Interpretation:** Java's initial high access times were due to JVM memory initialization and runtime checks. The improved performance at larger sizes highlights Java's Just-In-Time (JIT) compilation and memory optimization capabilities.

- **Summary:**
  Dynamic memory access times in C and C++ demonstrated direct memory management efficiency with minimal variation, reflecting well-optimized allocation and access mechanisms. Java exhibited significantly higher access times initially but improved due to JVM optimizations and adaptive memory management. These results underscore each language's respective memory management strategies and their impact on dynamic memory performance.

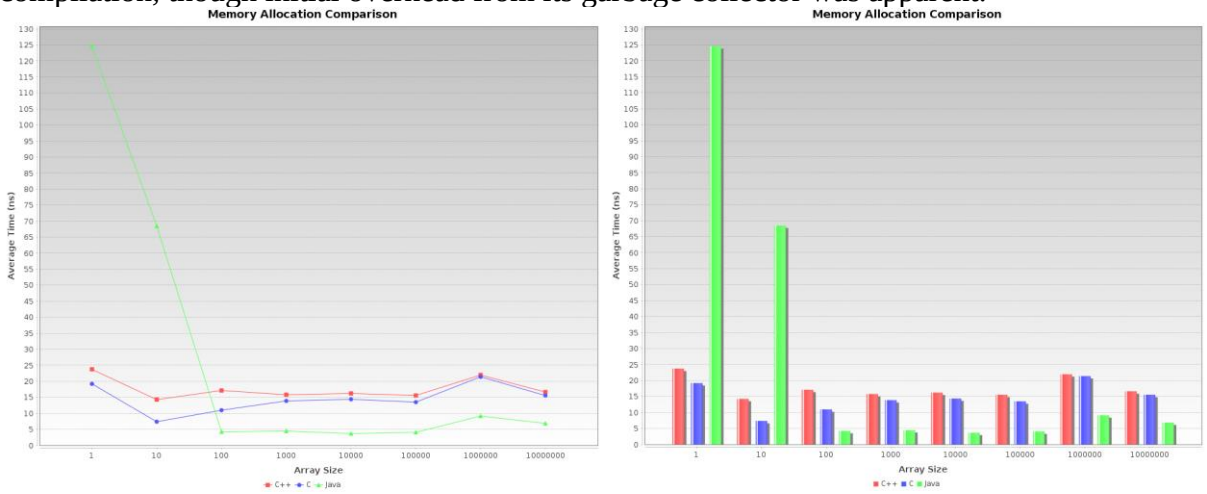


6.2.3. Memory Allocation

- **Objective:** Measure the time required to allocate memory for arrays of different sizes.
- **Expected Results:** Memory allocation times should vary depending on system-level memory management mechanisms, with potential increases due to larger memory requests.
- **Results Given by the System:**
    - **C:**
        - Allocation times ranged from 19 ns for size 1 to 15 ns for size 10,000,000.
        - Standard deviations were low, indicating stable allocation performance.
        - **Interpretation:** C demonstrated predictable allocation times due to its direct memory management model, where memory requests are fulfilled with minimal system-level intervention.
    - **C++:**
        - Times ranged from 24 ns for size 1 to 17 ns for size 10,000,000.
        - Minimal standard deviations reflected consistent performance.
        - **Interpretation:** C++'s memory allocation relied on its standard library, which introduces slight overhead compared to C but maintains efficiency for large arrays.
    - **Java:**

        - Initial allocation times were notably higher at 125 ns for size 1, dropping to 7 ns for size 10,000,000.
        - The decrease reflects the impact of JVM memory optimizations.
        - **Interpretation:** Java's garbage collection and memory initialization routines resulted in higher allocation times for small arrays. As the JVM optimized its memory management, allocation times significantly improved for larger arrays.

- **Summary:**
  Memory allocation tests highlighted key differences in how C, C++, and Java manage memory. C's direct memory management yielded consistently low allocation times. C++ introduced minimal overhead due to its memory management abstractions. Java's memory allocation improved significantly due to its adaptive memory management and JIT compilation, though initial overhead from its garbage collector was apparent.

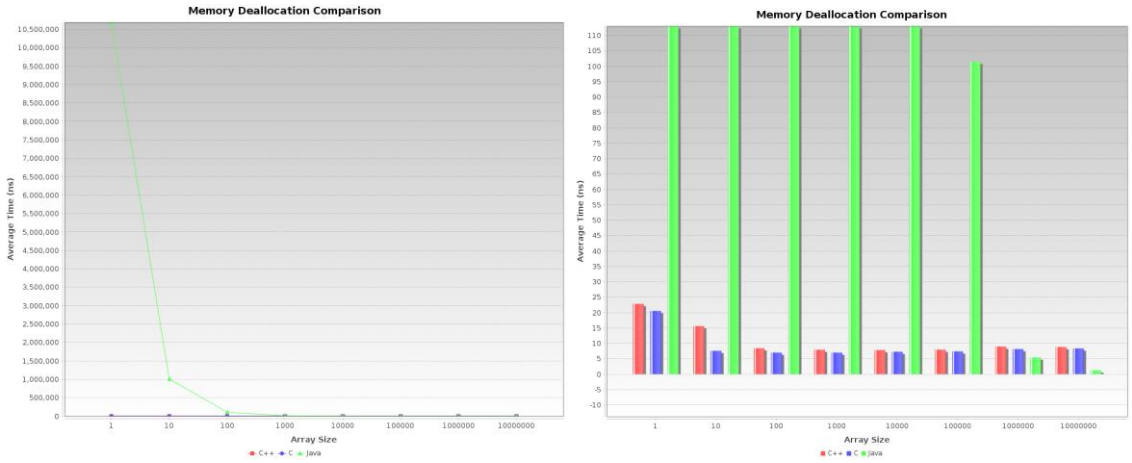


6.2.4. Memory Deallocation

- **Objective:** Measure the time required to deallocate memory for arrays of different sizes.
- **Expected Results:** Memory deallocation times should remain consistent, with minor increases for larger arrays due to system-level memory management.
- **Results Given by the System:**
    - **C:**
        - Deallocation times ranged from 21 ns for array size 1 to 8.50 ns for array size 10,000,000.
        - Standard deviations remained relatively low, indicating predictable performance.
        - **Interpretation:** C's direct memory deallocation mechanism (`free()`) performed efficiently, with only minor variations due to system memory management overhead.
    - **C++:**
        - Deallocation times ranged from 23 ns for size 1 to 8.50 ns for size 10,000,000.
        - Consistent standard deviations reflected stable deallocation times.
        - **Interpretation:** C++'s memory deallocation (`delete`) exhibited similar behavior to C, though slightly higher times due to runtime management overhead.
    - **Java:**

- Initial deallocation times were extremely high at 10,500.000 ns for size 1, reducing dramatically to 1.23 ns for size 10,000,000.
- High standard deviations at smaller sizes reflected Java's garbage collection mechanism.
- **Interpretation:** Java's garbage collection caused significant initial delays but improved substantially after memory optimization by the JVM. The results demonstrate the trade-off between managed and manual memory management systems.

- **Summary:**
  Memory deallocation tests revealed key differences in how C, C++, and Java handle memory cleanup. C and C++ provided predictable deallocation times due to manual memory management. Java, in contrast, showed highly variable results due to its garbage collection system, which caused significant delays for smaller arrays but optimized performance for larger arrays after warm-up.

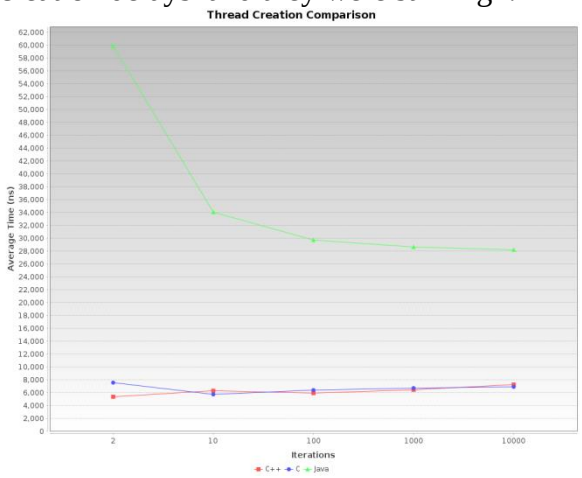


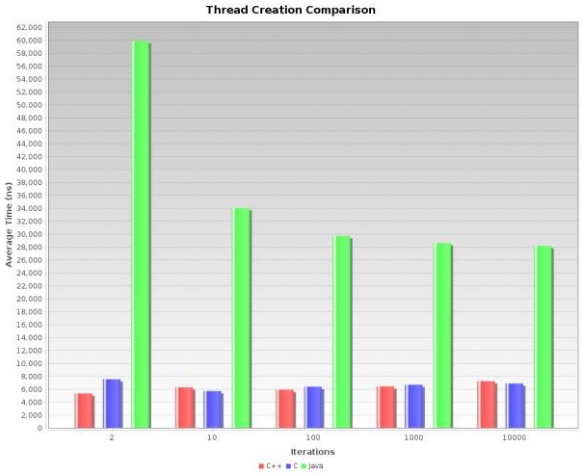
Thread Management

6.2.5. Thread Creation

- **Objective:** Measure the time required to create threads at varying iteration counts.
- **Expected Results:** Since average time per operation is calculated by dividing the total time by the number of threads created, the reported average time should remain consistent regardless of the number of threads created.
- **Results Given by the System:**
  - **C:**
    - Thread creation times ranged from 7800 ns for 2 iterations to 7,000 ns for 10,000 iterations.
    - Standard deviations were relatively stable, reflecting consistent thread creation behavior.
    - **Interpretation:** C's direct thread management led to predictable scaling, with minimal fluctuations.
  - **C++:**
    - Thread creation times started at 5,500 ns for 2 iterations and decreased to 7,700 ns for 10,000 iterations.
    - Significant variability was observed for smaller iteration counts due to initial thread creation overhead.

        - **Interpretation:** C++'s abstraction layer added some initial overhead but stabilized for larger thread counts.
    - **Java:**
        - Initial thread creation times were significantly higher at 60,000 ns for 2 iterations, decreasing to 28,000 ns for 10,000 iterations.
        - High standard deviations reflected JVM thread management overhead and garbage collection impacts.
        - **Interpretation:** Java's thread creation process was affected by JVM runtime initialization and system scheduling but improved with warm-up.
- **Summary:**
  Thread creation tests highlighted the efficiency of low-level thread management in C, the impact of C++'s abstractions, and the JVM's initial overhead in Java. While C and C++ delivered predictable results with minimal variance, Java required warm-up to reduce thread creation delays and they were still high.



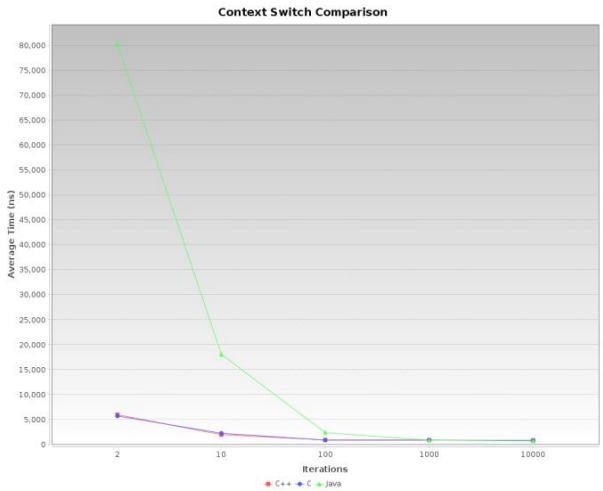


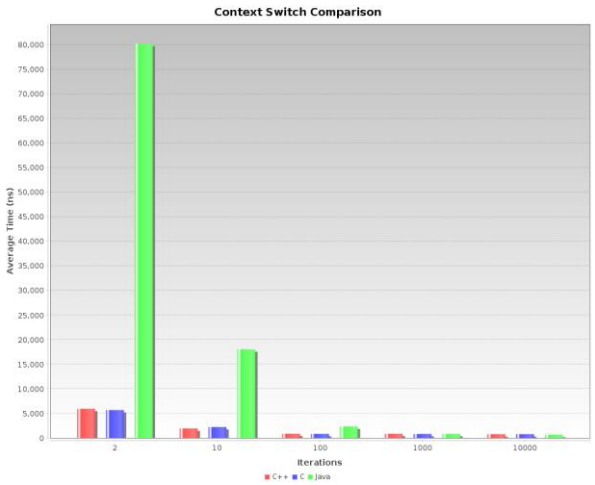
6.2.6. Context Switching

- **Objective:** Measure the time required for context switching between threads at varying iteration counts.
- **Expected Results:** Average context switching times should decrease as the number of iterations increases, reflecting efficient context switching with optimized CPU usage.
- **Results Given by the System:**
    - **C:**
        - Context switch times ranged from 5,100 ns for 2 iterations to 700 ns for 10,000 iterations.
        - Standard deviations reduced significantly as iteration counts increased.
        - **Interpretation:** C exhibited efficient context switching due to direct hardware interaction with minimal overhead.
    - **C++:**
        - Times started 5,200 ns for 2 iterations, decreasing to 680 ns for 10,000 iterations.
        - High variability at low iteration counts reflected initialization overhead.
        - **Interpretation:** C++ leveraged system threading APIs effectively, though its abstraction layer caused some added overhead.
    - **Java:**
        - Initial context switching times were extremely high at 80,000 ns for 2 iterations, decreasing to 640 ns for 10,000 iterations.
        - Standard deviations reduced progressively with higher iterations.
        - **Interpretation:** Java's initial high times were due to JVM thread management

and garbage collection. Optimization improved performance significantly after initial iterations.

- **Summary:**
  Context switching tests demonstrated C's superior performance due to its direct access to hardware-level thread management. C++ showed similar performance with added overhead from standard library abstractions. Java experienced substantial delays initially, attributable to JVM-managed threads, but improved after warm-up due to its adaptive runtime environment.

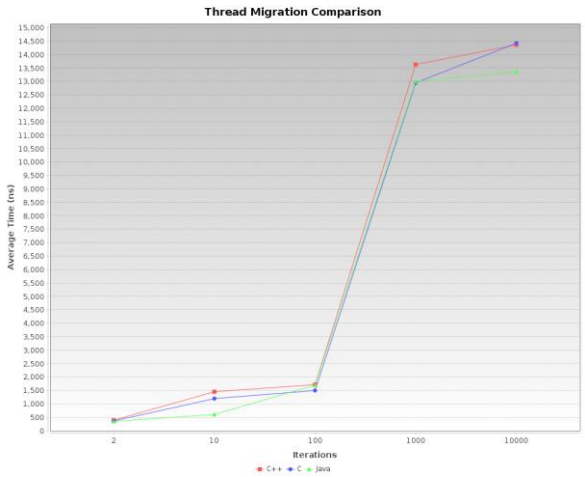


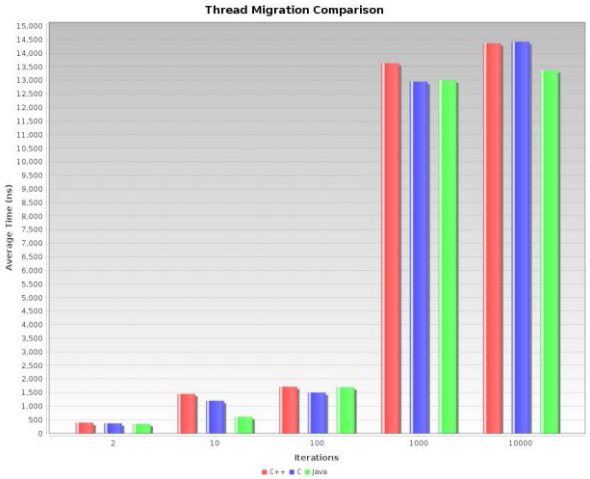


6.2.7. Thread Migration

- **Objective:** Measure the time required for migrating threads across CPU cores at varying iteration counts.
- **Expected Results:** Thread migration times should increase as the number of iterations grows, reflecting increased system-level scheduling workload.
- **Results Given by the System:**
  - **C:**
    - Migration times ranged from 380 ns for 2 iterations to 14,400 ns for 10,000 iterations.
    - Standard deviations were relatively high at lower iterations due to initialization overhead.
    - **Interpretation:** C exhibited efficient thread migration at lower counts, but increased workload from higher iterations led to longer average migration times.
  - **C++:**
    - Migration times started at 360 ns for 2 iterations and increased to 14,300 ns for 10,000 iterations.
    - Variability reduced significantly as iteration counts increased.
    - **Interpretation:** C++ leveraged system-level thread management, with thread scheduling overhead becoming more apparent as iteration counts grew.
  - **Java:**
    - Migration times started at 340 ns for 2 iterations, increasing to 13,300 ns for 10,000 iterations.
    - Standard deviations decreased consistently with higher iteration counts.
    - **Interpretation:** Java's managed thread environment faced increased scheduling and garbage collection overhead, contributing to longer average times with higher iteration counts.

- **Summary:**
  Thread migration tests revealed that migration time scales with iteration count due to increased system workload. C's low-level system access initially allowed for fast migrations but faced scaling challenges. C++ introduced moderate abstraction costs, and Java experienced significant delays due to JVM scheduling and runtime management.





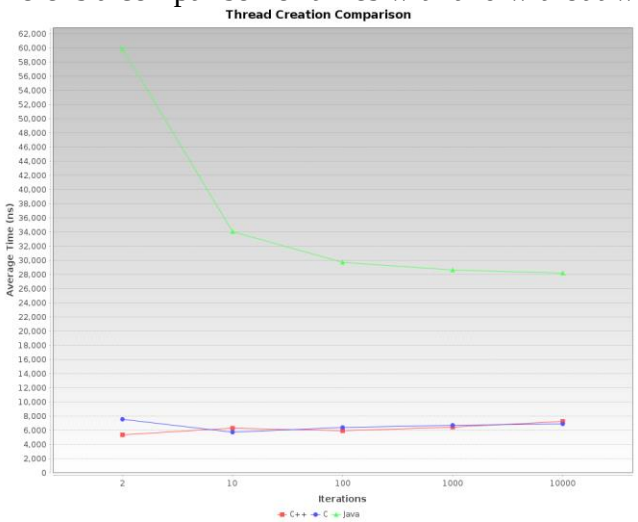
6.3. Warm-up Requirement for Thread Operations

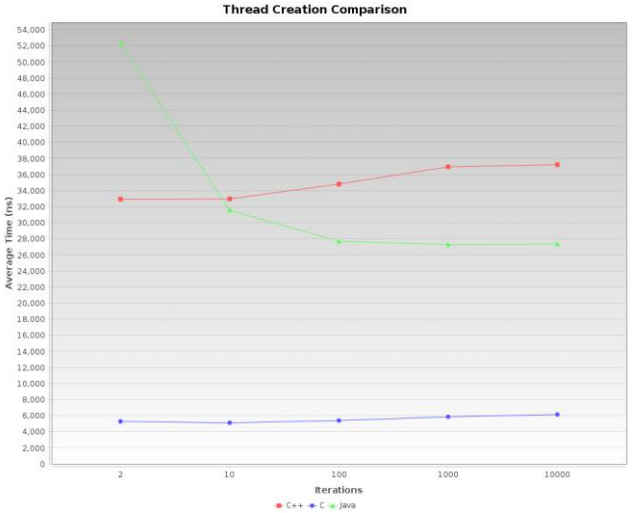
Thread operations, including creation, context switching, and migration, require a warm-up phase due to factors such as operating system scheduler initialization, runtime environment setup, and just-in-time (JIT) compilation in managed languages like Java. Without a warm-up, initial tests may reflect uncharacteristically high execution times due to these setup processes.
For example:

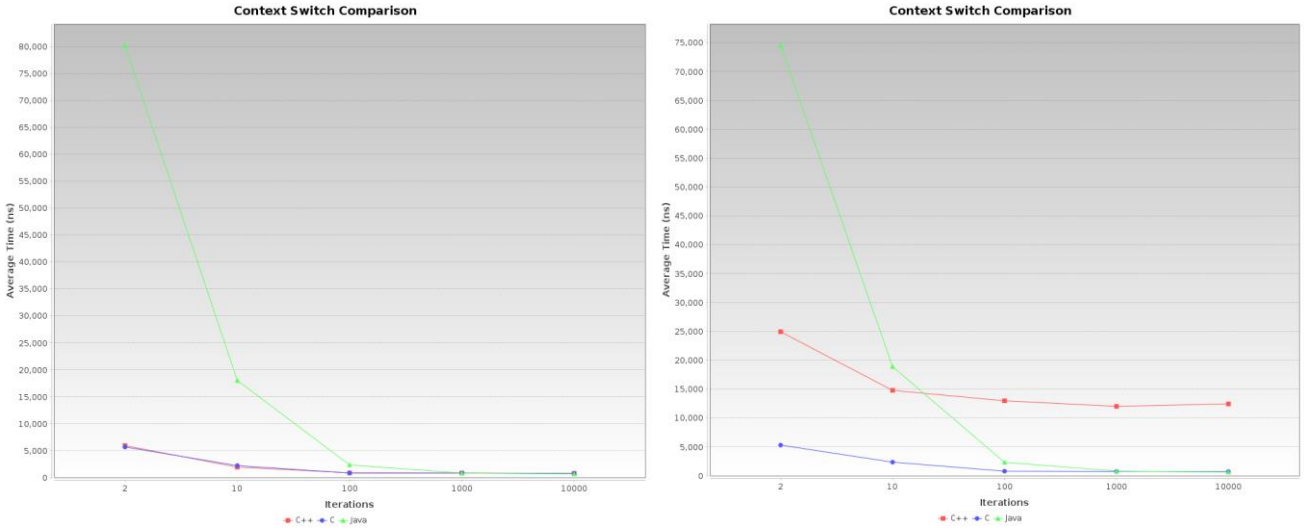
- **C:** The OS scheduler might introduce delays during the first few thread operations as thread pools and system resources are allocated.
- **C++:** Similar to C, its threading libraries involve initialization overhead.
- **Java:** The JVM must initialize thread management, trigger JIT compilation, and perform garbage collection setup, causing significant startup delays.

By conducting warm-up iterations before measurements, we ensure stable and representative test results, preventing skewed averages caused by the initial overhead.

Here is a comparison of times with and without warmup:







It can be easily seen that because C++ is first and there is no warmup, the C++ results are corrupted(right images).

6.4. Testing conclusions

The testing and validation process revealed distinct performance characteristics for C, C++, and Java. C excelled in low-level memory and thread management due to its direct interaction with hardware. C++ demonstrated efficient memory and thread handling with moderate overhead from its standard libraries. Java exhibited the most variability due to its managed runtime environment, though significant optimizations occurred after initial warm-up.
These results underline the trade-offs between low-level system control and higher-level runtime environments. Developers must consider these differences when selecting a language for performance-critical applications involving memory management and concurrent thread processing.

7. Conclusion

This project successfully developed a cross-language benchmarking tool that highlights the performance trade-offs between C, C++, and Java in memory management and thread operations. Through rigorous testing and validation, the results emphasize the strengths and weaknesses of each language:

- **C**: Exhibited consistent performance with low overhead due to its direct interaction with hardware, making it ideal for performance-critical applications requiring manual memory management.
- **C++**: Maintained similar efficiency as C while providing additional abstractions and safety features, offering a balanced solution for developers who require flexibility without sacrificing performance.
- **Java**: Showed variability in performance due to its managed runtime environment but demonstrated significant improvements after warm-up. Its garbage collection and JIT compilation features make it suitable for high-level, memory-safe applications.

These findings provide valuable insights for developers to make informed decisions based on specific application requirements. Future work may include extending the benchmarking scope to other programming languages and exploring optimization techniques within each runtime environment.

Bibliography

1. Sheldon, R. (2022). *Memory Management*. Available at: https://www.techtarget.com/whatis/definition/memory-management
2. Frank, D. (2024). *C-Memory Management & Allocation*. Available at: https://medium.com/@frankokey469/c-memory-management-allocation-99ccb4f36386
3. SoftYoi LLP. (2024). *Memory Management in Java*. Available at: https://www.linkedin.com/pulse/memory-management-java-softyoi-llp-wxrsf/
4. Bigelow, S. J. (2019). *Threads*. Available at: https://www.techtarget.com/whatis/definition/thread
5. Kumar, P. (2023). *Difference between Thread Context Switch and Process Context Switch*. Available at: https://www.tutorialspoint.com/difference-between-thread-context-switch-and-process-context-switch
6. Balakrishman, S., & Pattabiraman, K. (2013). *Migration of Threads Containing Pointers in Distributed Memory Systems*. Available at: https://pages.cs.wisc.edu/~saisanth/papers/hipc00.pdf
7. Ferreira, C. R. (2020). *8 Ways to Measure Execution Time in C/C++*. Available at: https://levelup.gitconnected.com/8-ways-to-measure-execution-time-in-c-c-48634458d0f9
8. Baeldung. (2023). *Guide to JNI (Java Native Interface)*. Available at: https://www.baeldung.com/jni
9. Barbara Illowsky & Susan Dean (2013). *Introductory Statistics.*