Performance comparison of different programming languages

(C, Java, Python)

Student: Tanase Luisa Elena

Structure of Computer Systems Project

Technical University of Cluj-Napoca

1. Introduction

1.1 Context

In modern computing, performance analysis of programming languages plays a crucial role in understanding how efficiently different languages execute tasks, such as memory management and thread handling. Given the diversity in the design and execution models of programming languages, each type presents distinct advantages and trade-offs. For example, compiled languages like C are known for their speed and control over low-level resources, while JVM-based languages like Java balance portability and performance. On the other hand, interpreted languages such as Python trade off execution speed for ease of use and flexibility. This study compares the performance characteristics of three languages: C, a compiled language; Java, a language executed on the JVM (Java Virtual Machine); and Python, an interpreted language. The analysis aims to provide insights into how the execution models of these languages affect key performance metrics.

1.2 Objectives

Measure and Compare Memory Allocation:

- Static Memory Allocation: Measure the time it takes to allocate memory at compile-time (for C) or during the program's start (for Java and Python).
- Dynamic Memory Allocation: Measure runtime memory allocation using constructs such as malloc() in C, new in Java, and list/dictionary creation in Python.

Measure and Compare Memory Access:

- Evaluate the time required to read and write data to memory in each language.

Evaluate Thread Creation and Management:

- Thread Creation Time: Compare the time it takes to spawn a new thread in each language, using libraries such as pthread in C, Java's Thread class, and Python’s threading module.
- Thread Context Switching: Measure how quickly each language can switch between threads.
- Thread Migration Time: Analyze how efficiently threads move between processor cores in a multi-core system, assessing overhead involved in thread scheduling and migration.

Compare Execution Time of Critical Sections:

- Compiled vs Interpreted vs JVM Performance: Through memory and thread benchmarks, assess how the different execution models of C (compiled), Java (JVM), and Python (interpreted) influence performance in areas such as speed, resource usage, and concurrency handling.

Analyze Results and Draw Conclusions:

- Performance Trade-offs: Highlight the trade-offs in terms of speed, efficiency, and complexity that come with using a compiled language (C), a JVM-based language (Java), and an interpreted language (Python).
- Real-World Implications: Provide insight into the suitability of each language for specific types of applications, especially those requiring high performance, such as systems programming or large-scale web services.

2. Bibliographic research

Compiled vs interpreted programming languages

- In a compiled language, the target machine directly translates the program. Compiled languages are converted directly into machine code that the processor can execute. They also give the developer more control over hardware aspects, like memory management and CPU usage. Compiled languages need a “build” step – they need to be manually compiled first.
- In an interpreted language, the source code is not directly translated by the target machine. Instead, a different program, aka the interpreter, reads and executes the code. Interpreters run through a program line by line and execute each command.
- JVM(Java Virtual Machine) runs Java applications as a run-time engine. JVM is the one that calls the main method present in a Java code. JVM is a part of JRE(Java Runtime Environment). Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and expect it to run on any other Java-enabled system without any adjustment. This is all possible because of JVM. What’s special about Java is, it uses a combination of both compilation and interpretation. So, in Java, source code is first compiled to a class file with bytecode. Then this class file is executed with the Interpreter or JIT compiler.

Processes, threads and context swithching

A thread is the smallest unit of execution within a program, allowing for multiple tasks to be performed simultaneously (or appear to be simultaneous) within the same application. Threads run within a process, which is the larger context that contains the program's code, data, and resources. Threads share the same memory space of the process but can execute independently, allowing for multitasking within a single application. A context switch is the process by which an operating system (OS) saves the state of a currently running thread or process and then restores the state of a different thread or process, allowing the CPU to switch from executing one to the other. This is an essential part of multitasking, where a system runs multiple processes or threads seemingly at the same time, sharing CPU resources.

Memory allocation and management

Static and dynamic memory allocation are two methods of allocating memory to programs during their execution. The key difference lies in when and how the memory is allocated and managed.

Static and dynamic memory allocation differ significantly across C, Java, and Python. In C, static memory allocation occurs when memory is allocated at compile time for fixed-size arrays or structures. This memory remains reserved for the program's entire execution and is automatically released when the program ends. In contrast, dynamic memory allocation in C allows for memory to be allocated at runtime using functions like malloc() and free(), giving developers flexibility but requiring careful management to avoid memory leaks.

In Java, static memory allocation is used for fixed-size arrays and instance variables, with memory allocated when classes are loaded or methods are invoked. Dynamic memory allocation in Java is managed automatically through the new keyword, and the garbage collector reclaims memory for objects no longer in use, reducing the risk of memory leaks.

Python, while primarily using dynamic memory allocation, does not explicitly support traditional static allocation. However, it allows the creation of immutable types like tuples, which behave similarly to statically allocated structures. Python’s dynamic memory allocation occurs automatically as variables and objects are created, with a built-in garbage collector managing memory, making it easy for developers to focus on coding without worrying about manual memory management.

Overall, while C requires manual memory management for dynamic allocation, Java and Python provide more automated approaches, making them more convenient but also abstracting away some of the control that C offers, and adding complexity in performane.

C: Low-Level Control and Manual Threading

As a low-level language, C gives developers direct access to memory via pointers and manual memory management. This fine-grained control extends to threading, typically handled through libraries such as POSIX Threads (pthreads). While pthreads enables efficient concurrent programming, it requires meticulous attention to synchronization. Improper handling of shared resources can easily lead to concurrency issues such as race conditions and deadlocks. Furthermore, managing threads in C involves low-level constructs like mutexes and condition variables, which, although powerful, demand careful design to avoid complex bugs.

When it comes to processes, C employs system calls like fork() and exec() to create and manage them. These calls offer significant control and flexibility, but they also introduce challenges in terms of memory management. Developers must ensure efficient use of system resources, as process creation can lead to high overhead if not carefully optimized. Although process creation in C is lightweight compared to higher-level languages, managing memory for each process manually can increase the complexity of the program.

Java: High-Level Abstraction with Built-in Threading

Java, on the other hand, provides a higher-level abstraction, especially in its approach to concurrency. Java's built-in threading model simplifies concurrent programming with the Thread class and the Executor framework. These abstractions make it easier to create and manage threads while reducing the likelihood of concurrency issues, as Java offers built-in synchronization mechanisms such as synchronized methods and blocks. Additionally, higher-level concurrency constructs, like Locks and Executors, enable safe and scalable multithreading without requiring the manual effort that C demands.

Java runs on the Java Virtual Machine (JVM), which this makes Java platform-independent, it also introduces some performance overhead. Memory management is handled by automatic garbage collection, which simplifies development but can lead to unpredictable pauses during execution, especially in memory-intensive applications. Unlike C, where developers explicitly manage memory allocation and deallocation, Java's garbage collector reclaims memory automatically. This abstraction simplifies development but can occasionally result in performance bottlenecks.

Python: Flexibility and Simplicity with Interpreted Execution

Python, an interpreted language, offers the highest level of abstraction among the three. Its simplicity and ease of use make it a popular choice for rapid development, but this comes at the cost of lower performance, especially in areas like threading and memory management. Python handles threading through the threading module, but it is constrained by the Global Interpreter Lock (GIL), which prevents multiple native threads from executing Python bytecode simultaneously in a single process. This severely limits Python's ability to perform true parallelism for CPU-bound tasks, although it remains effective for I/O-bound tasks.

For process management, Python provides higher-level abstractions through the multiprocessing module, which allows the creation of separate processes rather than threads to bypass the GIL. This model is easier to use than C’s low-level process management, but Python’s interpreted nature and reliance on the operating system for process handling lead to higher overhead compared to C and Java.

In terms of memory management, Python relies entirely on garbage collection, much like Java, but its reference counting mechanism and cyclic garbage collector can introduce additional performance overhead. Python abstracts away memory allocation and deallocation from the developer, which simplifies programming but leads to slower execution for memory-intensive tasks compared to C’s manual memory management.

3. Analysis

3.1 Project proposal

Program Development
Create sample programs in C, Java, and Python that implement basic algorithms, handle memory allocation, and manage threads or processes. For threading, use pthreads in C, the Thread class in Java, and threading/multiprocessing modules in Python (Py-Spy). This ensures that the comparison between languages is consistent and fair.

Performance Testing
Use profiling tools to measure each program's execution time, memory usage, and threading efficiency. In C, tools like gprof and Valgrind will track performance. Java programs will be profiled using VisualVM or teh Intellij IDEA built-in Profiler or similar tools, and Python's performance will be measured using cProfile and memory_profiler. Multiple runs will ensure accurate data collection.

Data Analysis
Analyze the collected data to compare how efficiently each language handles algorithm execution, memory management, and thread handling. Look for patterns or differences that highlight where each language excels or falls behind.

Algorithm ideas:

Parallel Merge Sort
Description: Implement a parallelized Merge Sort where threads sort different sections of an array. Each thread handles a portion of the array, and the results are merged, requiring synchronization and context switching.
Resources:

- Memory Allocation: Dynamically allocate subarrays.
- Memory Access: Threads access and sort parts of the array.
- Thread Creation: Each thread sorts a portion of the array.
- Context Switching: Synchronization during the merge phase.

Matrix Multiplication with Multithreading
Description: Multiply two large matrices using multiple threads. Each thread calculates a portion of the result matrix by accessing memory and performing computations concurrently. Memory is allocated statically or dynamically depending on matrix size.
Resources:

- Memory Allocation: Allocate matrices statically/dynamically.
- Memory Access: Access matrix elements during computation.
- Thread Creation: Threads compute different parts of the matrix.
- Context Switching: Synchronize threads to merge results

4. Design

4.1. Parallel Merge Sort Algorithm

The parallel merge sort algorithm is a variation of the standard merge sort that uses parallelism to improve performance, especially for large datasets. In the standard merge sort, the algorithm follows a divide-and-conquer approach, where it recursively splits the array into halves until each subarray has one element, then merges the sorted halves back together. In a parallel version, the splitting and sorting of subarrays can happen concurrently, taking advantage of multiple processors or cores.

How Parallel Merge Sort Works

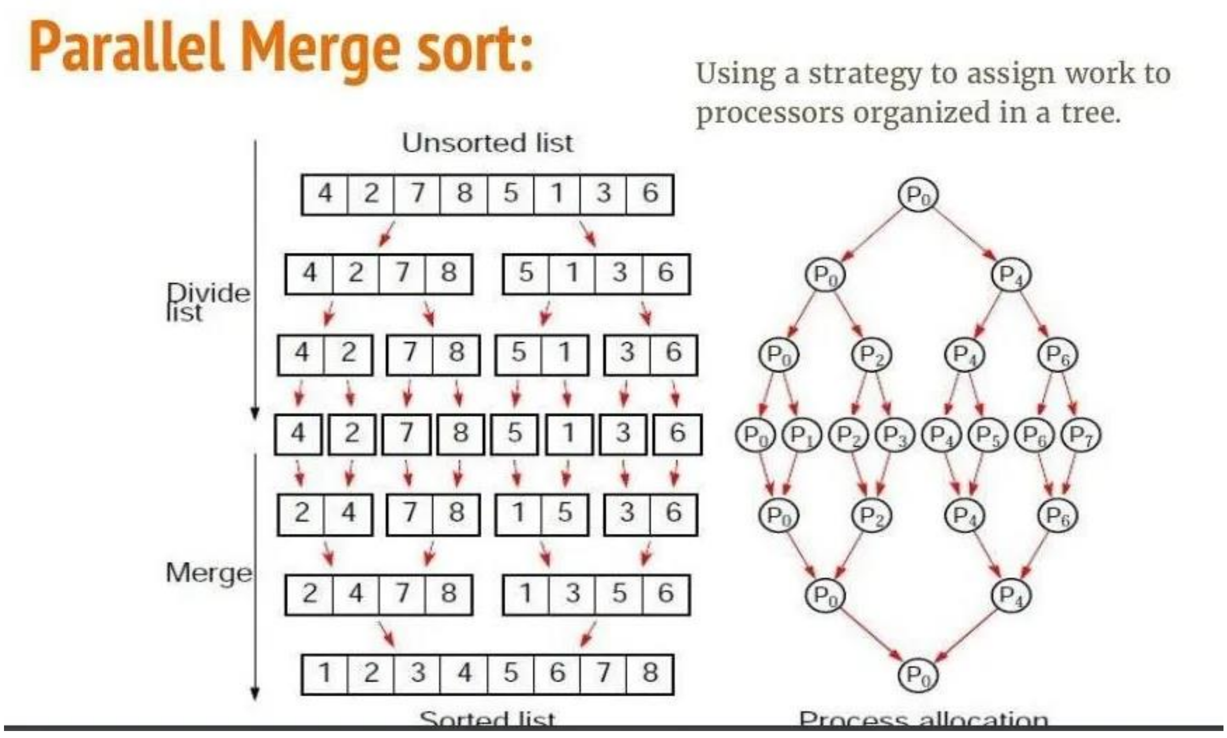
1. **Splitting the Array**:
   - The array is recursively split into two halves until the subarrays reach a small enough size, where they can be sorted sequentially (this threshold is chosen based on empirical performance testing).
2. **Sorting Each Half in Parallel**:
   - For each recursive split, the left and right halves are sorted in parallel. This can be done by creating separate threads or processes for each half.
3. **Merging the Sorted Halves**:
   - Once the subarrays are sorted, the merge step combines two sorted halves into a single sorted array.

Complexity Analysis

- **Time Complexity**:
  - **Average Case**: O(nlogn), as in standard merge sort, but with potentially improved constant factors due to parallel execution.
  - **Parallel Speedup**: In an ideal scenario, parallel merge sort can achieve a speedup close to O((nlogn)/p), where p is the number of processors.
- **Space Complexity**:
  - O(n) for storing intermediate results during merging, similar to standard merge sort.

Explanation of the Pseudocode

1. **Function `parallel_merge_sort(arr, left, right, threshold)`**:
   - Takes an array `arr`, the indices `left` and `right`, and a `threshold`.
   - If the size of the subarray is less than or equal to the threshold, it calls a sequential sorting function to sort that subarray.
   - Otherwise, it splits the array into two halves, creating two threads to sort each half in parallel.
   - After the threads complete, it merges the two sorted halves.



Parallel Merge sort:

Using a strategy to assign work to processors organized in a tree.

Algorithm 1 Merge Sort with Multithreading

```
1: procedure MergeSort($params$)
2:     $args \leftarrow castparamstoSortParamspointer$
3:     $left \leftarrow args.left$
4:     $right \leftarrow args.right$
5:     $array \leftarrow args.array$
6:     if $left < right$ then
7:         $mid \leftarrow left + (right - left)/2$
8:         if $(right - left) \leq THRESHOLD$ then    ▷ Sequential sort if below threshold
9:             $left\_params \leftarrow \{array, left, mid\}$
10:            $right\_params \leftarrow \{array, mid + 1, right\}$
11:            MergeSort($left\_params$)
12:            MergeSort($right\_params$)
13:        else                                          ▷ Parallel sort if above threshold
14:            $left\_params \leftarrow \{array, left, mid\}$
15:            $right\_params \leftarrow \{array, mid + 1, right\}$
16:            $left\_thread \leftarrow$ CreateThread($NULL, 0, MergeSort, left\_params$)
17:            $right\_thread \leftarrow$ CreateThread($NULL, 0, MergeSort, right\_params$)
18:        end if
19:        Merge($array, left, mid, right$)
20:    end if
21:    return 0
22: end procedure
```

2. **Function** `merge(arr1, arr2, result)`

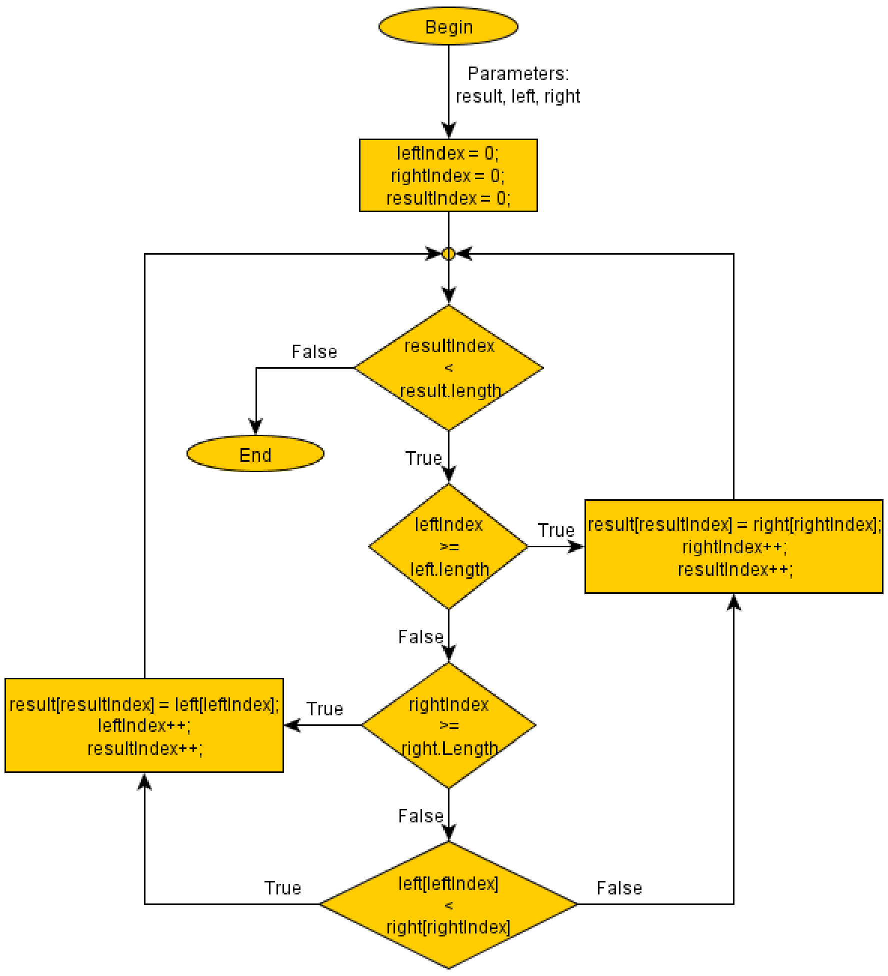
Merge Process:

- Pointers i and j are initialized to keep track of the current index in the arr1, arr2, and the main array result, respectively.
- A loop is used to compare elements from both arrays that must be merged:
    - The smaller element is placed into the original array at position i+j-1, and the corresponding pointer (i or j) is incremented.
    - This continues until one of the arrays is fully traversed.

Copy Remaining Elements:

- After the main merging loop, any remaining elements from arr1 are copied back into arr, followed by any remaining elements from arr2. This ensures that all elements are included in the final sorted array.

Flowchart for merging


Begin
Parameters: result, left, right
leftIndex = 0; rightIndex = 0; resultIndex = 0;
resultIndex < result.length
False
End
True
leftIndex >= left.length
True
result[resultIndex] = right[rightIndex]; rightIndex++; resultIndex++;
False
rightIndex >= right.Length
True
result[resultIndex] = left[leftIndex]; leftIndex++; resultIndex++;
False
left[leftIndex] < right[rightIndex]
True
False

Algorithm 1 Merge Function

```
procedure MERGE($array$, $left$, $mid$, $right$)
    $left\_size \leftarrow mid - left + 1$
    $right\_size \leftarrow right - mid$
    $left\_array \leftarrow$ allocate array of size $left\_size$
    $right\_array \leftarrow$ allocate array of size $right\_size$
    for $i \leftarrow 0$ to $left\_size - 1$ do
        $left\_array[i] \leftarrow array[left + i]$
    end for
    for $i \leftarrow 0$ to $right\_size - 1$ do
        $right\_array[i] \leftarrow array[mid + 1 + i]$
    end for
    $i \leftarrow 0, j \leftarrow 0, k \leftarrow left$
    while $i < left\_size$ and $j < right\_size$ do
        if $left\_array[i] \leq right\_array[j]$ then
            $array[k] \leftarrow left\_array[i]$
            $i \leftarrow i + 1$
        else
            $array[k] \leftarrow right\_array[j]$
            $j \leftarrow j + 1$
        end if
        $k \leftarrow k + 1$
    end while
    while $i < left\_size$ do
        $array[k] \leftarrow left\_array[i]$
        $i \leftarrow i + 1, k \leftarrow k + 1$
    end while
    while $j < right\_size$ do
        $array[k] \leftarrow right\_array[j]$
        $j \leftarrow j + 1, k \leftarrow k + 1$
    end while
    deallocate $left\_array$
    deallocate $right\_array$
end procedure
```

4.2. Parallel Matrix Multiplication Algorithm

Parallel matrix multiplication using multithreading is an efficient way to multiply large matrices by distributing the workload across multiple threads.

Matrix multiplication involves computing the dot product of rows of the first matrix with columns of the second matrix. Given two matrices A (of size m*k) and B (of size k*n), the resulting matrix C (of size m×n) is calculated as follows: each element C[i][j] in the resulting matrix is calculated by taking the dot product of the i-th row of matrix A and the j-th column of matrix B.

In a multithreaded approach, one can assign each thread to calculate the values of one or more elements in the result matrix independently.

Complexity Analysis

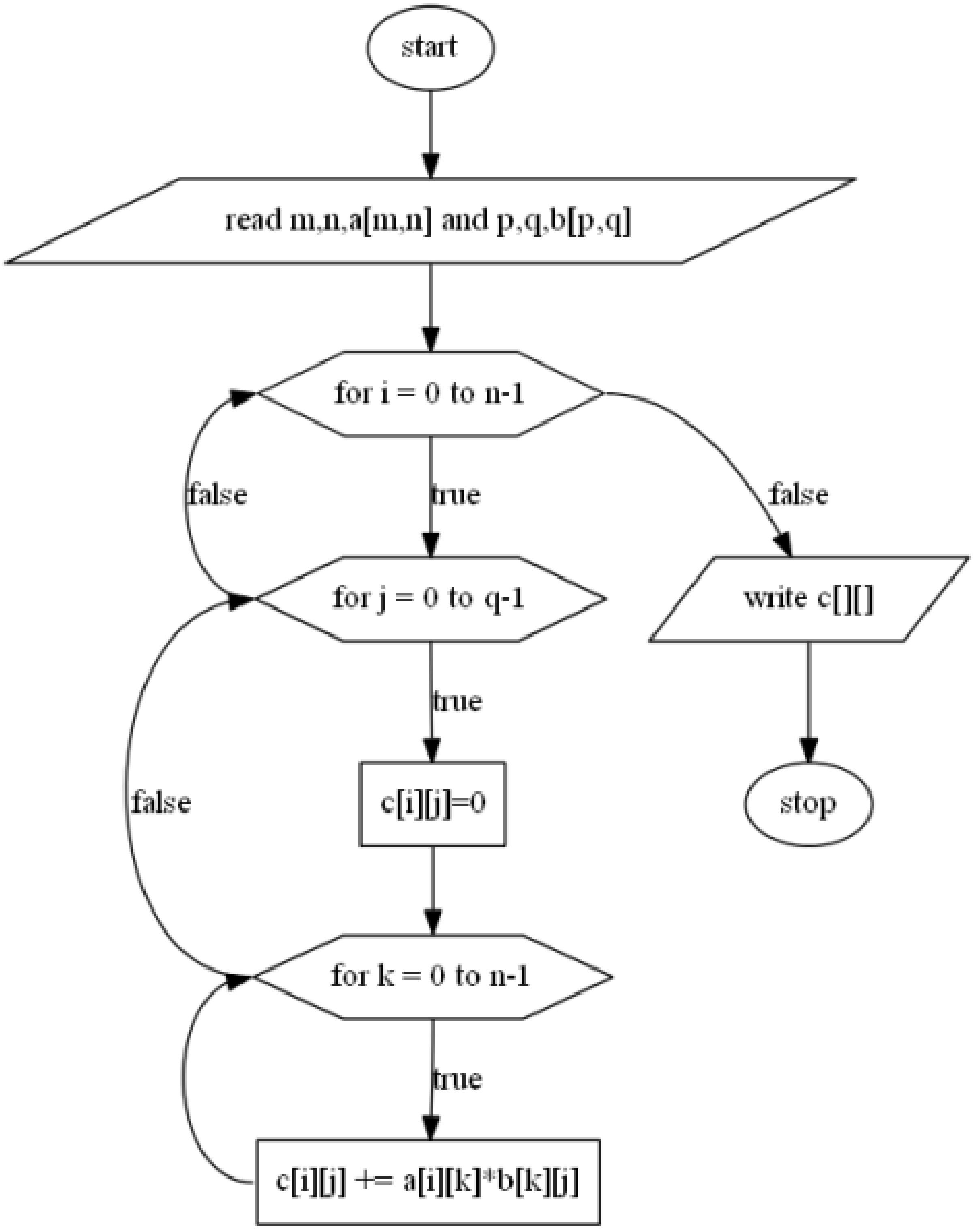
1. **Time Complexity**:
   - The naive approach for multiplying two matrices is O(m*n*k), where m is the number of rows in A, n is the number of columns in B, and k is the number of columns in A (which is equal to the number of rows in B).
   - When using multithreading, while the theoretical complexity remains the same, the actual performance time can be reduced significantly due to parallel execution, particularly on multi-core processors.
2. **Space Complexity**:
   - The space complexity is O(m*n) for the resulting matrix C.
   - Temporary storage for intermediate results may also be required, but this is generally minimal compared to the result matrix.

Pseudocode

Algorithm 1 Matrix Multiplication Using Threads

```
procedure MAIN
    $A \leftarrow CreateMatrix(m, p)$                                ▷ Matrix $A$ of size $m \times p$
    $B \leftarrow CreateMatrix(p, n)$                                ▷ Matrix $B$ of size $p \times n$
    $C \leftarrow CreateMatrix(m, n)$                                ▷ Matrix $C$ of size $m \times n$
    for $i \leftarrow 0$ to $m - 1$ do
        $threadData[i] \leftarrow \{i, n, p, A, B, C\}$
        $threads[i] \leftarrow CreateThread(NULL, 0, MultiplyRow, threadData[i])$
    end for
end procedure
procedure MULTIPLYROW($param$)
    $data \leftarrow CastparamtoThreadData$
    $row \leftarrow data.row$
    for $j \leftarrow 0$ to $data.n - 1$ do
        $data.C[row][j] \leftarrow 0$
        for $k \leftarrow 0$ to $data.m - 1$ do
            $data.C[row][j] \leftarrow data.C[row][j] + data.A[row][k] \times data.B[k][j]$
        end for
    end for
end procedure
```

Flowchart


start
read m,n,a[m,n] and p,q,b[p,q]
for i = 0 to n-1
true
false
false
for j = 0 to q-1
write c[][]
true
c[i][j]=0
false
stop
for k = 0 to n-1
true
c[i][j] += a[i][k]*b[k][j]

4. Implementation

4.1. Implementation in C

Threads are created using the **Windows API.**

The function **CreateThread** creates a new thread and executes the specified function.

Example: HANDLE hThreadArray[MAX_THREADS];

The function transmitted must be declared like this:
DWORD WINAPI MyThreadFunction(LPVOID lpParam);

```
PMYDATA pDataArray[MAX_THREADS]; // for the arguments transmitted
DWORD   dwThreadIdArray[MAX_THREADS]; // returned thread identifier
```

Example:

```
   hThreadArray[i] = CreateThread(
        NULL,                  // default security attributes
        0,                     // use default stack size
        MyThreadFunction,      // thread function name
        pDataArray[i],         // argument to thread function
        0,                     // use default creation flags
        &dwThreadIdArray[i]);  // returns the thread identifier
```

The function **WaitForMultipleObjects** waits for the threads to complete execution.
WaitForMultipleObjects(MAX_THREADS, hThreadArray, TRUE, INFINITE);

Where INFINITE is the timeout interval in miliseconds (wait infinitely in this case)

The function **CloseHandle** closes a handle (thread) and frees the resources.

CloseHandle(hThreadArray[i]);

4.1.1 Parallel Merge Sort in C

```c
typedef struct {
    int* array; // the array
    int left;   // left index
    int right;  // right index
} SortParams;

DWORD WINAPI merge_sort(LPVOID params) {
    // Cast the parameters type
    SortParams* args = (SortParams*)params;

    // Destructure the params into array, left, and right
    int* array = args->array;
    int left = args->left;
    int right = args->right;

    if (left < right) {
        int mid = left + (right - left) / 2;

        if ((right - left) <= THRESHOLD) {
            // Sequential sort if below threshold
            SortParams left_params = { array, left, mid };
            SortParams right_params = { array, mid + 1, right };

            // Call merge sort for the left and right halves of the array
            merge_sort(&left_params);
            merge_sort(&right_params);
        }
        else {
            // Parallel sort if above threshold
            HANDLE left_thread, right_thread;
            DWORD left_thread_id, right_thread_id;

            SortParams left_params = { array, left, mid };
            SortParams right_params = { array, mid + 1, right };

            // Start the threads
            left_thread = CreateThread(NULL, 0, merge_sort, &left_params, 0, &left_thread_id);
            right_thread = CreateThread(NULL, 0, merge_sort, &right_params, 0, &right_thread_id);

            // Wait for left and right threads to finish their execution
            WaitForSingleObject(left_thread, INFINITE);
            WaitForSingleObject(right_thread, INFINITE);

            // Close the handles and free the resources
            CloseHandle(left_thread);
            CloseHandle(right_thread);
        }

        // After returning from the recursive calls, merge the left and right arrays
        merge(array, left, mid, right);
    }
    return 0;
}
```

Listing 1: Multithreaded Merge Sort

4.1.2. Parallel matrix multiplication in C

```c
typedef struct {
    int row; // Row number in the result matrix
    int n;   // Number of columns in matrix B
    int m;   // Number of columns in matrix A and rows in matrix B
    int** A; // Matrix A
    int** B; // Matrix B
    int** C; // Result matrix C
} ThreadData;

DWORD WINAPI multiply_row(LPVOID param) {
    // Cast the parameters to the required type ThreadData
    ThreadData* data = (ThreadData*)param;

    // Destructure the parameter structure
    int row = data->row;

    // Iterate through the result matrix C on the given row and compute its elements
    for (int j = 0; j < data->n; j++) {
        for (int k = 0; k < data->m; k++) {
            data->C[row][j] += data->A[row][k] * data->B[k][j];
        }
    }
    return 0;
}

int main(void) {
    // ...
    HANDLE threads[MAX_THREADS];
    // Create the threads and their parameter structures
    ThreadData threadData[MAX_THREADS];

    // Create threads for each row
    for (int i = 0; i < m; i++) {
        // Provide the parameters for threadData[i]
        threads[i] = CreateThread(NULL, 0, multiply_row, &threadData[i], 0, NULL);
    }

    // Wait for the threads to finish their execution, and then close the resources
    WaitForMultipleObjects(m, threads, TRUE, INFINITE);
    for (int i = 0; i < m; i++) {
        CloseHandle(threads[i]);
    }
    // ...
}
```

Listing 1: Multithreaded Matrix Row Multiplication

4.2. Implementation in Python

Use **Python threading module** to create threads.

Create a new thread example:

import Threading
thread = **threading.Thread**(myThreadFunction, threadArgs)
thread.start()

thread.join() is a method used to ensure that the main program waits for a thread to complete before continuing execution. It is a way to synchronize the main thread (or any thread calling .join()) with the completion of another thread.

thread.append() in Python refers to appending thread objects to a list. This is a common technique to manage multiple threads dynamically and it is used in order to be able to join the threads later.

4.2.1. Parallel merge sort in Python

```python
import threading

def merge_sort_thread(params):
    array, left, right = params
    if left < right:
        mid = left + (right - left) // 2

        if (right - left) <= THRESHOLD:
            # Sequential sort if below threshold
            merge_sort_thread((array, left, mid))
            merge_sort_thread((array, mid + 1, right))
        else:
            # Parallel sort if above threshold
            left_thread = threading.Thread(
                target=merge_sort_thread, args=((array, left,
    mid),))
            right_thread = threading.Thread(
                target=merge_sort_thread, args=((array, mid +
    1, right),))

            left_thread.start()
            right_thread.start()

            left_thread.join()
            right_thread.join()

        merge(array, left, mid, right)
```

Listing 1: Multithreaded Merge Sort

4.2.2. Parallel matrix multiplication in Python

```python
import threading

def multiply_row(data):
    row, n, m, A, B, C = data
    for j in range(n):
        C[row][j] = sum(A[row][k] * B[k][j] for k in range(m))

def main():
    # ...
    # Create threads for each row of matrix C
    threads = []
    for i in range(m):
        thread_data = (i, n, p, A, B, C)
        thread = threading.Thread(target=multiply_row, args=(thread_data,))
        threads.append(thread)
        thread.start()

    # Wait for all threads to finish
    for thread in threads:
        thread.join()
    # ...
```

4.3. Implementation in Java

Create a thread in Java by extending the Thread class.

Example:

```java
class ExampleThreadTask extends Thread {

  // constructor
  ExampleThread(String name) {
    // call the constructor in the parent class
    // and pass the name
    super(name);
  }

  // override the run() method in
  // the parent Thread class
  @Override
  public void run() {
    // run whatever code you like
  }
}
```

Create a thread and start its execution by calling the method **start()** on that thread.

myThread = new ExampleThreadTask(myThreadArguments);
myThread.start();

When creating multiple threads, we must use the **join()** method to synchronize the threads termination between them. This method ensures that one thread waits for another to complete its task before continuing.

```
try {
    for (int i = 0; i < m; i++) {
        threads[i].join();  // Wait for each thread to finish
    }
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

4.3.1. Parallel merge sort in Java

```java
public class MergeSortTask extends Thread {
    private int[] array;
    private int left;
    private int right;
    private static final int THRESHOLD = 1000; // Threshold for switching to sequential merge sort

    public MergeSortTask(int[] array, int left, int right) {
        this.array = array;
        this.left = left;
        this.right = right;
    }

    @Override
    public void run() {
        if (left < right) {
            int mid = left + (right - left) / 2;

            if (right - left <= THRESHOLD) {
                mergeSort(array, left, mid);
                mergeSort(array, mid + 1, right);
            } else {
                MergeSortTask leftTask = new MergeSortTask(array, left, mid);
                MergeSortTask rightTask = new MergeSortTask(array, mid + 1, right);

                leftTask.start();
                rightTask.start();

                try {
                    leftTask.join();
                    rightTask.join();
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }

            merge(array, left, mid, right);
        }
    }

    // Classical merge sort for array with length below THRESHOLD
    private void mergeSort(int[] array, int left, int right) {
        if (left < right) {
            int mid = left + (right - left) / 2;

            mergeSort(array, left, mid);
            mergeSort(array, mid + 1, right);

            merge(array, left, mid, right);
        }
    }

    public static void main(String[] args) {
        // Example array to be sorted
        int[] array = {5, 2, 9, 1, 5, 6};
        int n = array.length;

        MergeSortTask task = new MergeSortTask(array, 0, n - 1);
        task.start();

        try {
            task.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        // Output sorted array
        for (int num : array) {
            System.out.print(num + " ");
        }
    }
}
```

Listing 1: Multithreaded Merge Sort

4.3.2. Parallel matrix multiplication in Java

```java
private static class MultiplyRowTask extends Thread {
    private int row; // Row number in the result matrix
    private int n; // Number of columns in matrix B
    private int m; // Number of columns in matrix A and rows in matrix B
    private int[][] A; // Matrix A
    private int[][] B; // Matrix B
    private int[][] C; // Result matrix C

    public MultiplyRowTask(int row, int n, int m, int[][] A, int[][] B, int[][] C) {
        this.row = row;
        this.n = n;
        this.m = m;
        this.A = A;
        this.B = B;
        this.C = C;
    }

    @Override
    public void run() {
        for (int j = 0; j < n; j++) {
            C[row][j] = 0;
            for (int k = 0; k < m; k++) {
                C[row][j] += A[row][k] * B[k][j];
            }
        }
    }
}

public static void main(String[] args) {
    // ...
    Thread[] threads = new Thread[m];

    for (int i = 0; i < m; i++) {
        threads[i] = new MultiplyRowTask(i, n, p, A, B, C);
        threads[i].start();
    }

    try {
        for (int i = 0; i < m; i++) {
            threads[i].join();  // Wait for each thread to finish
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    // ...
}
```

Listing 1: Multithreaded Matrix Row Multiplication

5. Measurements

For measuring the performance metrics for each algorithm and log the results we used a python program for generating random inputs for both algorithms and a program for running each implementation. This way, we ensured that the algorithms run with the same input and the comparison is fair. For parallel merge sort, the number of elements in the array is generated randomly but in increasing order.

Measuring the execution time in C

Measuring the execution time of a C program using clock_t involves using the **clock()** function provided by the <time.h> library. The process measures the amount of CPU time consumed by the program between two points in its execution.

- **clock()**: A function that returns the number of **clock ticks** elapsed since the program started execution. A clock tick is a unit of time defined by the implementation, often based on the CPU clock.
- **clock_t**: A data type that represents the value returned by clock(). It is typically an integer or a floating-point type.
- **CLOCKS_PER_SEC**: A constant defined in <time.h> that indicates the number of clock ticks per second.

<u>**Steps in Code**</u>

1. **Include the <time.h> library**:
   - This provides access to clock_t, clock(), and CLOCKS_PER_SEC.
2. **Store the starting clock value**:
   - Use clock() to capture the initial time before the code block to be measured.
3. **Store the ending clock value**:
   - Use clock() again after the code block completes.
4. **Calculate the elapsed time**:
   - Subtract the start time from the end time to get the elapsed clock ticks.
   - Divide the result by CLOCKS_PER_SEC to convert ticks to seconds.

Static vs dynamic memory access in C

In order to be able to compare the performance metrics when using static vs dynamic memory, it is required to run the algorithm once on a static data structure and then on a dynamic data structure and measure the execution time for both cases. This way, it is ensured that the main differences appear due to memory management (allocation and access).

Thread creation time in C

To measure the time it takes to create a thread in C, you can use high-resolution performance counters such as **QueryPerformanceCounter()** on Windows. This function provides precise timing, making it suitable for measuring small time intervals like thread creation time.

QueryPerformanceCounter(LARGE_INTEGER* counter):

- Retrieves the current value of the high-resolution performance counter.
- The counter is a hardware feature that provides high-precision time measurements.

QueryPerformanceFrequency(LARGE_INTEGER* frequency):

- Retrieves the frequency of the high-resolution performance counter in ticks per second.
- Used to convert counter values into time in seconds or milliseconds.

Steps in code

1. **Capture the Starting Time**:
   - o Use QueryPerformanceCounter() before starting the thread creation process.
2. **Create the Thread**:
   - o Use an appropriate thread creation function (like CreateThread on Windows).
3. **Capture the Ending Time**:
   - o Use QueryPerformanceCounter() after the thread is created.
4. **Calculate Elapsed Time**:
   - o Subtract the start time from the end time to get the elapsed ticks.
   - o Convert the ticks into seconds or milliseconds using the frequency retrieved by QueryPerformanceFrequency().

Thread context switch in C

A context switch in computing occurs when the CPU switches from executing one thread to another. This involves saving the state of the currently executing thread and restoring the state of the next thread to run. In C, while the concept of a context switch is managed at the operating system (OS) level, it is closely tied to multithreading, multitasking, and interrupt handling.

Example for measuring context switch in code:

```
QueryPerformanceCounter(&start);
WaitForMultipleObjects(m, threads, TRUE, INFINITE);
QueryPerformanceCounter(&end);
```

The WaitForMultipleObjects function waits for all the threads in the threads array to complete or reach a specific state. During this wait, context switching can occur because the operating system scheduler might switch between the threads if any of them are in a waiting or running state.

After the WaitForMultipleObjects() function has completed (meaning the threads have signaled or completed), the current value of the performance counter is retrieved again. Then we compute the elapsed time by end – start and convert it in seconds using the frequency:

```
double wait_time = (double)(end.QuadPart - start.QuadPart) / frequency.QuadPart;
```

Measuring the execution time in Python

In Python, you can measure the execution time of a block of code or function using time.time() to record timestamps before and after the code execution.

time.time():

- This function from the time module returns the current time as a floating-point number in seconds since the epoch (January 1, 1970, UTC).
- Used to measure elapsed time by subtracting two timestamps.

Steps to Measure Execution Time:

- Record the start time just before the code execution.
- Execute the code or function you want to measure.
- Record the end time immediately after the code execution.
- Calculate the elapsed time as the difference between the end time and the start time.

Thread creation time in Python

Thread Creation Start Time (thread_start):

- time.time() is called just before a thread is created and started.
- Captures the timestamp immediately before the thread creation starts.

Thread Creation (thread = threading.Thread(...) and thread.start()):

- A new thread is instantiated using threading.Thread().
- thread.start() begins executing the thread's target function (multiply_row).

Thread Creation End Time (thread_end):

- time.time() is called immediately after the thread is created and started.
- Captures the timestamp after the thread has been successfully created and started.

Time Taken for Thread Creation:

- Calculated as thread_end - thread_start.
- Represents the duration from the initiation of thread creation to when it starts execution.

Storing the Time:

- Each thread's creation time is appended to the list threads_creation_times.

Context switch time in Python

```
join_start_time = time.time()
for thread in threads:
   thread.join()
join_end_time = time.time()
```

In this snippet, **context switching contributes to the thread joining time** because:

- When the main thread calls thread.join(), it might have to wait for the worker threads to finish.
- If multiple threads are running on a limited number of CPU cores, frequent context switches occur, especially if the threads are compute-intensive.

Thus, the time captured by join_duration indirectly reflects:

- The **execution time of threads**.
- The **overhead of context switching** while the threads are running.

Measuring the execution time in Java

In Java, you can measure the execution time of a block of code or a method by recording the time before and after its execution using System.nanoTime(). This method is preferred over System.currentTimeMillis() for precise time measurements because it provides a higher resolution.

Steps to Measure Execution Time

1. Record the start time using System.nanoTime().
2. Execute the code or function whose execution time you want to measure.
3. Record the end time using System.nanoTime().
4. Calculate the elapsed time as the difference between the end time and the start time.
5. Divide the elapsed time by 1_000_000_000 to convert it to seconds.

Thread creation time in Java

Loop to Create and Start Threads:

- **threadStartTime**: Captures the time just before the thread creation starts.
- **Thread Creation**: A new thread is instantiated (using new MultiplyRowTask(...)) and immediately started (threads[i].start()).
- **threadEndTime**: Captures the time just after the thread is created and started.
- **Thread Creation Time Calculation**: threadEndTime - threadStartTime calculates the elapsed time for creating and starting a thread.
- The value is converted from nanoseconds to seconds by dividing by 1_000_000_000.0 and stored in the threadCreationTimes array.

What Does Thread Creation Time Measure?

For each thread, **thread creation time** measures the duration between:

- **Initiating the thread creation** (new MultiplyRowTask(...)).
- **Starting the thread** (threads[i].start()).

This includes:

- Allocating resources for the thread (e.g., memory and CPU context).
- Setting up the thread's internal data structures.
- Scheduling the thread to run on the CPU.

Context switch in Java

```
long joinStartTime = System.nanoTime();
try {
    for (int i = 0; i < m; i++) {
        threads[i].join();  // Wait for each thread to finish
    }
 } catch (InterruptedException e) {
    e.printStackTrace();
}
 long joinEndTime = System.nanoTime();
```

joinStartTime:

- Captures the starting time (in nanoseconds) just before the main thread begins waiting for the worker threads to complete using the join() method.

threads[i].join():

- The main thread calls join() for each thread in the array threads.
- This method blocks the main thread until the thread threads[i] has completed its execution.
- While waiting, the main thread may be context-switched out by the operating system scheduler to allow other threads or processes to execute.

joinEndTime:

- Captures the ending time (in nanoseconds) after all threads have completed their execution and rejoined the main thread.

Elapsed Join Time:

- Calculated as: long joinDuration = joinEndTime – joinStartTime, represents the total time the main thread spent waiting for all worker threads to finish.

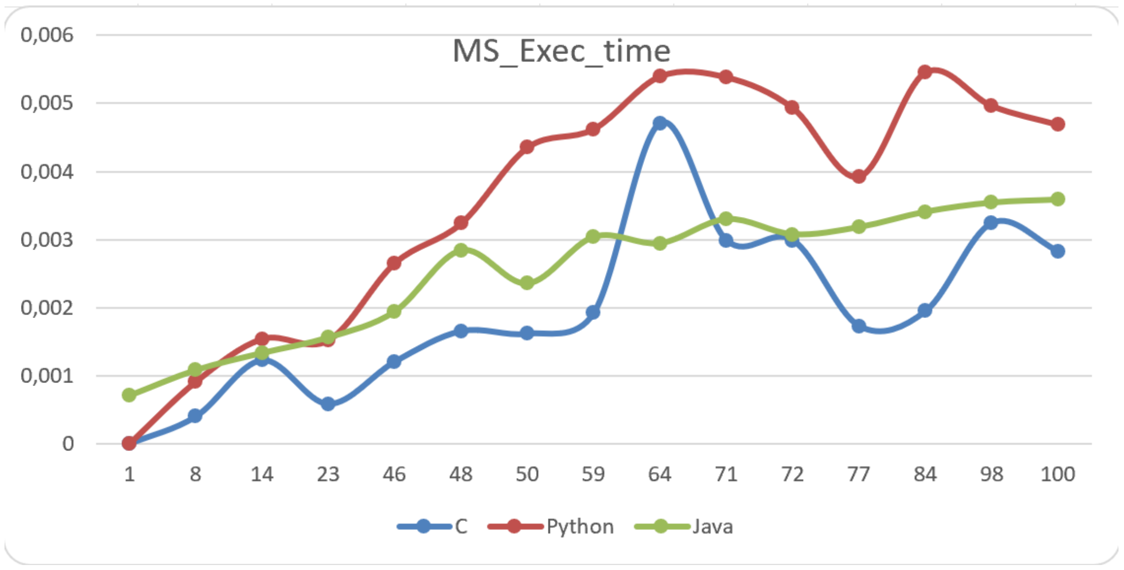
During the join() process:

1. The main thread is **blocked** while waiting for the worker threads.
2. The operating system's scheduler may repeatedly **switch contexts**:
   - From worker threads to other tasks or threads.
   - From the main thread to worker threads once they finish.
3. These context switches indirectly affect the total join time, as they impact the worker threads' execution time.

6. Comparisons and Graphs

6.1. Merge Sort

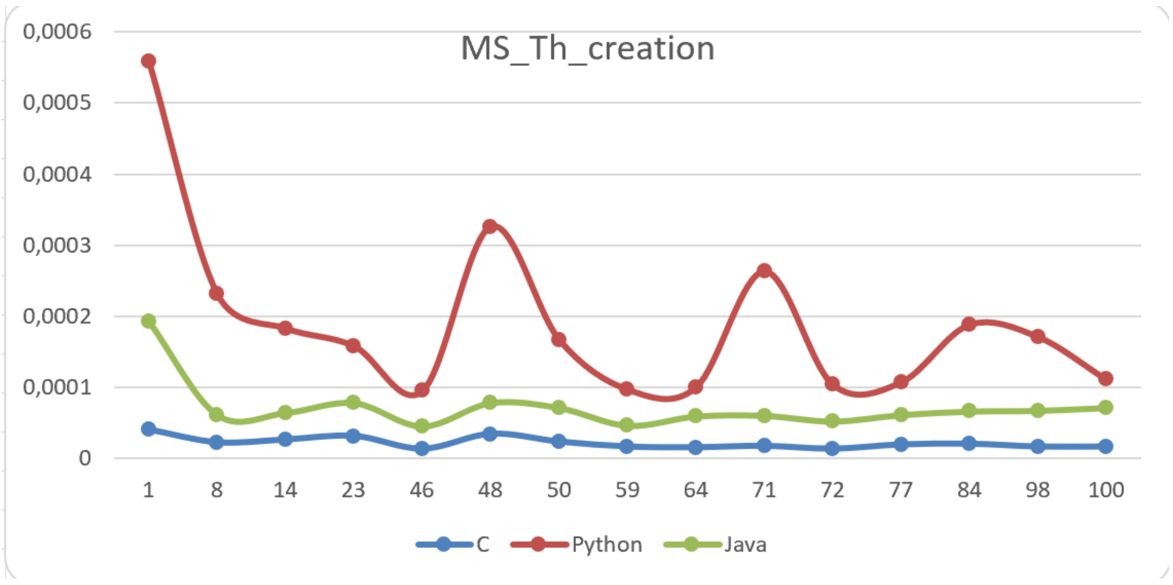
6.1.1. Execution time



C is often regarded as the best choice for performance due to its low-level control over memory and hardware resources. As a compiled language, C directly translates code into machine instructions, avoiding the overhead associated with interpreters or virtual machines. This results in faster execution, especially when combined with manual memory management, which eliminates the need for garbage collection. Additionally, C allows for various optimization techniques, such as efficient memory access patterns and fine-tuned control over system resources, making it highly efficient for performance-critical tasks. Its simplicity and minimal abstraction layers further contribute to its speed, particularly in systems programming and embedded environments.

6.1.2. Thread creation time

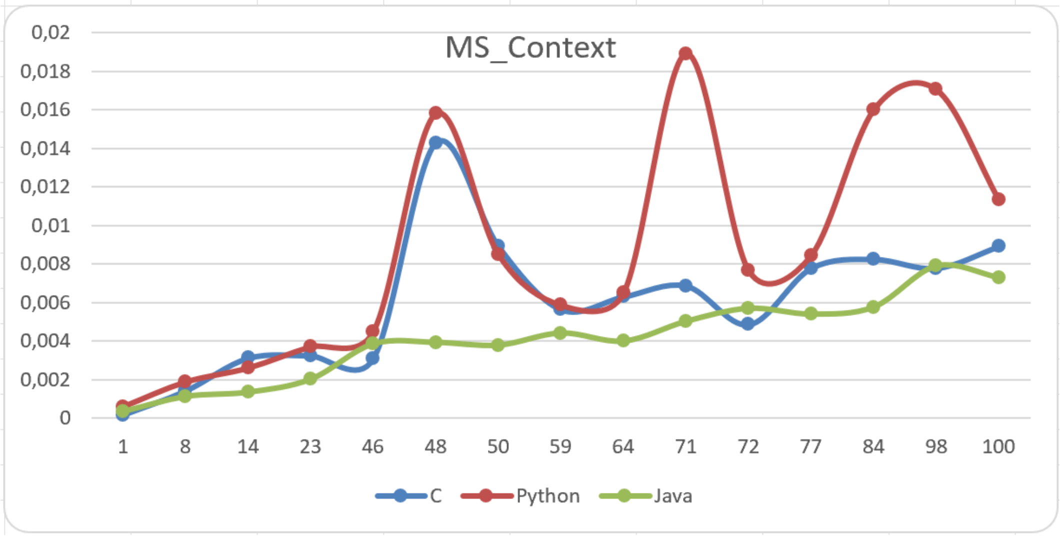
- This was measured by starting n threads, summing up all the creation times and then computing the average value. (n is the number of elements in the array to be sorted).



C is the fastest in terms of thread creation due to its direct access to low-level system resources and minimal runtime overhead. It uses highly optimized system calls like CreateThread, which are executed directly by the operating system without additional layers of abstraction. Java, while faster than Python, is slower than C because it runs on the Java Virtual Machine (JVM), which introduces overhead through JVM-level abstractions, memory management, and garbage collection. Java threads are managed through the JVM's memory heap, which can slow down thread creation times. Python is the slowest due to its interpreted nature and the Global Interpreter Lock (GIL), which prevents true parallel execution of threads and adds delays. Additionally, Python's threading model involves higher-level abstractions and lacks the optimizations available in C and Java, further contributing to slower thread creation times.

6.1.3. Context switch time

- This was measured by computing the execution time of n threads to increment a shared variable.



C is the fastest in terms of context switches because it operates at a low level, with direct access to the operating system's threading and scheduling mechanisms. In C, context switches happen with minimal overhead due to its ability to make direct system calls, avoiding any additional layers of abstraction. However, despite C's advantages, the differences in context switching times across C, Java, and Python tend to be relatively small in practice. This is because, in all three languages, context switching is primarily managed by the underlying operating system and its kernel, which controls thread scheduling and resource allocation. Therefore, while C can offer faster context switches, all three languages are still subject to the same operating system constraints, leading to relatively close measurements in terms of context switching times. The differences, though present, may not be significant enough to drastically impact performance in many real-world scenarios.

6.1.4. Context switch time

- This was measured by explicitly starting a thread on core 0 and then changing it to run on core 1.

```
HANDLE thread = CreateThread(NULL, 0, thread_function, NULL, 0, NULL);


void set_thread_affinity(HANDLE thread, DWORD cpuCore) {
   QueryPerformanceFrequency(&f);
   QueryPerformanceCounter(&start_th_mig);

   // Set the thread's CPU affinity mask
   DWORD_PTR affinityMask = (1 << cpuCore);  // Move 1 to the position of the
specified CPU core
```

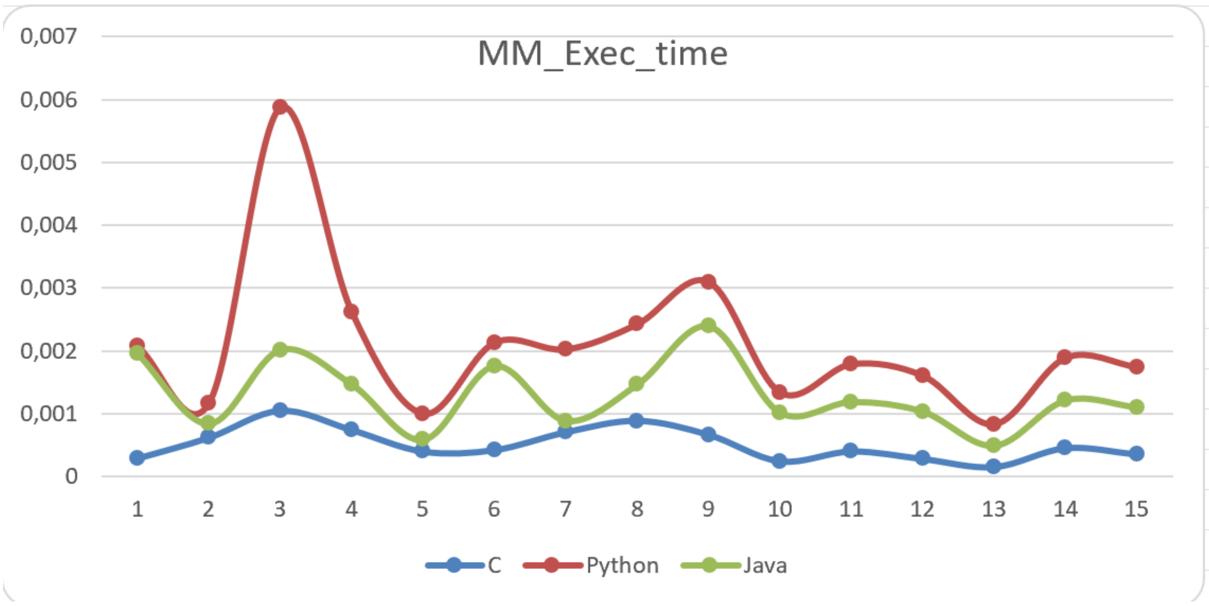
```
    if (SetThreadAffinityMask(thread, affinityMask) == 0) {
        printf("Error setting thread affinity\n");
    } else {
        printf("Thread affinity set to CPU core %d\n", cpuCore);
    }
    QueryPerformanceCounter(&end_th_mig);
}

DWORD WINAPI thread_function(LPVOID lpParam) {
    int* coreToUse = (int*)lpParam;
    set_thread_affinity(GetCurrentThread(), 0);
    DWORD currentCore = GetCurrentProcessorNumber();
    printf("Thread started on CPU core %d\n", currentCore);
    set_thread_affinity(GetCurrentThread(), 1);
    currentCore = GetCurrentProcessorNumber();
    printf("Thread is now running on CPU core %d\n", currentCore);
    return 0;
}
```

Java and Python do not allow explicit core assignment for threads because both rely on higher-level abstractions for portability and ease of use. Java's JVM delegates thread scheduling to the operating system to maintain platform independence, aligning with its "write once, run anywhere" philosophy. Similarly, Python's threading module abstracts away low-level hardware details, focusing on simplicity and cross-platform compatibility. Both languages rely on the operating system's scheduler to manage core assignments, as modern schedulers are optimized for efficient resource allocation without requiring explicit user intervention.

6.2. Matrix Multiplication

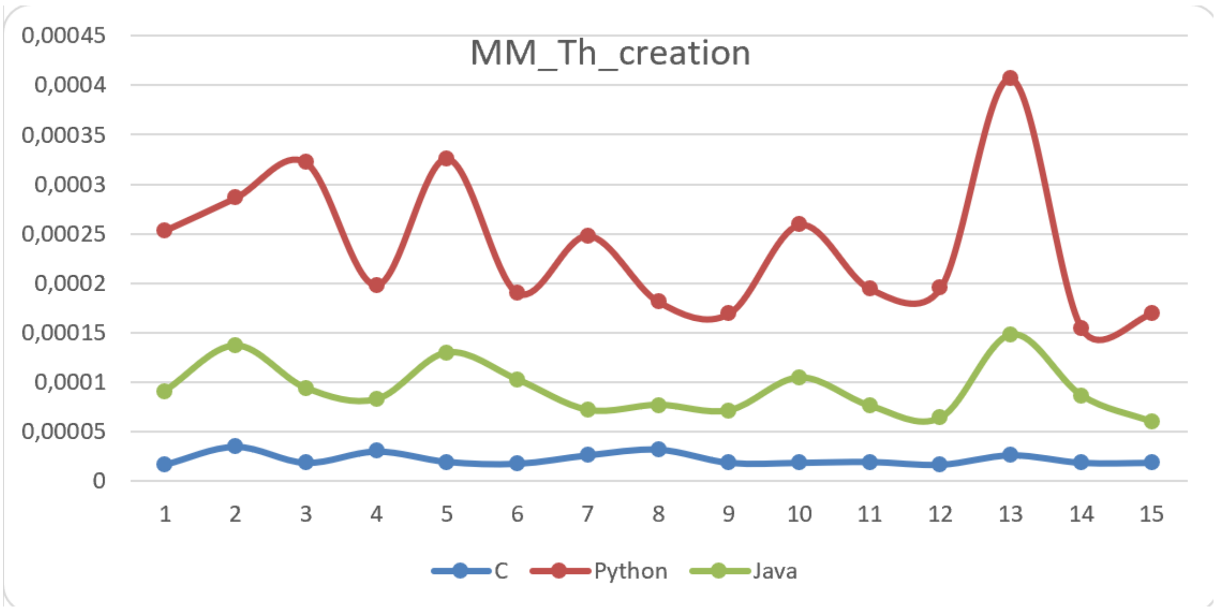
6.2.1. Execution time



Same as in the case of the parallel merge sort algorithm, C is the fastest, then Java, and then Python.

6.2.2. Thread creation time

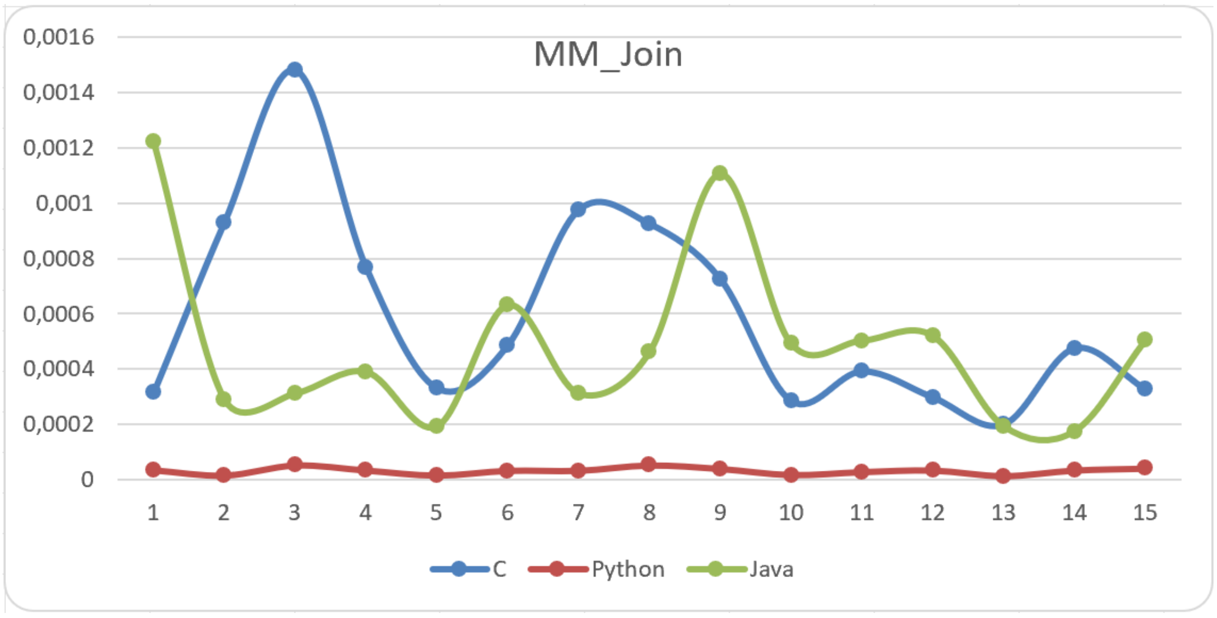
- Measured by summing up all thread creation times (for all threads which compute a row in the result matrix) and then computing the average thread creation time value.



Same as before, in the parallel merge sort algorithm, C has the best time performance, then Java and then Pyhton being the slowest.

6.2.3. Thread joining time

- In order to measure the context switch time and matching the measurements with the algorithm, we measured the time it took for joining the threads since there is most likely to appear context switches when a thread waits for another.



Java is the fastest when joining threads because its threading model is highly optimized within the JVM, which leverages Just-In-Time (JIT) compilation and efficient management of thread lifecycle. Python, despite the Global Interpreter Lock (GIL), performs well in thread joining because it delegates much of the underlying work to the operating system, similar to C. However, Python benefits from its simplified, high-level threading abstractions, which streamline the join process. C, while typically faster in other threading tasks, can appear slightly slower in thread joining due to the manual handling of low-level system calls like WaitForMultipleObjects, which may introduce minor synchronization overhead, making it close to Python in performance.

6.2.4. Thread migration time

- Similarly to the parallel merge sort algorithm measurements, the thread migration time was measured by starting a thread explicitly on core 0 and then setting it on core 1.

Results are also logged into a file.

This is an example for a run of each program:

```
(n) n = 100 elements
Parallel merge sort in C results:
        (ms-c) Execution time - dynamic: 0.002959 seconds
        (ms-c) Execution time - static: 0.002693 seconds
        (ms-c) Average execution time: 0.002826 seconds
        (ms-c) Average thread creation time: 0.000016 seconds
        (ms-c) Context switch time: 0.008915 seconds
        (ms-c) Thread migration time: 0.000030 seconds

Parallel merge sort in Python results
        (ms-py) Execution time: 0.004683 seconds.
        (ms-py) Average thread creation time: 0.000111 seconds.
        (ms-py) Context switch time: 0.011327 seconds.

Parallel merge sort in Java results:
        (ms-java) Execution time: 0.003584 seconds.
        (ms-java) Average thread creation time: 0.000071 seconds.
        (ms-java) Context switch time: 0.007288 seconds.


Parallel matrix multiplication in C results:
        (mm-c) Execution time - dynamic: 0.000518 seconds
        (mm-c) Execution time - static: 0.000188 seconds
        (mm-c) Average execution time: 0.000353 seconds
        (mm-c) Thread creation times:
                Thread 0: 0.000027100 seconds
                Thread 1: 0.000016000 seconds
                Thread 2: 0.000015000 seconds
                Thread 3: 0.000017900 seconds
                Thread 4: 0.000017400 seconds
                Thread 5: 0.000017000 seconds
                Thread 6: 0.000020300 seconds
                Thread 7: 0.000015200 seconds
                Thread 8: 0.000016600 seconds
                Thread 9: 0.000014400 seconds
        (mm-c) Average thread creation time: 0.000018 seconds
        (mm-c) Context switch time: 0.000329 seconds
        (mm-c) Thread migration time: 0.000029 seconds
```

Parallel matrix multiplication in Python results:

```
(mm-py) Execution time: 0.001742 seconds.
(mm-py) Thread creation times:
        Thread 0: 0.000673 seconds
        Thread 1: 0.000173 seconds
        Thread 2: 0.000125 seconds
        Thread 3: 0.000108 seconds
        Thread 4: 0.000108 seconds
        Thread 5: 0.000101 seconds
        Thread 6: 0.000102 seconds
        Thread 7: 0.000104 seconds
        Thread 8: 0.000100 seconds
        Thread 9: 0.000099 seconds
(mm-py) Average thread creation time: 0.000169 seconds
(mm-py) Context switch time: 0.000041 seconds
```

Parallel matrix multiplication in Java results:

```
(mm-java) Execution time: 0.001101 seconds.
(mm-java) Thread creation times (seconds):
        Thread 0: 0,000293800 seconds
        Thread 1: 0,000039300 seconds
        Thread 2: 0,000033000 seconds
        Thread 3: 0,000033000 seconds
        Thread 4: 0,000032900 seconds
        Thread 5: 0,000028500 seconds
        Thread 6: 0,000034500 seconds
        Thread 7: 0,000041900 seconds
        Thread 8: 0,000030500 seconds
        Thread 9: 0,000025100 seconds
(mm-java) Average thread creation time: 0.000059250 seconds
(mm-java) Context switch time: 0.000504300 seconds
```

7. Conclusions:

The project provided valuable insights into the performance characteristics and system-level behavior of C, Java, and Python when executing computationally intensive parallel tasks such as merge sort and matrix multiplication. C consistently demonstrated superior execution times, primarily due to its low-level optimizations and minimal runtime overhead. Java showed competitive performance with efficient thread management and context switching, benefiting from its robust multithreading capabilities in the JVM. Python, while offering simplicity and ease of implementation, lagged behind in execution time due to the Global Interpreter Lock (GIL) and higher-level abstractions that introduce latency. Thread creation times and context switches were notably faster in C, reflecting its lightweight threading model, while Java exhibited effective thread migration across cores, leveraging its dynamic runtime optimizations. Python's performance was less predictable, influenced by interpreter overhead and reliance on libraries, for performance-critical tasks. Overall, the results underscore the trade-offs between performance, ease of use, and system-level control among these languages.

8. Bibliography:

Online resources:

1. **Compiled Versus Interpreted Languages**, https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/, [accessed Oct 2024]
2. **Difference Between Compiled and Interpreted Language**, https://www.geeksforgeeks.org/difference-between-compiled-and-interpreted-language/, [accessed Oct 2024]
3. **How JVM Works: JVM Architecture**, https://www.geeksforgeeks.org/jvm-works-jvm-architecture/, [accessed Oct 2024]
4. **Understanding How Java Virtual Machine (JVM) Works**, https://hasithas.medium.com/understanding-how-java-virtual-machine-jvm-works-a1b07c0c399a, [accessed Oct 2024]
5. **JVM Tutorial: Java Virtual Machine Architecture Explained for Beginners**, vhttps://www.freecodecamp.org/news/jvm-tutorial-java-virtual-machine-architecture-explained-for-beginners/, [accessed Oct 2024]
6. **Multithreaded algorithms**, https://www.baeldung.com/cs/multithreaded-algorithms, [accessed Oct 2024]
7. **Static vs Dynamic memory allocation in C,** https://www.naukri.com/code360/library/difference-between-static-and-dynamic-memory-allocation-in-c, [accessed Oct 2024]
8. **Java Memory Management**, https://www.geeksforgeeks.org/java-memory-management/, [accessed Oct 2024]
9. **Python memory allocation**, https://www.geeksforgeeks.org/memory-management-in-python/, [accessed Oct 2024]
10. **Python GIL**, https://wiki.python.org/moin/GlobalInterpreterLock, [accessed Oct 2024]
11. **Creating threads in C in Windows**, https://learn.microsoft.com/en-us/windows/win32/procthread/creating-threads, [accessed Nov 2024]
12. **Creating excel charts**, https://www.geeksforgeeks.org/python-plotting-charts-in-excel-sheet-using-openpyxl-module-set-1/, , [accessed Dec 2024]

Books:

1. Butenhof, David R., *Programming with POSIX Threads*, Addison-Wesley, 1997, pp. 1-432.
2. Goetz, Brian, *Java Concurrency in Practice*, Addison-Wesley, 2006, pp. 1-352.
3. Beazley, David, Jones, Brian K., *Python Cookbook*, O'Reilly Media, 2013, pp. 1-706.