*Execution time measurements of processes in different programming languages*

*Student: Minuț Vlad-Costin*

*Group: 30433*

*Teacher: Zoltan Czako*

**Contents**

1. Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . pg 3
2. Objectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . pg 5
3. Theoretical considerations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . pg 6
4. Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . pg 10
5. Testing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . pg 17
6. Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . pg 21
7. *Introduction*

**What is execution time?**

Execution time, also known as CPU time of a task represents the time that the CPU is spending to execute the task, including the time spent executing run time or system services (in contrast to other program lifecycle phases such as compile time, link time and load time).

A run time error is detected after or during the execution (running state) of a program, while a compile time error is detected by the compiler before the program is ever executed. Type checking, register allocation, code generation and code optimization are typically done at compile time, but may be done at run time depending on the particular language and compiler. Many other run time errors exist and are handled differently depending on languages, such as division by zero errors.

**How is a program executed?**

When a program is executed, a loader first performs the necessary memory setup and links the program with any dynamically lined libraries it needs and then the execution begins starting from the program’s entry point. In some cases, a language or implementation will have these tasks done by the language runtime instead, though it is unusual in common languages/customer operating systems.

**Memory allocation**

Memory allocation is a process by which computer programs and services are assigned with physical or virtual memory space. Memory allocation is primarily a computer hardware operation but it is managed through operating system and software applications. The process is quite similar in physical and virtual memory management. Programs are assigned with a specific memory for their requirements when they are executed. Once the program has finished its operation, the memory is released.

There are two types of memory allocation:

* Static memory allocation (allocation at compile time);
* Dynamic memory allocation (allocation at run time).

**Threads**

A thread (of execution) is a way for a program to divide into two or more simultaneously running tasks. They are lightweight in terms of system resources they consume, comparing to processes. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within a one process, executing concurrently and sharing resources such as memory, while different processes do not share resources.

In particular, the threads of a process share its executable code and the value of its dynamically allocated variables and non-thread-local global variables at any given time.

1. *Objectives*

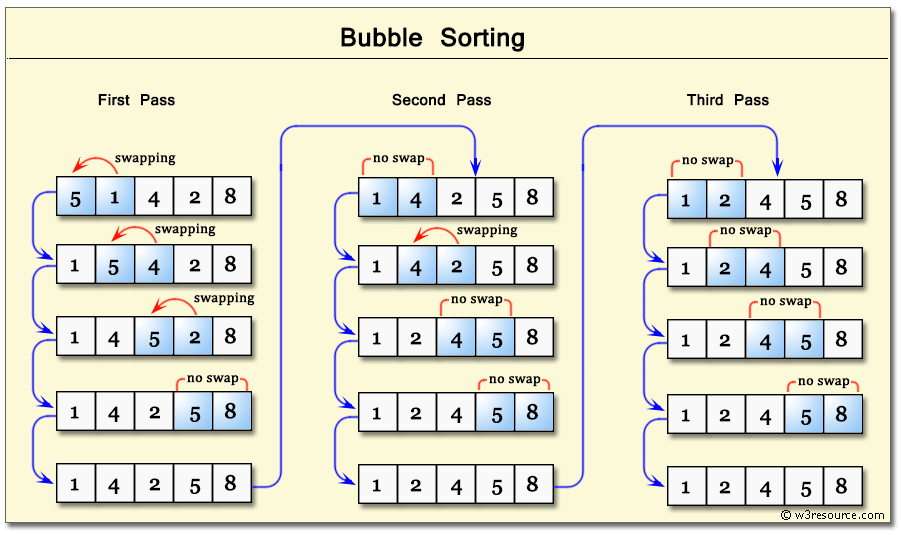
The aim of this project is measuring and comparing the execution time of certain processes in two or three different programming languages, such as C, C++, Java, C#.

The targeted processes are regarding memory: memory allocation and memory accessing (both static and dynamic); threading: thread creation, thread context switch and thread migration.

After all the measurements are done, a chart with all the execution times will be created to compare and draw the conclusions from all the information collected.

1. *Theoretical considerations*

In order to measure the execution time of processes, we will need a function already implemented so we can work with it. The function that we will use will be a classical sorting method: Bubble sort (also known as Sinking sort).



It is a simple sorting algorithm, which repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the array or the list is sorted. Although the algorithm is simple, it is considered to be slow and impractical for most problems even if we compare it to Insertion sort. Bubble sort can be practical if the input is in mostly sorted with some elements that are close to their correct position.

Complexity

The bubble sort worst-case complexity is O(n2), n being the number of elements that need to be sorted. The number of comparisons and the number of swaps in this case are both n2.

In the average case, the algorithm performs the same number of comparisons and swaps as in the worst case, which makes the average-case complexity equal to O(n2).

Lastly, the best case performance for bubble sort means n comparisons and only one swap, which means O(n).

The reason we use Bubble sort is that it accesses memory and operates with it.

Further on, we will use another algorithm to measure the performance of the threading part of the project. We will have a text and the task will be to count its total number of characters. In order to do that, the text will be divided in n equal parts, each one of them being having its letters counted by one thread.

The complexity of this algorithm is approximated at n/t, where n is the number of letters and t is the number of threads, so it is a linear algorithm.

Thread context switch vs process context switch

The main distinction between a thread switch and a process switch is that during a thread switch, the virtual memory space remains the same, while it does not during a process switch. Both types involve handing control over to the operating system kernel to perform the context switch. The process of switching in and out of the OS kernel along with the cost of switching out the registers is the largest fixed cost of performing a context switch.

A more fuzzy cost is that a context switch messes with the processors cacheing mechanisms. Basically, when you context switch, all of the memory addresses that the processor "remembers" in its cache effectively become useless. The one big distinction here is that when you change virtual memory spaces, the processor's Translation Lookaside Buffer (TLB) or equivalent gets flushed making memory accesses much more expensive for a while. This does not happen during a thread switch.

Tools for measuring

1. Java

There are two ways to measure the elapsed execution time in Java. One of them is by using  System.currentTimeinMillis() and the other one by using System.nanoTime().These two methods can be used to measure execution time between two method calls or event in Java. Calculating elapsed time is one of the first thing a Java programmer does to find out how many seconds a method is taking to execute or how much time a particular code block is taking. Most of the Java programmers are used to the first method mentioned above whichi exists from the beginning, while the second method is introduced in Java 1.5, along several new features.

Ideally, System.currentTimeinMillis() should be used to measure wall-clock time and System.nanoTime() should be used to measure the elapsed time of the program. If the elapsed time is measured with the first method, the results might not be that accurate.

1. C/C++

To compute the time taken by a process in C, we can use the clock(); function

which is available in the time.h library. We can call the clock function at the

beginning and the end of the code for which we measure time, substract the

values and then divide by CLOCKS\_PER\_SEC (the number of clock ticks per

second) to get processor time.

C#

We might need to know how to find out the execution time of a particular segment of code in C# for various reasons, for example for optimizing the performance of the program. For example we may want to know how much time is taken for reading multiple files in the file system, or fetching data from a database or executing business logic.

C# includes the Stopwatch class in System.Diagnostics namespace, which can acuurately measure the time taken for code execution. You don’t need to use DateTime and calculate the time manually.

*IV.Implementation*

As I have written before, I will use the Bubble Sort for measuring memory allocation and (static and dynamic) memory access. For measuring the threading part, I will use a program that counts the number of vowels in a text, using n threads. The text will be divided in n equal parts and each thread will count the number of vowels in its part, then the total sum will be computed and returned/printed.

1. Bubble sort
2. The function ‘swap(int, int)’ uses a temporary value to swap two numbers and it will be used for the sort.

Bubble sort uses two loops to iterate over an array and compare the adjacent elements. If their order is not the correct one, they are swapped. The function ‘print()’ prints the sorted array.

In the main function, the user can input the number of elements in the array.

#include <iostream>

#include <stdlib.h>

#include <time.h>

using namespace std;

void swap(int \*a, int \*b)

{

int \*aux;

aux = a;

a = b;

b = aux;

}

void bubbleSort(int \*arr, int n)

{

int i, j;

for (i = 0; i < n - 1; i++)

for (j = 0; j < n - i - 1; j++)

if (arr[j] > arr[j + 1])

swap(arr[j], arr[j + 1]);

}

void print(int \*arr, int n)

{

cout << "The sorted array is: "<<endl;

for (int i = 0; i < n; i++)

cout << arr[i] << ' ';

}

int main()

{

clock\_t start, end;

double cpu\_time\_used;

int \*arr = NULL;

int n;

cout << "Please insert the size of the array: "<<endl;

cin >> n;

for (int i = 0; i < n; ++i)

{

arr[i] = (rand() % 100) + 1;

cout << arr[i] << ", ";

if (i = n - 1)

{

cout << " that's randomly generated." << endl;

}

}

start = clock();

bubbleSort(arr, n);

print(arr,n);

end = clock();

cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC;

cout << cpu\_time\_used<<" seconds";

while(1);

return 0;

}

1. Java

The task is executed using one class , “BubbleSort”, containing three methods, “bubbleSort”, “printArray”, and the main method.

import java.text.DecimalFormat;  
import java.util.Random;  
  
public class BubbleSort  
{  
 void bubbleSort(int arr[])  
 {  
 int n = arr.length;  
 for (int i = 0; i < n-1; i++)  
 for (int j = 0; j < n-i-1; j++)  
 if (arr[j] > arr[j+1])  
 {  
 int temp = arr[j];  
 arr[j] = arr[j+1];  
 arr[j+1] = temp;  
 }  
 }  
  
 void printArray(int arr[])  
 {  
 int n = arr.length;  
 for (int i=0; i<n; ++i)  
 System.*out*.print(arr[i] + " ");  
 System.*out*.println();  
 }  
  
 public static void main(String args[])  
 {  
  
 BubbleSort ob = new BubbleSort();  
 int n = 100000;  
 int [] arr = new int[n];  
 Random rd = new Random();  
 for(int i=0; i<n; ++i)  
 {  
 arr[i] = rd.nextInt()%100000;  
 }  
 double startTime = System.*nanoTime*();  
 ob.bubbleSort(arr);  
  
 double elapsedTime = System.*nanoTime*() - startTime;  
  
 elapsedTime/=1000000000;  
 System.*out*.println("Sorted array");  
 //ob.printArray(arr);  
 System.*out*.println(new DecimalFormat("#.##########").format(elapsedTime));  
 }  
}

1. Functions using threads
2. C++

This code uses the library ‘thread’ for the threading part. The function ‘numara(char, int, int, int)’ counts the number of vowels in a text between two positions given as parameters and stores the result in a counter given as parameter. (the result is given as parameter to simplify the use of threads). The variable “lockMutex” is used for synchronizing the threads so they will not overlap.

In the main function, I created a text already initialized, three threads that will count the number of vowels in three different parts of that text, and three variables to store the results. The number of threads is picked by the user.

After the three results have been determined, the threads are joined and the final result is computed and printed.

#include <iostream>

#include <thread>

#include <ctime>

#include <vector>

#include <mutex>

using namespace std;

std::mutex lockMutex;

void numara(char \*text, int pos1, int pos2, int \* counter)

{

for (int i = pos1; i < pos2; ++i)

if (text[i] == 'a' || text[i] == 'e' || text[i] == 'i' || text[i] == 'o' || text[i] == 'u')

{

lockMutex.lock();

(\*counter)++;

lockMutex.unlock();

}

}

int main()

{

char globaltext[100];

int ctr, total, end, start, n;

clock\_t start\_time, end\_time, mid\_time;

double time\_used, time\_used\_join;

int stop;

strcpy\_s(globaltext, "To be measured text");

cout << "introduceti numarul de threadrui: ";

cin >> stop;

n = strlen(globaltext)/stop;

total = ctr = 0;

start\_time = clock();

std::vector<std::thread\*> thread\_vector;

for (int i = 0; i < stop; i++)

{

ctr = 0;

if (i == stop - 1)

{

end = strlen(globaltext);

}

else

{

end = (i + 1)\*n;

}

start = i \* n;

thread\_vector.push\_back(new thread(numara, globaltext, start, end, &ctr));

total+= ctr;

}

for (int i = 0; i < stop; i++)

{

thread\_vector[i]->join();

}

end\_time = clock();

time\_used = ((double)(end\_time - start\_time)) / CLOCKS\_PER\_SEC;

cout << endl;

cout << globaltext << endl << endl << "Textul de mai sus contine: ";

cout << total<<" vocale"<<endl;

cout << "CPU time: " << time\_used << " seconds"<<endl;

while (1);

return 0;

}

1. Java

The measurements for the threading part in Java are done using a muchi simpler program. It creates n threads using the Runnable interface. Each thread displays a message and they are joined. There are two classes “MultithreadingExample” and “Main”. In the first class, we can find the job that each thread does: it prints its id, while in the second class we can find the thread creation, the measurements and the thread join.

public class MultithreadingExample implements Runnable{  
  
 @Override  
 public void run() {  
 try{  
 System.*out*.println("Thread"+Thread.*currentThread*().getName()+"is now running");  
 }  
 catch (Exception e){  
 System.*out*.println("Exception");  
 }  
 }  
}

public class Main {  
 public static void main(String[] args) {  
  
 double startTime = System.*nanoTime*();  
 int noThreads = 100000;  
 Thread[] threadsArray = new Thread[noThreads];  
 for(int i = 0; i<noThreads;i++)  
 {  
 threadsArray[i] = new Thread(new MultithreadingExample());  
 threadsArray[i].start();  
 }  
 for (int i=0; i<noThreads;i++)  
 {  
 try {  
 threadsArray[i].join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 double elapsedTime = System.*nanoTime*() - startTime;  
  
 elapsedTime/=1000000000;  
  
 System.*out*.println(elapsedTime);  
 }  
}

1. Implementing the tools for measuring execution time

This method requires declaring two variables that can measure the current time in each language, one before the process we need to measure is executed, and one after that. At the end, we will substract them and convert them into numbers representing seconds.

1. C++ (using <ctime.h>)

clock\_t start\_time, end\_time;

double total\_time;

start\_time = clock();

code\_to\_measure();

end\_time = clock();

total\_time = ((double) (end\_time-start\_time))/ CLOCKS\_PER\_SEC;

1. Java (using java.lang.\*)

long start, finish, total\_time;

start = System.nanoTime();

codeToBeMeasured();

finish = System.nanoTime();

total\_time = (finish-start) / pow(10, 9);

1. C#

var watch = System.Diagnostics.Stopwatch.StartNew();

codeToMeasure();

watch.Stop();

var elapsedTime = watch.ElapsedMilliseconds;

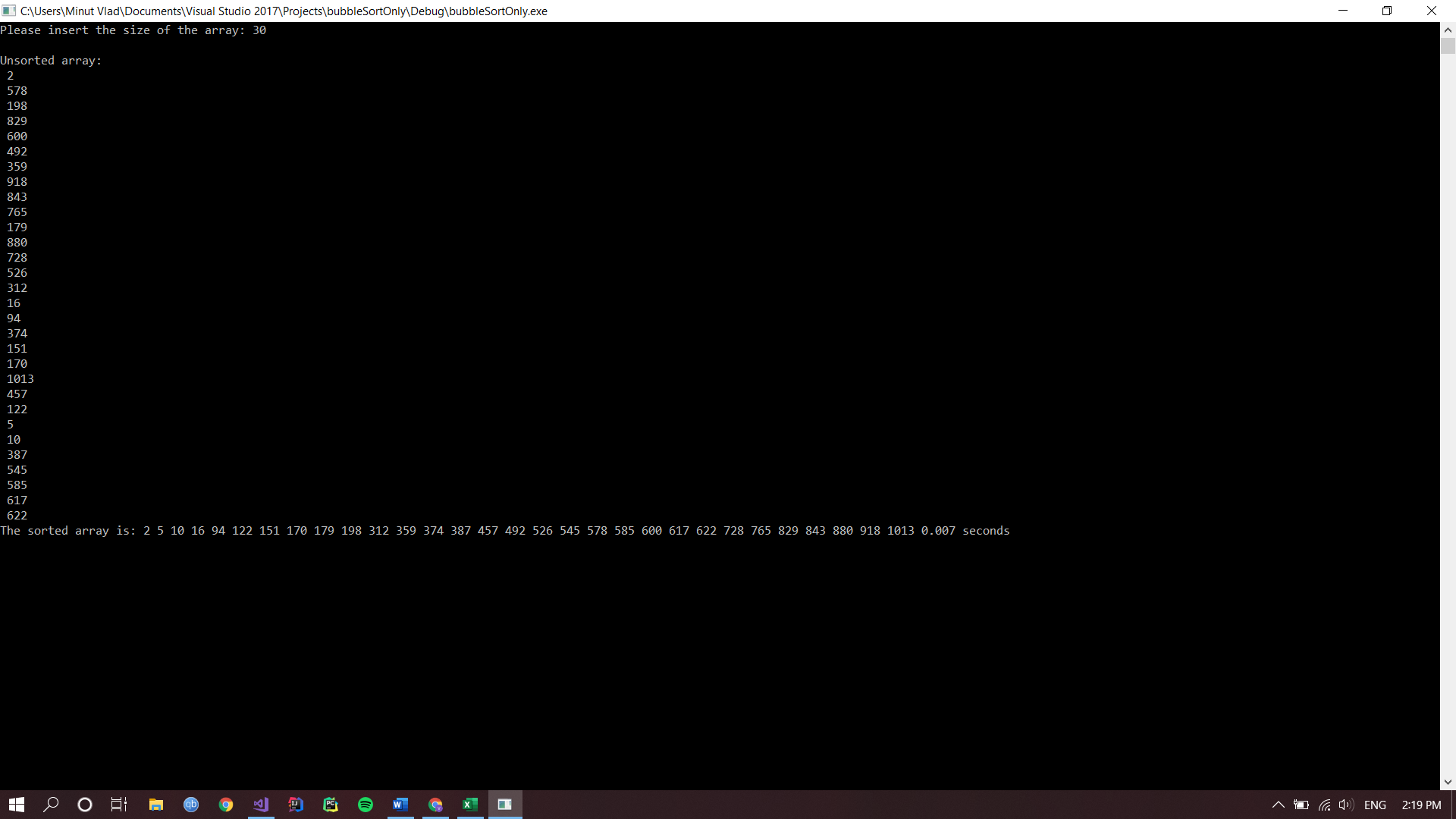
elapsedTime = elapsedTime/1000;

*V. Testing*

To test the execution time, I created some graphics using excel.

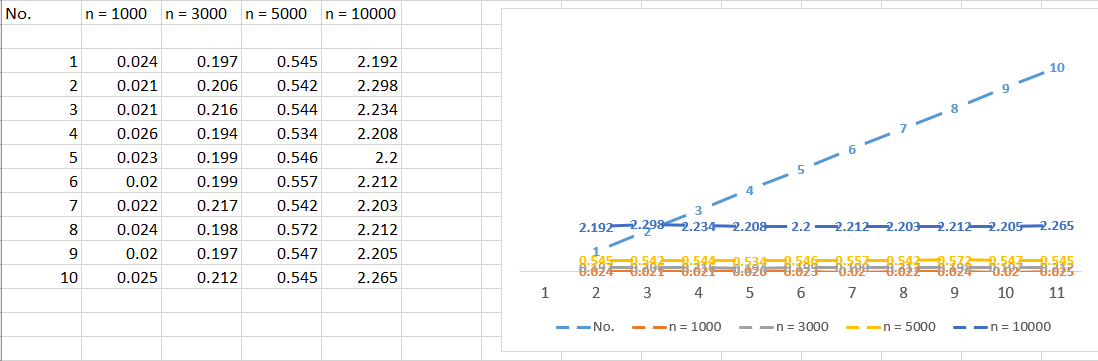
For testing the execution time of bubble sort I used a randomly generated array with n = 1000, n= 3000, n = 5000, n = 10000. n is the size of the array.

The output of the program will show:



For that example, I used n = 30 because for higher sizes the array would not fit into the screen.

The actual measurement:



We can observe that the time used for n = 1000, n = 3000 are really close, but when it comes to n = 10000 the time used gets longer, because bubble sort has the complexity O(n2).

For testing the execution time of the threading part of the project, I ran the algorithm described in the implementation using 3 threads, 10 threads and 20 threads.

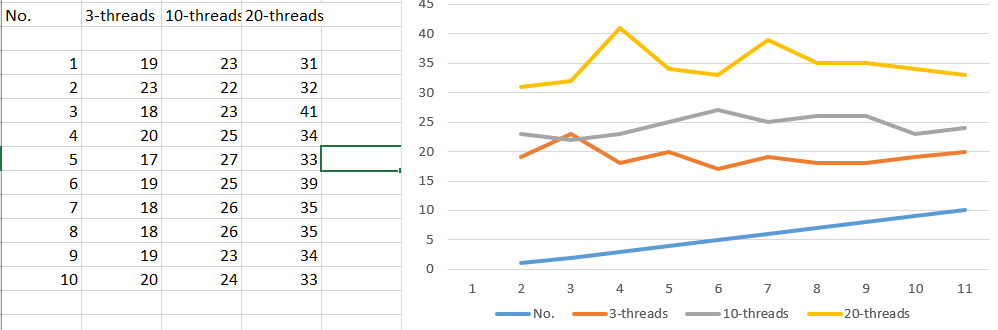
The output of the program is:



The number of threads is read from the keyboard, then the output means:

* The text to be measured;
* The number of vowels of the text;
* The execution time of the program.

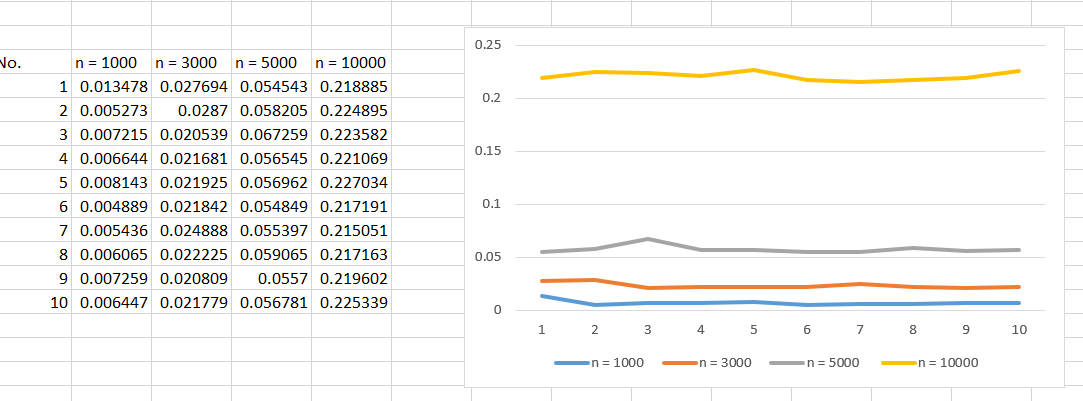
The excel graph looks like this:



Every result is multiplied with 100 because the graph looks better if the “No.” (number of the measurement) is displayed too.

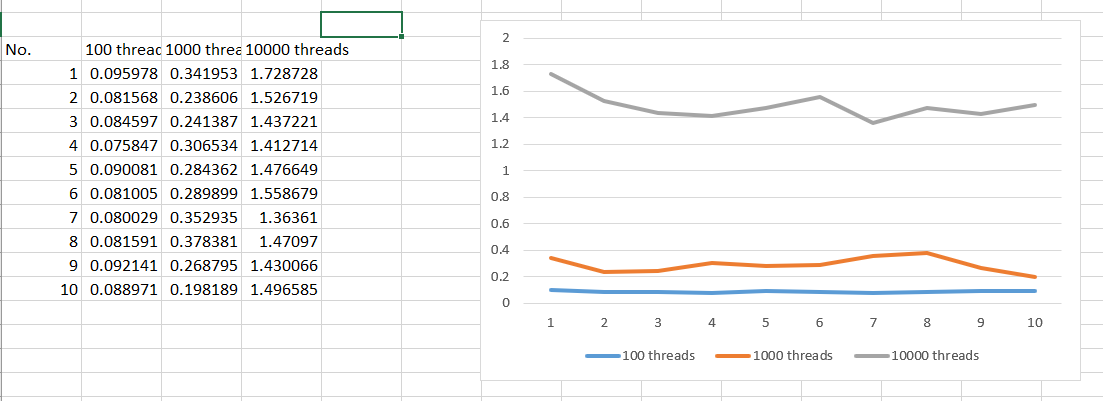
Testing in Java:

Bubble Sort



These are the results of the Bubble Sort in Java. The array is randomly generated each time and n is the number of elements of the array. The time is displayed in seconds.

Threading:



The measuremets were made for 100, 1000 and 10000 threads. The results are displayed in seconds.

1. *Conclusions*

To conclude, the measurements for Bubble Sort were made in both C++ and Java and the result shows that the Java version is a little bit faster.

Also, neither in C++, nor in Java, the execution time is not proportional with the number of threads created/joined or with the number of elements of an array.