Benchmark for PC Performance Parameters

Horvath Ariana-Cristine

2021-2022

Table of Contents

1. Introduction

1.1 Context

A **benchmark** is a test that measures the performance of hardware, software or computer. These tests help compare how well a product may do against other products. When comparing benchmarks, the higher the value of the results, the faster the component, software, or overall computer is.

1.2 Objectives

The goal of this project is to design, implement and test a benchmark for PC performance parameters, such as the type of the processor, the frequency, the memory size, the transfer speed of a block of data and the execution speed of some arithmetical and logical operations.

The program should be used by anyone who wants to test the performance of his/hers PC, on any PC.

1.3 Specifications

The program code with be written in Visual Studio IDE by Microsoft using C++ and the graphical interface of the application will be written IntelliJ IDEA by JetBrains using Java and the Spring Framework. It should be very well organized in packages and classes. The results of the benchmarking will be stored in files in order to be displayed afterwards on the GUI.

2. Bibliographic study

- For the PC performance counters we can use the function GetSystemInfo[2] with the stucture SYSTEM_INFO[3], for getting:
  - *dwProcessorType:* the processor type: PROCESSOR_INTEL_386 (386); PROCESSOR_INTEL_486 (486); PROCESSOR_INTEL_PENTIUM (586); PROCESSOR_INTEL_IA64 (2200); PROCESSOR_AMD_X8664 (8664); PROCESSOR_ARM (Reserved)
  - *dwPageSize*: the page size and the granularity of page protection and commitment.
  - *lpMinimumApplicationAddress:* A pointer to the lowest memory address accessible to applications and dynamic-link libraries (DLLs).
  - *lpMaximumApplicationAddress:* A pointer to the highest memory address accessible to applications and DLLs.
  - *dwActiveProcessorMask*: A mask representing the set of processors configured into the system. Bit 0 is processor 0; bit 31 is processor 31
  - *dwNumberOfProcessors:* The number of logical processors in the current group.
  - *dwAllocationGranularity:* The granularity for the starting address at which virtual memory can be allocated.
  - *wProcessorLevel:* The architecture-dependent processor level. It should be used only for display purposes. If *wProcessorArchitecture* is PROCESSOR_ARCHITECTURE_INTEL, *wProcessorLevel* is defined by the CPU vendor.If wProcessorArchitecture is PROCESSOR_ARCHITECTURE_IA64, *wProcessorLevel* is set to 1.

```
void GetSystemInfo(
  [out] LPSYSTEM_INFO lpSystemInfo
);
typedef struct _SYSTEM_INFO {
  union {
    DWORD dwOemId;
    struct {
      WORD wProcessorArchitecture;
      WORD wReserved;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
  DWORD     dwPageSize;
  LPVOID    lpMinimumApplicationAddress;
  LPVOID    lpMaximumApplicationAddress;
  DWORD_PTR dwActiveProcessorMask;
  DWORD     dwNumberOfProcessors;
  DWORD     dwProcessorType;
  DWORD     dwAllocationGranularity;
  WORD      wProcessorLevel;
  WORD      wProcessorRevision;
} SYSTEM_INFO, *LPSYSTEM_INFO;
```

- For getting the RAM memory size: *GetPhysicallyInstalledSystemMemory*[4] function (sysinfoapi.h)
- For getting the processor's frequency (max frequency and current frequency): PROCESSOR_POWER_INFORMATION[5] structure and *CallNtPowerInformation*[6] function (powerbase.h)

```
typedef struct _PROCESSOR_POWER_INFORMATION {
  ULONG Number;
  ULONG MaxMhz;
  ULONG CurrentMhz;
  ULONG MhzLimit;
  ULONG MaxIdleState;
  ULONG CurrentIdleState;
} PROCESSOR_POWER_INFORMATION, *PPROCESSOR_POWER_INFORMATION;
```

```
NTSTATUS CallNtPowerInformation(
  [in]  POWER_INFORMATION_LEVEL InformationLevel,
  [in]  PVOID                   InputBuffer,
  [in]  ULONG                   InputBufferLength,
  [out] PVOID                   OutputBuffer,
  [in]  ULONG                   OutputBufferLength
);
```

3. Analysis

3.1 Requirements

3.1.1 Functional requirements

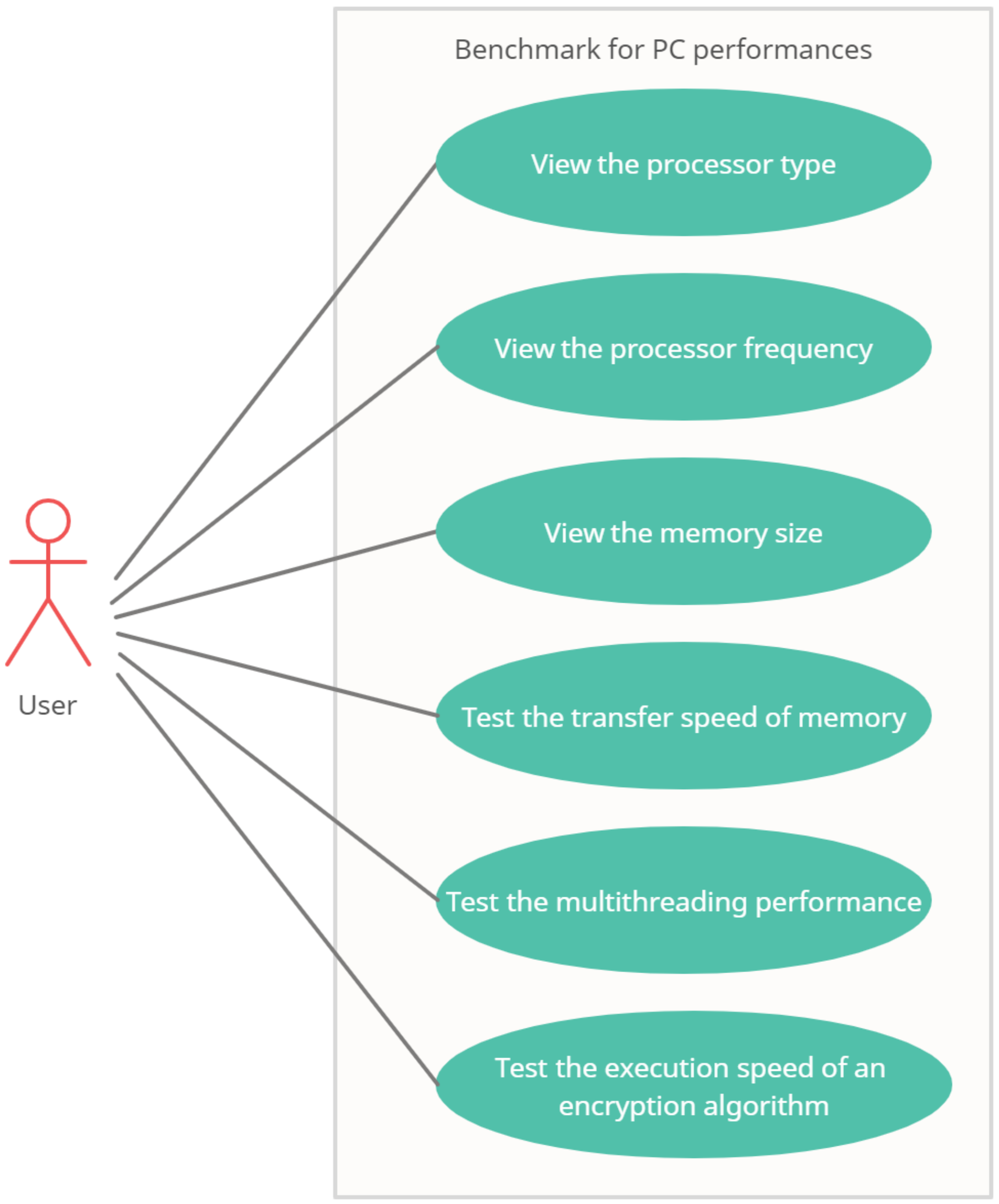
- The application should allow users to choose which test to do and display the results of the chosen test

- The application should display the type of the processor
- The application should display the frequency of the processor
- The application should display the memory size
- The application should test the transfer speed of a block of data, allocating and deallocating memory
- The application should test the execution speed of an encryption algorithm
- The application should test the multicore by a multithread algorithm

3.1.2 Non-Functional requirements

- The application should be intuitive and easy to use by the user
- The application should notify the users when the tests performed are successful
- The graphical interface should be organized and readable

3.2 Use-cases and scenarios


Benchmark for PC performances
View the processor type
View the processor frequency
View the memory size
Test the transfer speed of memory
Test the multithreading performance
Test the execution speed of an encryption algorithm
User

Use Case: Any use case from above

Primary Actor: user

Main Success Scenario:

1. The user selects (by pressing a button) the performance counter to be seen or the testing to be performed.
2. The user sees the results displayed on the GUI.
3. An information message is shown to let the user know that the operation was performed successfully.

Alternative Sequence: none

The inputs of the system are not proper inputs, the user only selects what he/she wants to see.

The outputs are the results of the testing made or the displayed PC performance parameters.

3.3 Data structures and algorithms

Encription Algorithm – for computing testing: RSA Algorithm (Rivest–Shamir–Adleman)[7]

- the algorithm is used by modern computers to encrypt and decrypt messages. It is an asymmetric cryptographic algorithm. Asymmetric means that there are two different keys. This is also called public key cryptography, because one of the keys can be given to anyone. The other key must be kept private.

1: Creating Keys

- Select two large prime numbers x and y
- Compute $n = x * y$ where n is the modulus of private and the public key
- Calculate totient function, $ø(n) = (x - 1)(y - 1)$
- Choose an integer e such that e is coprime to ø(n) and $1 < e < ø(n)$. e is the public key exponent used for encryption
- Now choose d, so that $d \cdot e \bmod ø(n) = 1$, i.e., >code>d is the multiplicative inverse of e in mod ø (n)

2: Encrypting Message

- Messages are encrypted using the Public key generated and is known to all.
- The public key is the function of both e and n i.e. {e,n}.
- If M is the message(plain text), then ciphertext C = M ^ n(mod n)

3: Decrypting Message

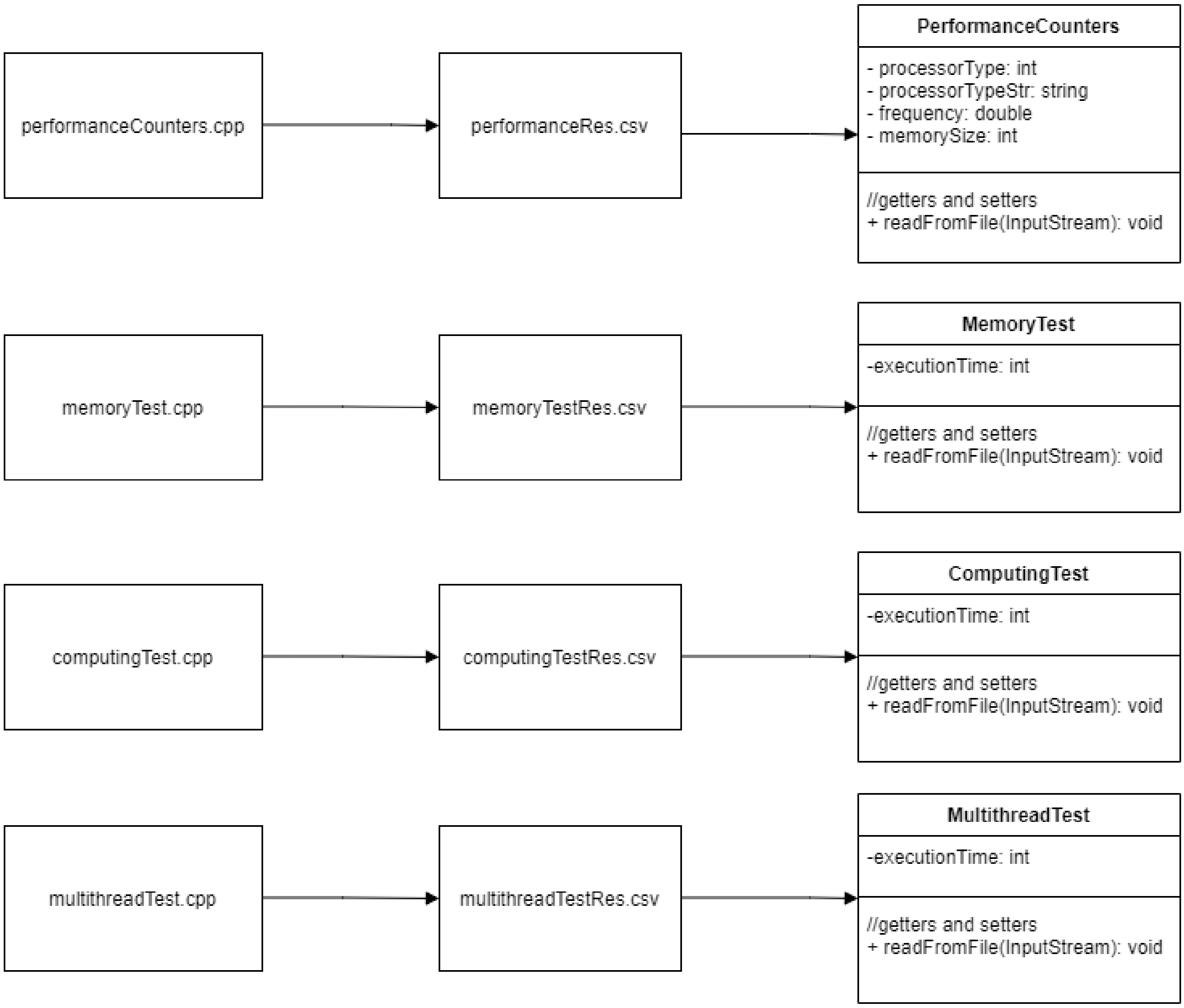
- The private key is the function of both d and n i.e {d,n}.
- If C is the encrypted ciphertext, then the plain decrypted text M is M = C ^ d (mod n)

4. <u>Design</u>

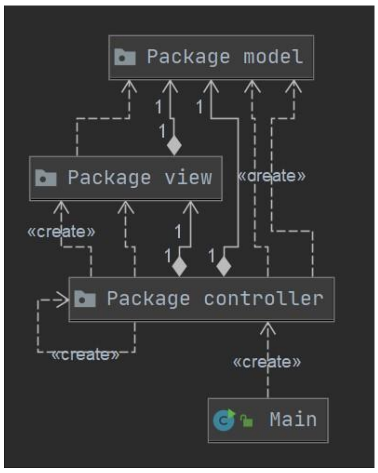
4.1 <u>Data Model</u>

Each part of the benchmark – reading the performance counters (memory size, type of the processor, frequency, etc.), allocation of memory testing (transfer speed of data), multicore testing (multithreading algorithm for counting the number of vowels from a text using different number of threads) and computing testing (encryption algorithm) – will be in a separate .cpp file. The results will be written in different .txt files and displayed in a more appealing way on a graphical user interface made in Java using Apache Maven. The performance counters file will contain the values separated with comma and the others will contain only one value, the execution time measured in miliseconds (ms).

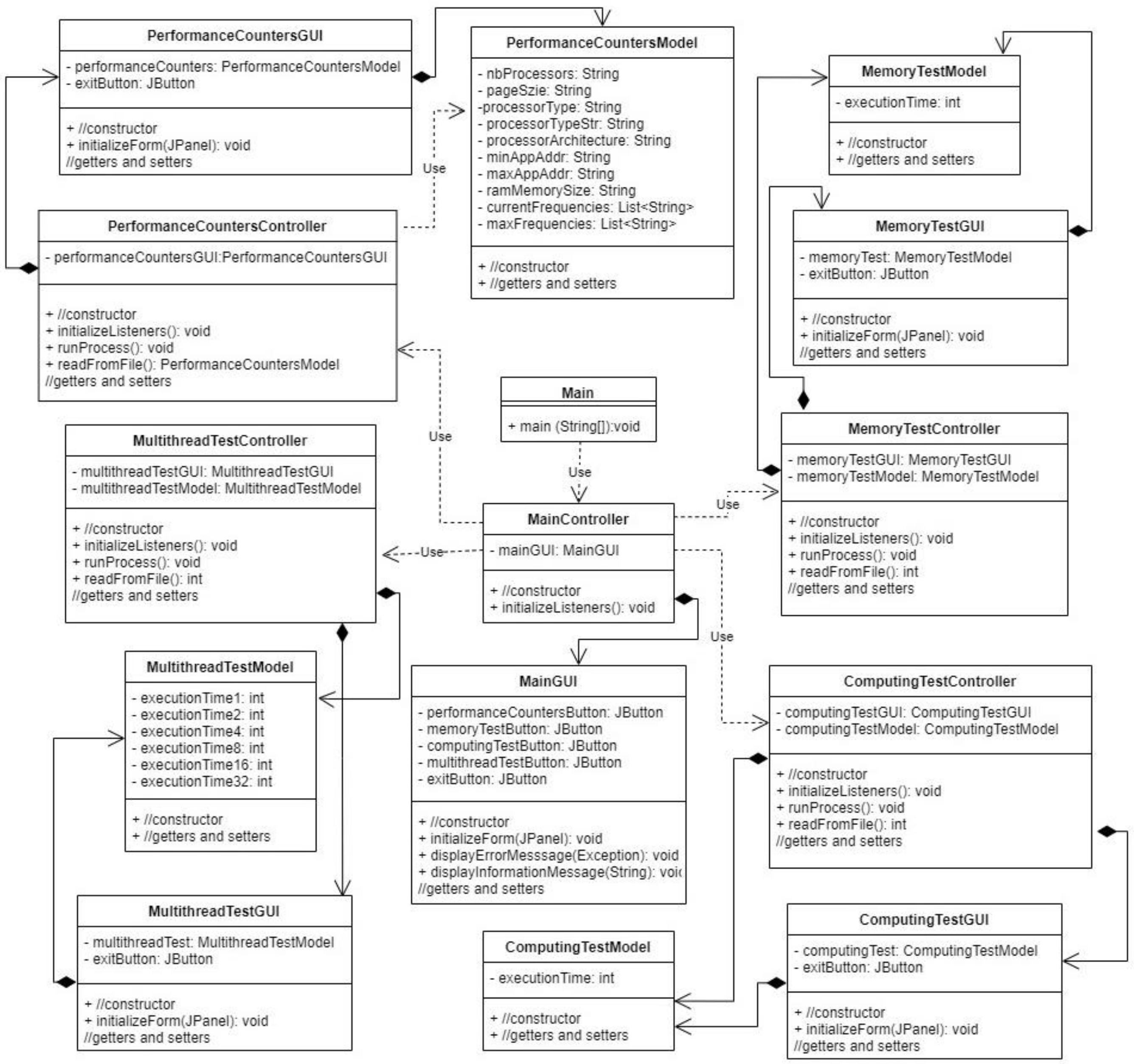
4.2 <u>Components</u>



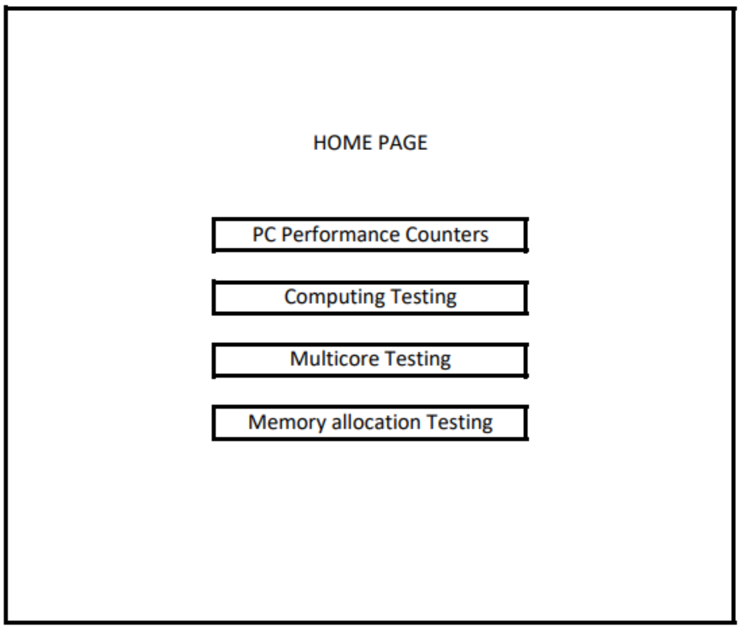
4.3 Package Diagram

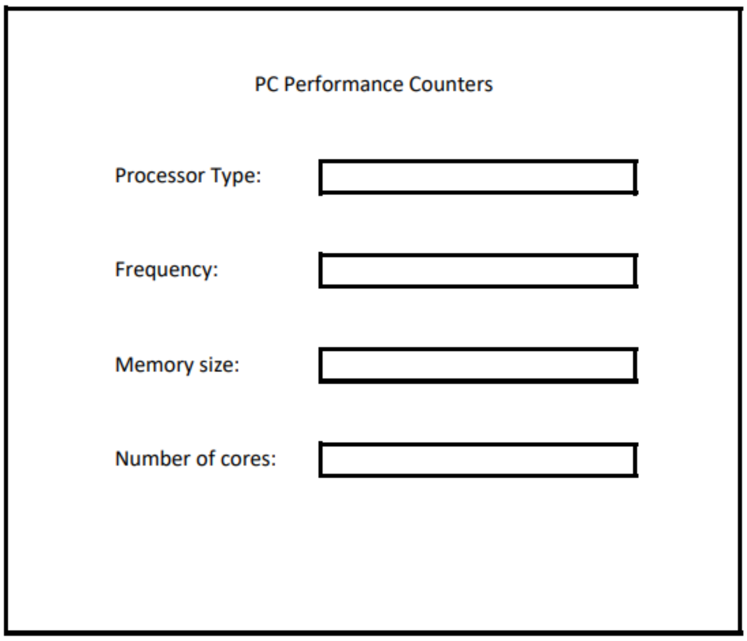


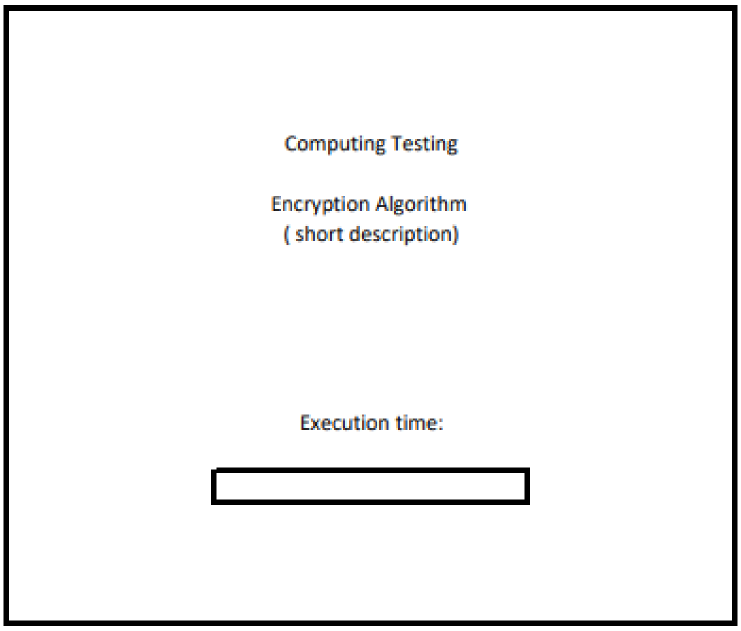
4.4 Class Diagram



4.5 GUI Design – Sketch



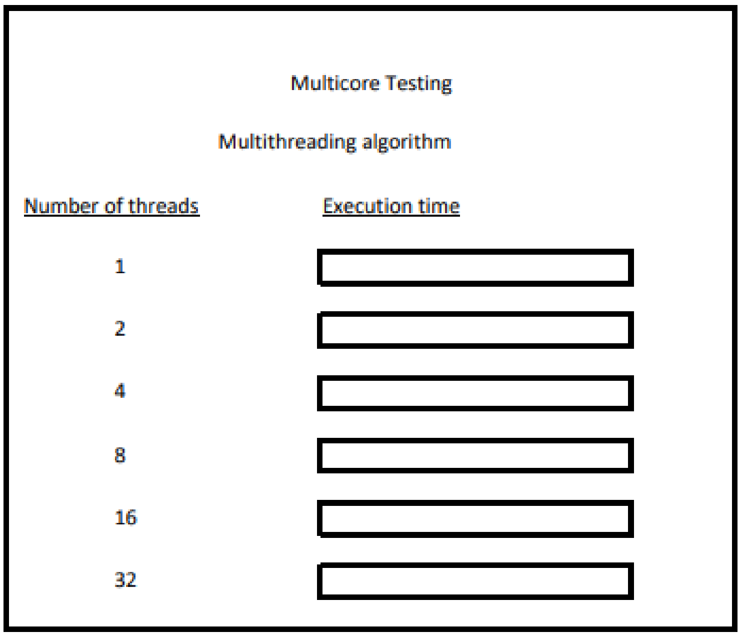




Computing Testing

Encryption Algorithm
(short description)

Execution time:



Multicore Testing

Multithreading algorithm

Number of threads	Execution time
1	
2	
4	
8	
16	
32	

5. Implementation

- *performanceCounters.cpp*

  The information about the processor (number of cores, type, architecture), the page size, the minimum and maximum application addresses and the maximum and current frequencies of each core are written in a file by calling the *GetSystemInfo* function and the RAM memory size by calling the *GetPhysicallyInstalledSystemMemory* function, both from *sysinfoapi.h.*

- *multithreadingTest.cpp*

  A random text of 1.000.000 characters is read from a .txt file and its length is divided almost equally to the number of threads that the program has to be given as command line argument (if not, the default number of threads is 8). The functions used are specific for threads on Windows. Each thread counts the vowels of the part of text that was given to it. The partial counts are stored in an array and then, when all threads had given their partial results, a sum is made with these partial counts. The execution time in miliseconds is computed, for different number of threads (1, 2, 4, 8, 16, 32). The operations are repeated 1000 times and the average execution time is written in a file, but the intermediary results are saved in a vector in order to generate a plot.

- *memoryTest.cpp*

  An array of 1.000.000 integers is allocated dynamically in memory; each element is given as value its position in the array. Then, a sum is made with all the elements (in order to access the memory) and the memory is freed. The operations are repeated 1000 times and the average execution time is written in a file, but the intermediary results are saved in a vector in order to generate a plot.

- *computingTest.cpp*

  For the computing test, the Rivest–Shamir–Adleman Encryption Algorithm (the algorithm itself is described in section 3.3) is used to encrypt and decrypt a text of 10.000 characters. The operations are repeated 10 times and the average execution time is written in a file, but the intermediary results are saved in a vector in order to generate a plot.

- Java Application

  The scope of the Java part is to display the results of benchmarking to the user in a more friendly and appealing way. It launches the C++ processes which write the results in files, reads the contents of the file and displays the results in the graphical user interface.

  The class files are divided in three packages: model, view and controller.

  The Main class starts the application.

  The view classes contain the layout for the GUI.

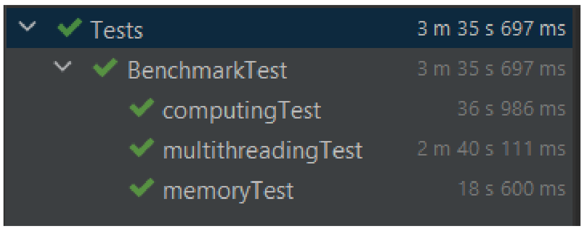
  The model classes of the computing test and memory test contain only the execution time field, the multithreading test contains the execution times for 1, 2, 4, 8, 16 and 32 number of threads and the model for performance counters contains the number of processors, page size, processor type, processor architecture, minimum application address, maximum application address, RAM memory size, current and maximum frequencies for each core.

The controller classes make the correspondence between the model and the view classes. Each one has the runProcess() and readFromFile() methods and is initialized when pressing the button from the main page, when the corresponding process is run and the corresponding file is read. The corresponding frame with the results is displayed only after the process terminates.

```
public void runProcess() throws IOException, InterruptedException {
    Process memoryTest =
Runtime.getRuntime().exec("performanceCounters.exe");
    memoryTest.waitFor();
}
```

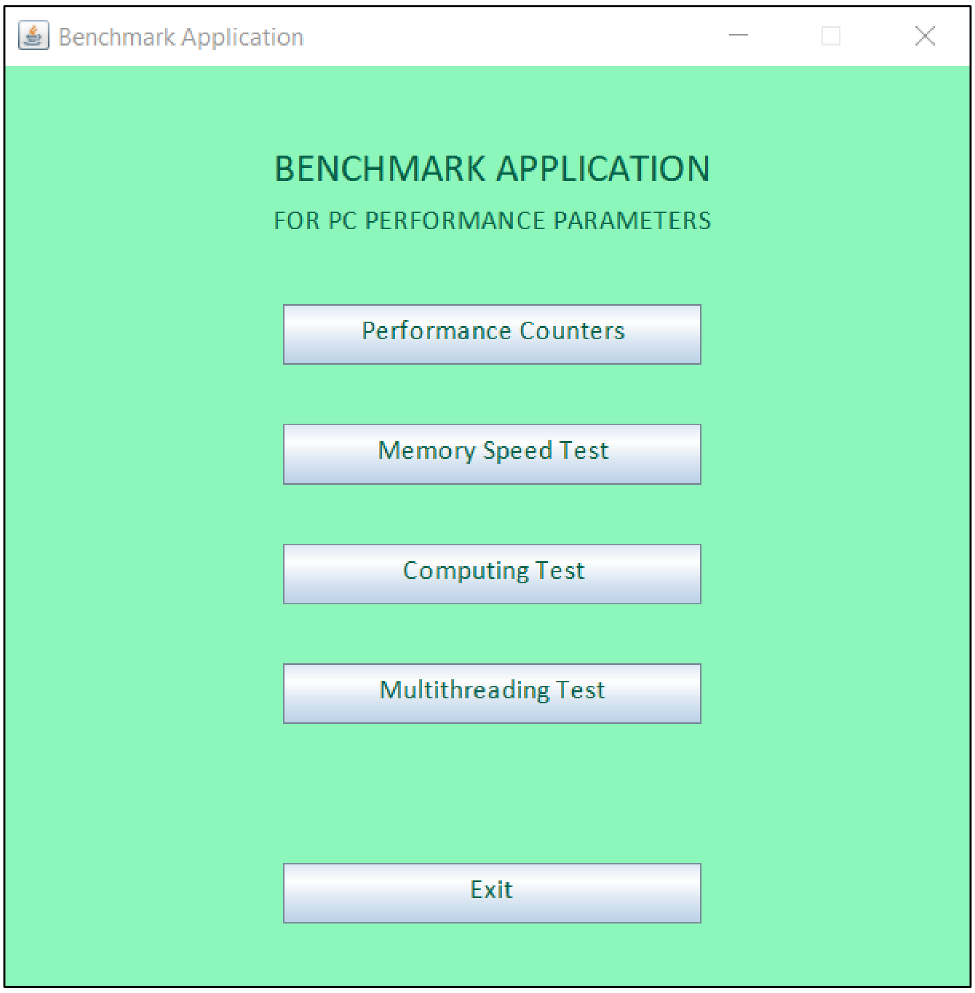
6. Testing and Results

- Mentions
    - all JUnit test pass
    - all plots from below have the number of repetitions on the x-axis and the execution time in miliseconds on the y-axis [8].

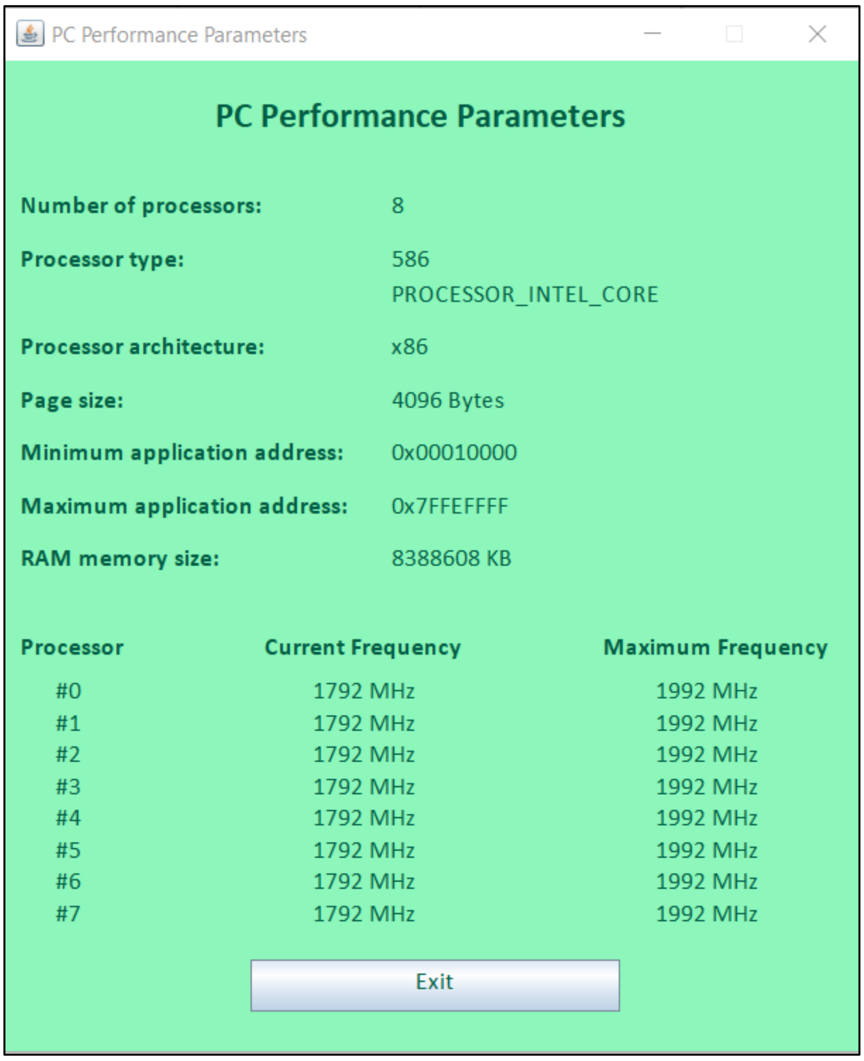


```
Tests                    3 m 35 s 697 ms
  BenchmarkTest          3 m 35 s 697 ms
    computingTest             36 s 986 ms
    multithreadingTest   2 m 40 s 111 ms
    memoryTest                18 s 600 ms
```

- Home Page



- Performance Counters – from the application



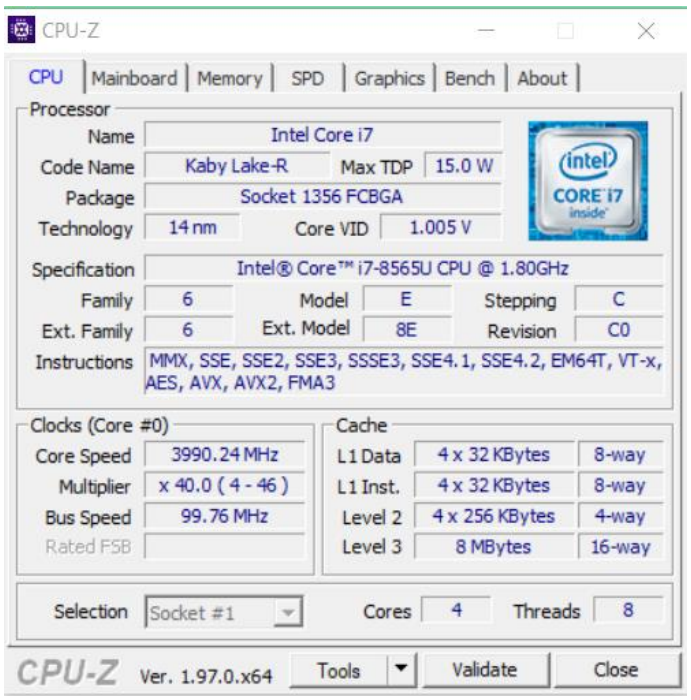
PC Performance Parameters

PC Performance Parameters

Number of processors:	8
Processor type:	586 PROCESSOR_INTEL_CORE
Processor architecture:	x86
Page size:	4096 Bytes
Minimum application address:	0x00010000
Maximum application address:	0x7FFEFFFF
RAM memory size:	8388608 KB

Processor	Current Frequency	Maximum Frequency
#0	1792 MHz	1992 MHz
#1	1792 MHz	1992 MHz
#2	1792 MHz	1992 MHz
#3	1792 MHz	1992 MHz
#4	1792 MHz	1992 MHz
#5	1792 MHz	1992 MHz
#6	1792 MHz	1992 MHz
#7	1792 MHz	1992 MHz

- Performance Counters – from CPUz



CPU-Z

Processor

Name	Intel Core i7		
Code Name	Kaby Lake-R	Max TDP	15.0 W
Package	Socket 1356 FCBGA		
Technology	14 nm	Core VID	1.005 V

Specification: Intel® Core™ i7-8565U CPU @ 1.80GHz

Family	6	Model	E	Stepping	C
Ext. Family	6	Ext. Model	8E	Revision	C0

Instructions: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3

Clocks (Core #0)

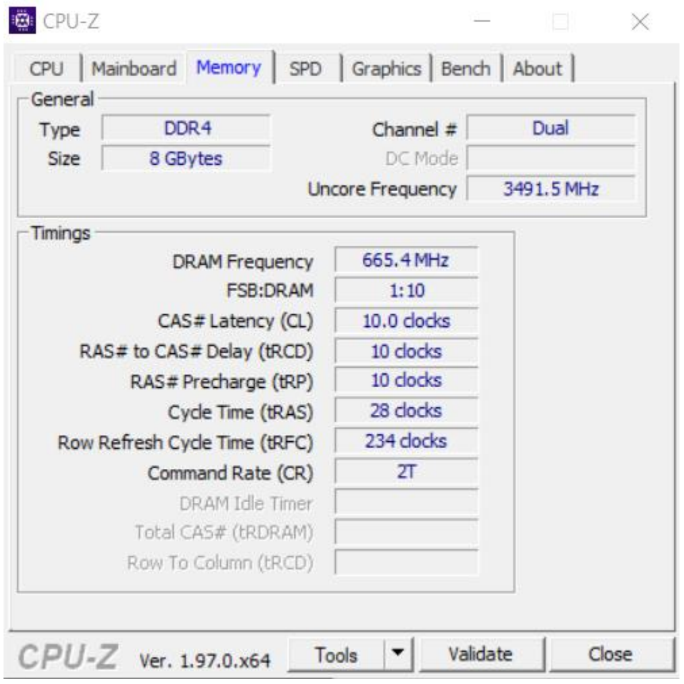
Core Speed	3990.24 MHz
Multiplier	x 40.0 (4 - 46)
Bus Speed	99.76 MHz
Rated FSB	

Cache

L1 Data	4 x 32 KBytes	8-way
L1 Inst.	4 x 32 KBytes	8-way
Level 2	4 x 256 KBytes	4-way
Level 3	8 MBytes	16-way

Selection: Socket #1 Cores: 4 Threads: 8

CPU-Z Ver. 1.97.0.x64



CPU-Z

General

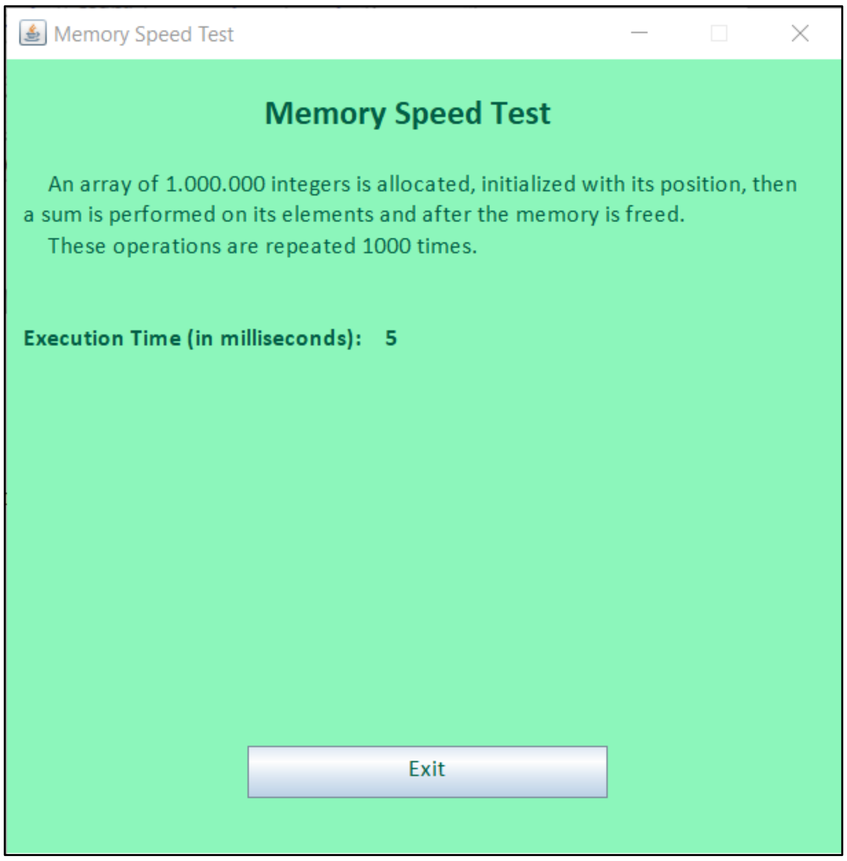
Type	DDR4	Channel #	Dual
Size	8 GBytes	DC Mode	
		Uncore Frequency	3491.5 MHz

Timings

DRAM Frequency	665.4 MHz
FSB:DRAM	1:10
CAS# Latency (CL)	10.0 clocks
RAS# to CAS# Delay (tRCD)	10 clocks
RAS# Precharge (tRP)	10 clocks
Cycle Time (tRAS)	28 clocks
Row Refresh Cycle Time (tRFC)	234 clocks
Command Rate (CR)	2T
DRAM Idle Timer	
Total CAS# (tRDRAM)	
Row To Column (tRCD)	

CPU-Z Ver. 1.97.0.x64

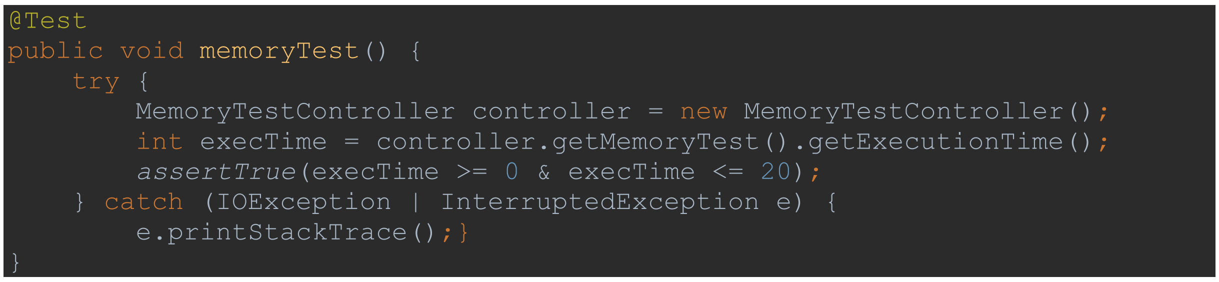
- Memory Speed Test



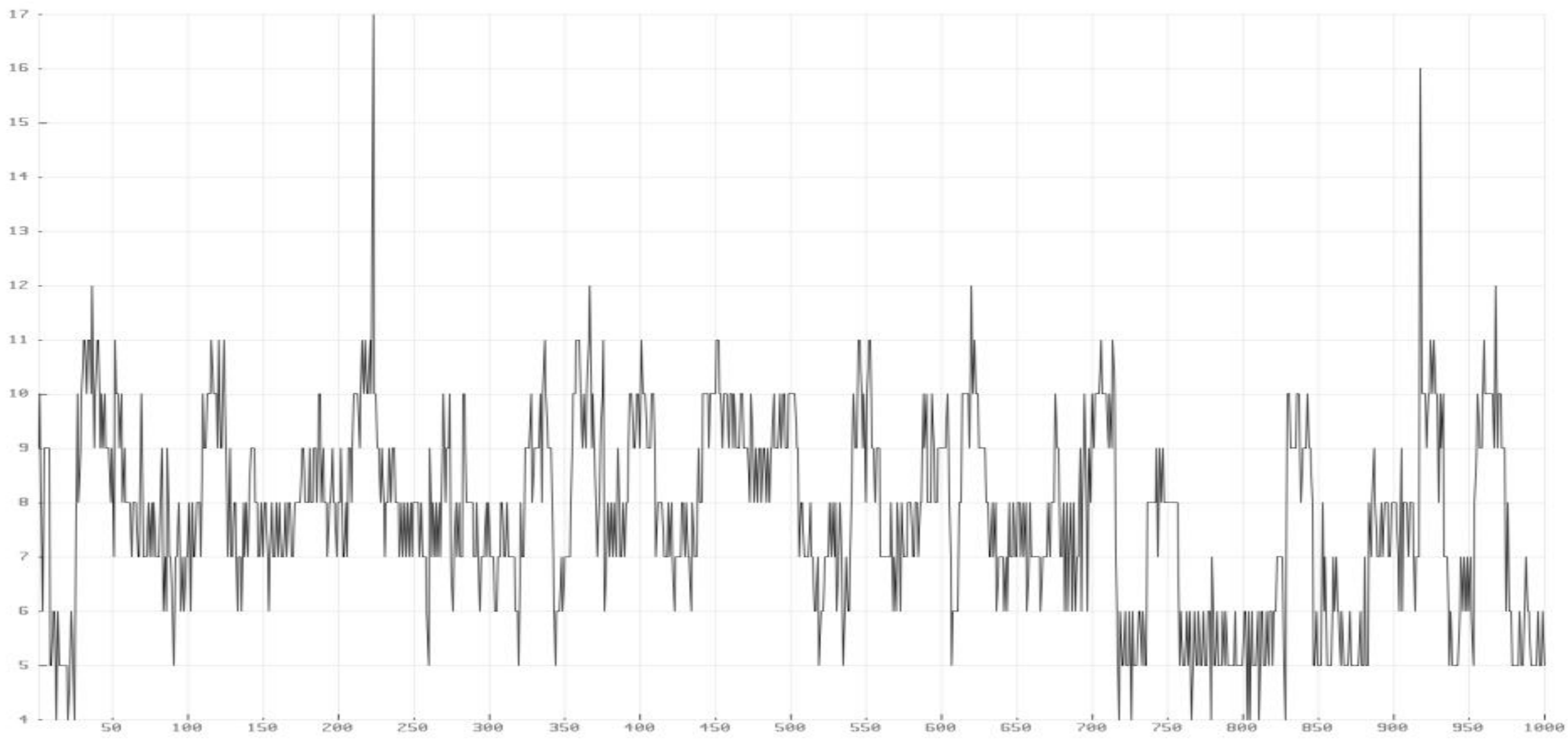
Memory Speed Test

An array of 1.000.000 integers is allocated, initialized with its position, then a sum is performed on its elements and after the memory is freed.
These operations are repeated 1000 times.

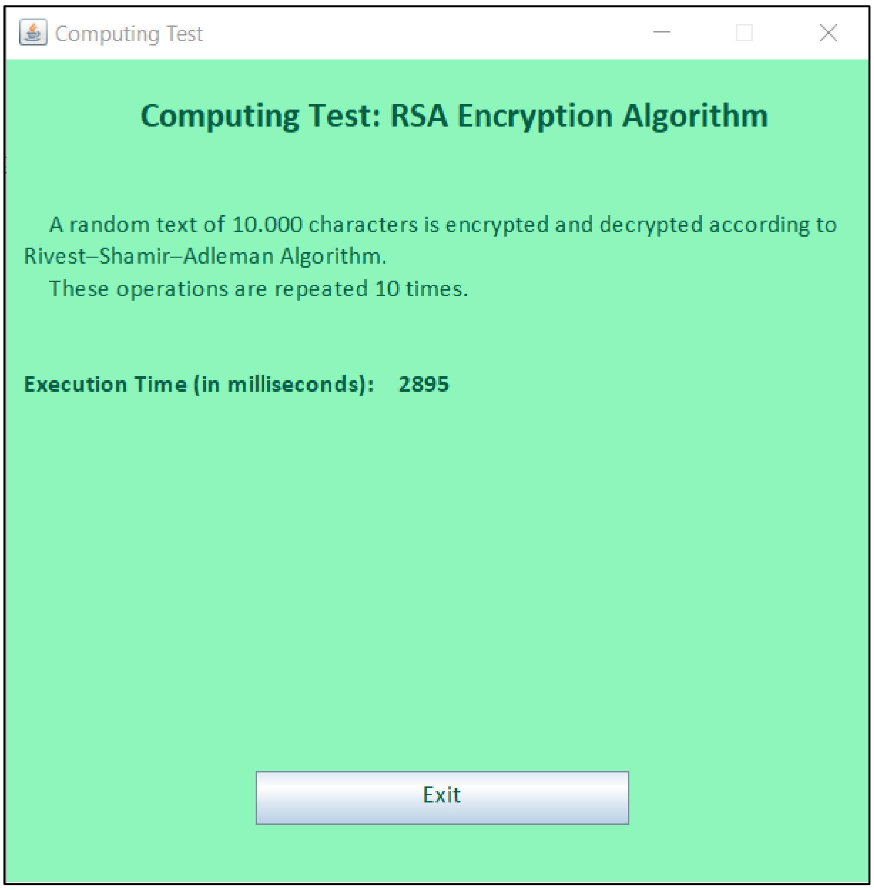
Execution Time (in milliseconds): 5



```
@Test
public void memoryTest() {
    try {
        MemoryTestController controller = new MemoryTestController();
        int execTime = controller.getMemoryTest().getExecutionTime();
        assertTrue(execTime >= 0 & execTime <= 20);
    } catch (IOException | InterruptedException e) {
        e.printStackTrace();}
}
```



- Computing Test

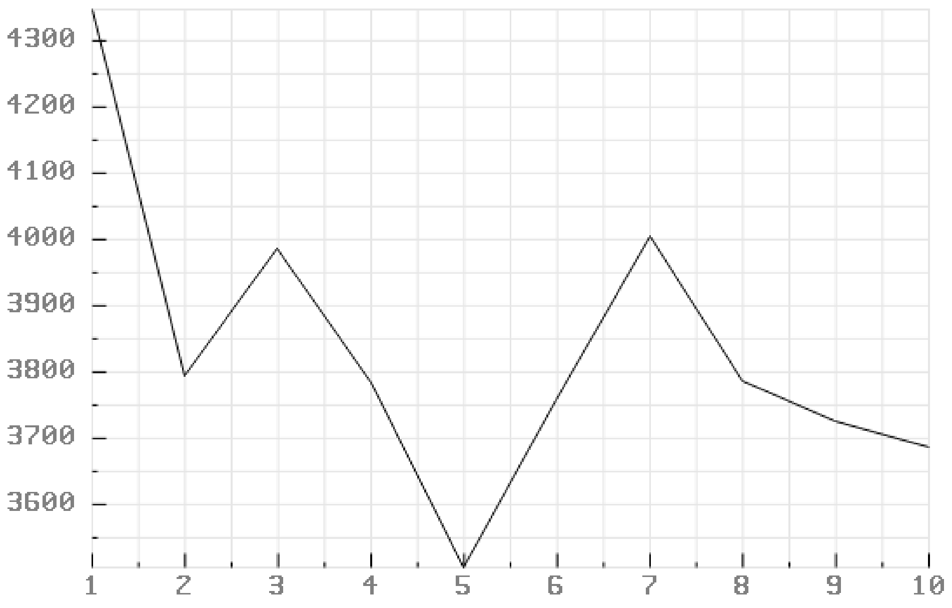


Computing Test: RSA Encryption Algorithm

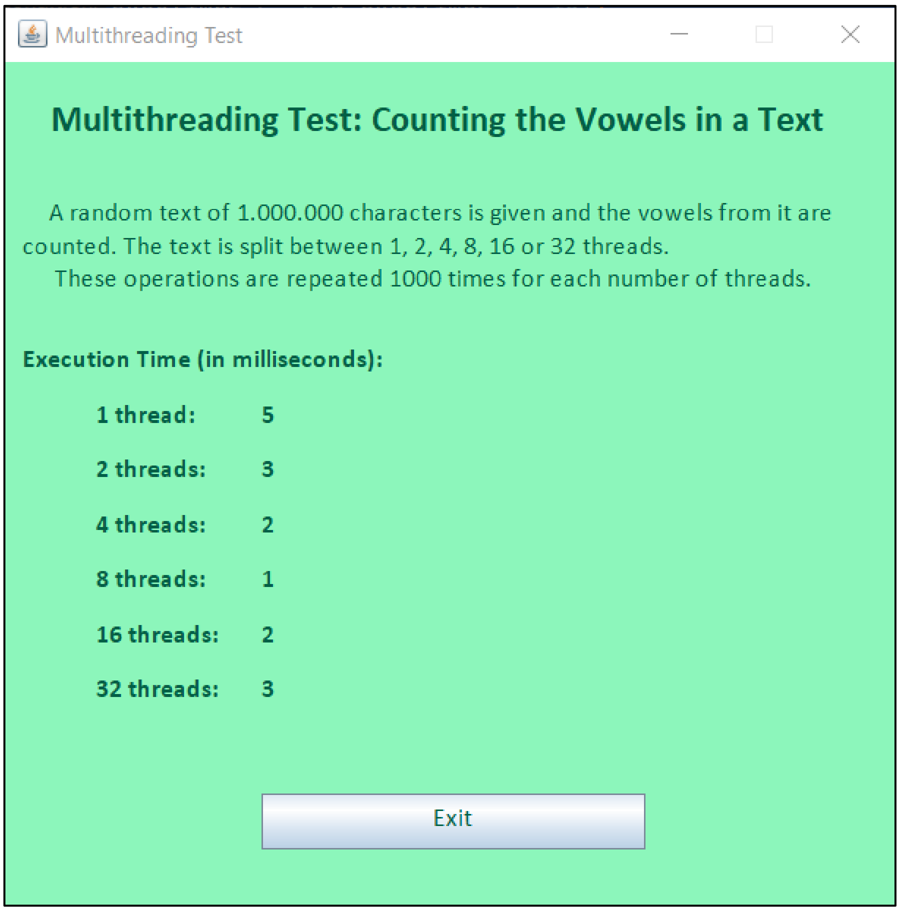
A random text of 10.000 characters is encrypted and decrypted according to Rivest–Shamir–Adleman Algorithm.
These operations are repeated 10 times.

Execution Time (in milliseconds): 2895

```
@Test
public void computingTest() {
    try {
        ComputingTestController controller = new ComputingTestController();
        int execTime = controller.getComputingTest().getExecutionTime();
        assertTrue(execTime >= 1000 & execTime < 10000);
    } catch (IOException | InterruptedException e) {
        e.printStackTrace();
    }
}
```



- Multithreading Test



Multithreading Test

Multithreading Test: Counting the Vowels in a Text

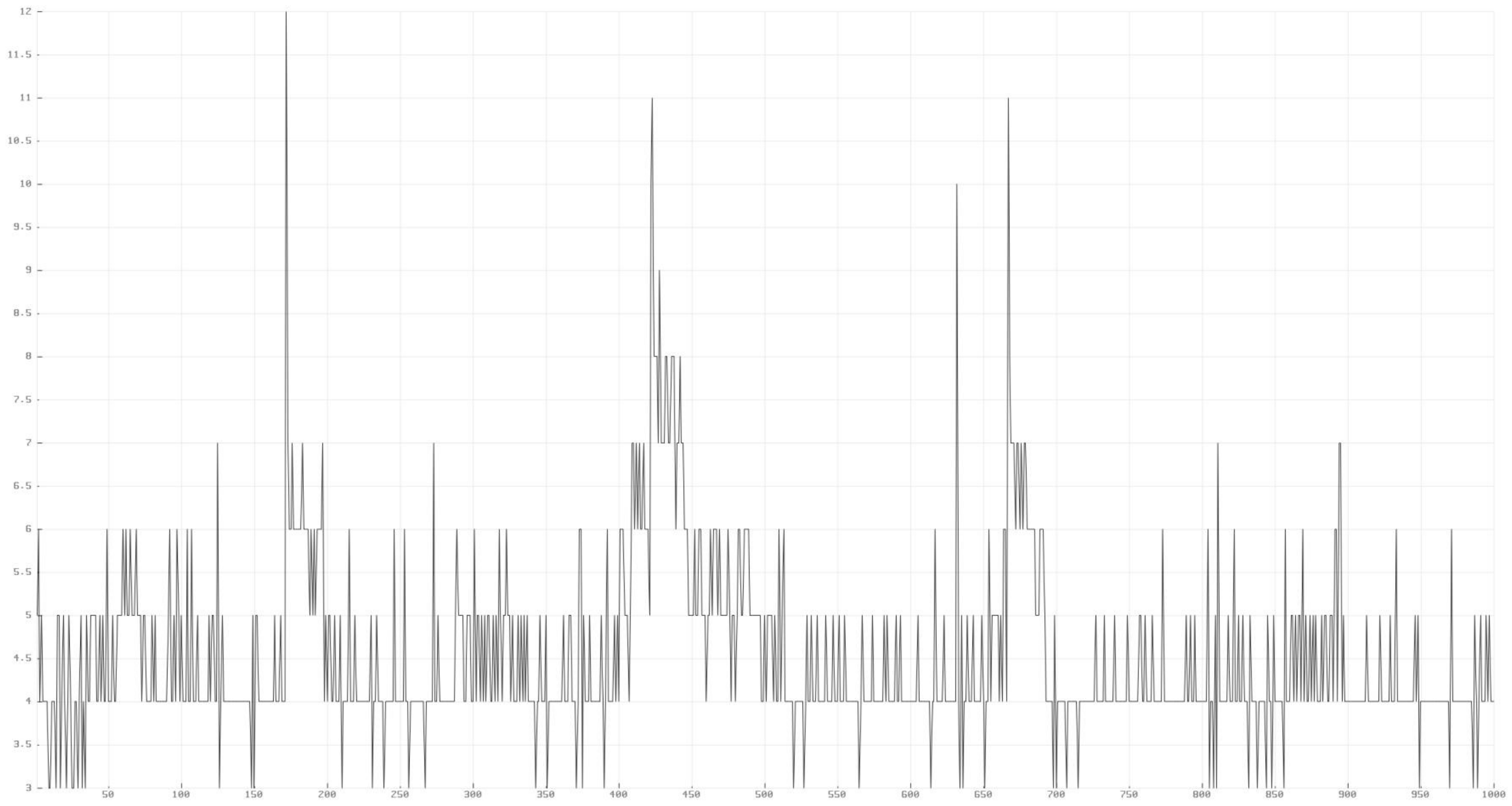
A random text of 1.000.000 characters is given and the vowels from it are counted. The text is split between 1, 2, 4, 8, 16 or 32 threads.
These operations are repeated 1000 times for each number of threads.

Execution Time (in milliseconds):

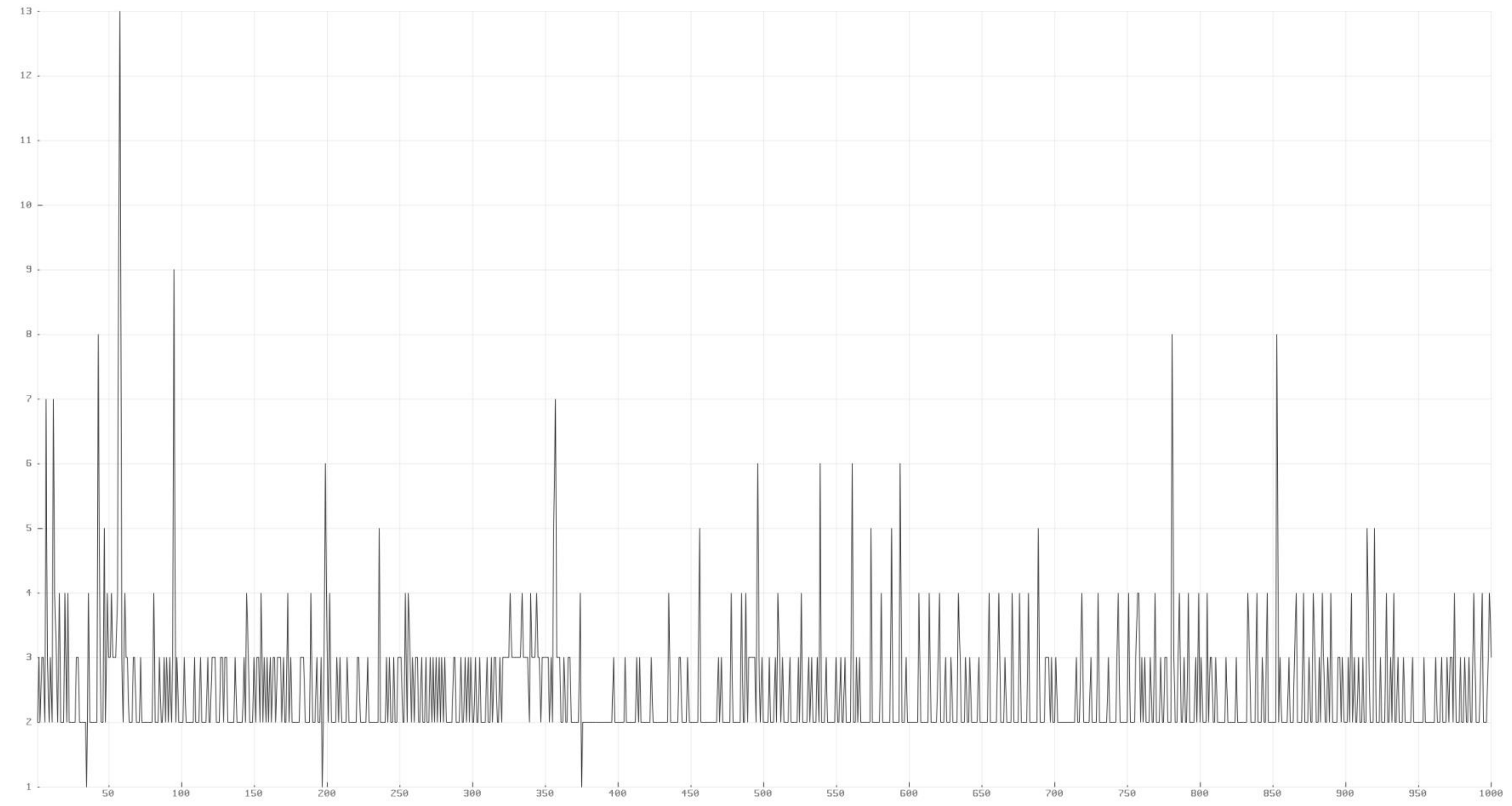
1 thread:	**5**
2 threads:	**3**
4 threads:	**2**
8 threads:	**1**
16 threads:	**2**
32 threads:	**3**

```
@Test
public void multithreadingTest() {
    try {
        MultithreadTestController controller = new
MultithreadTestController();
        int execTime1 =
controller.getMultithreadTest().getExecutionTime1();
        int execTime2 =
controller.getMultithreadTest().getExecutionTime2();
        int execTime4 =
controller.getMultithreadTest().getExecutionTime4();
        int execTime8 =
controller.getMultithreadTest().getExecutionTime8();
        int execTime16 =
controller.getMultithreadTest().getExecutionTime16();
        int execTime32 =
controller.getMultithreadTest().getExecutionTime32();
        assertTrue(execTime1 >= 0 & execTime1 <= 10);
        assertTrue(execTime2 >= 0 & execTime2 <= 10);
        assertTrue(execTime4 >= 0 & execTime4 <= 10);
        assertTrue(execTime8 >= 0 & execTime8 <= 10);
        assertTrue(execTime16 >= 0 & execTime16 <= 10);
        assertTrue(execTime32 >= 0 & execTime32 <= 10);
    } catch (IOException | InterruptedException e) {
        e.printStackTrace();
    }
}
```

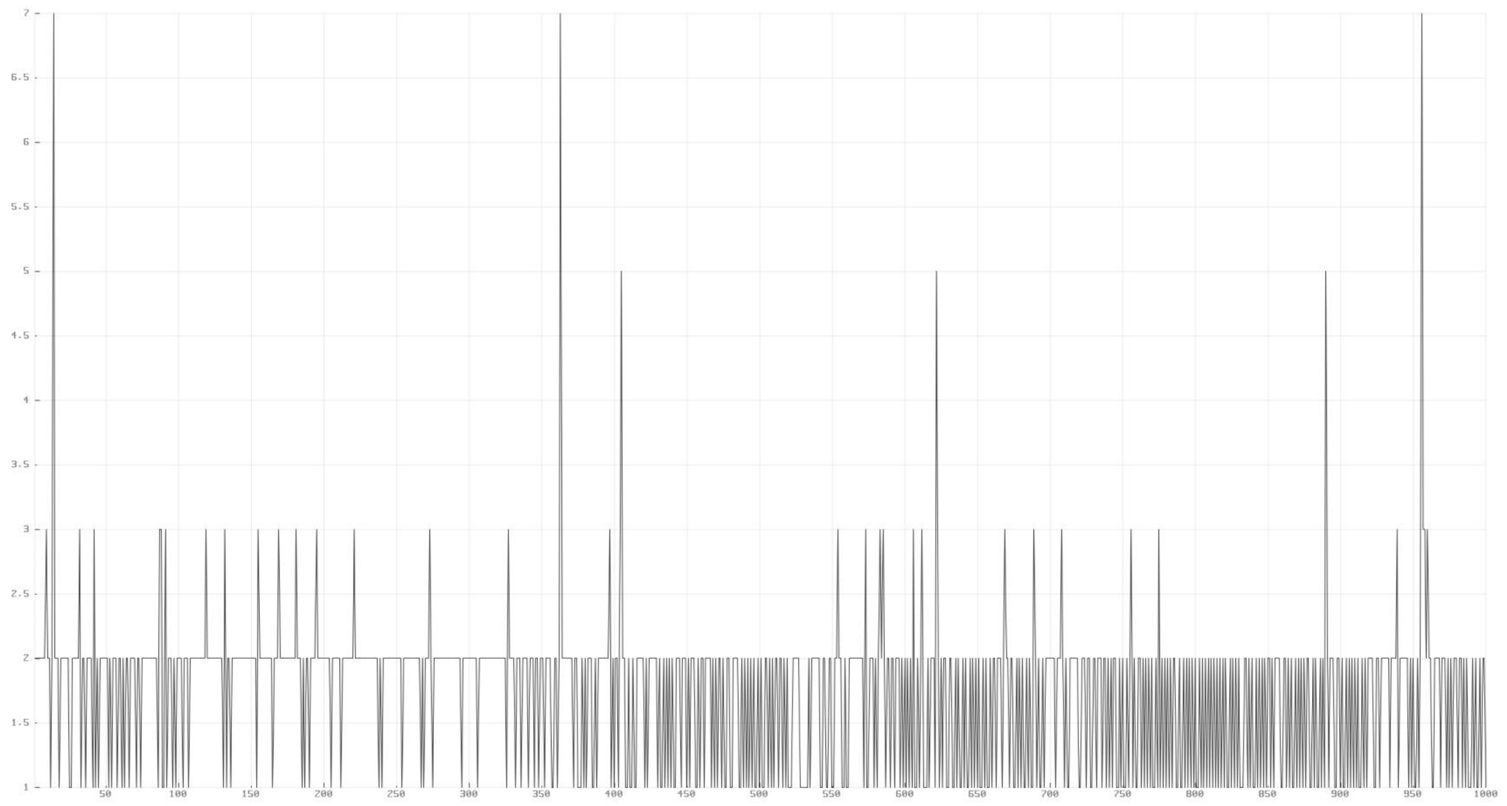
- 1 thread



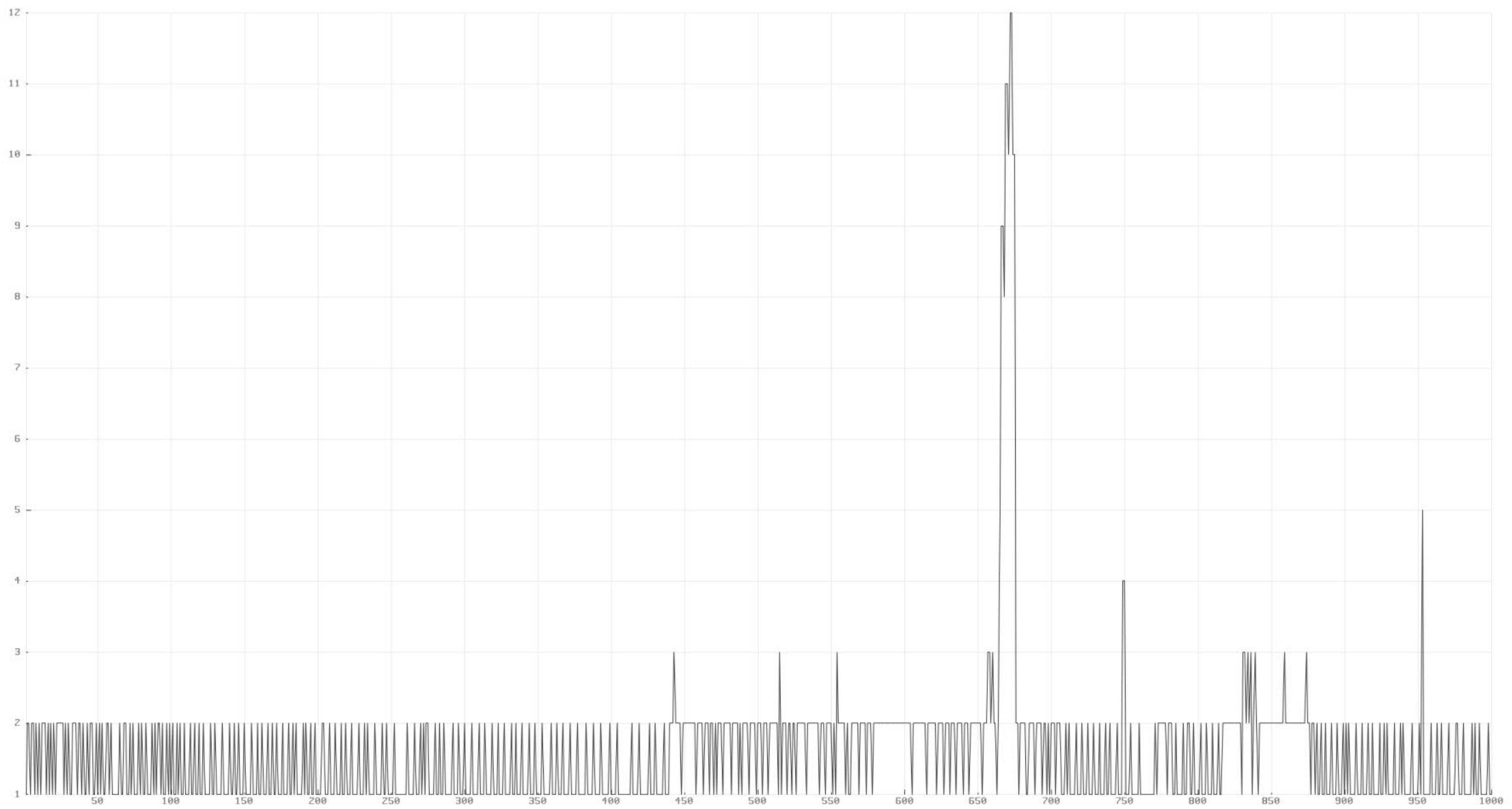
- 2 thread



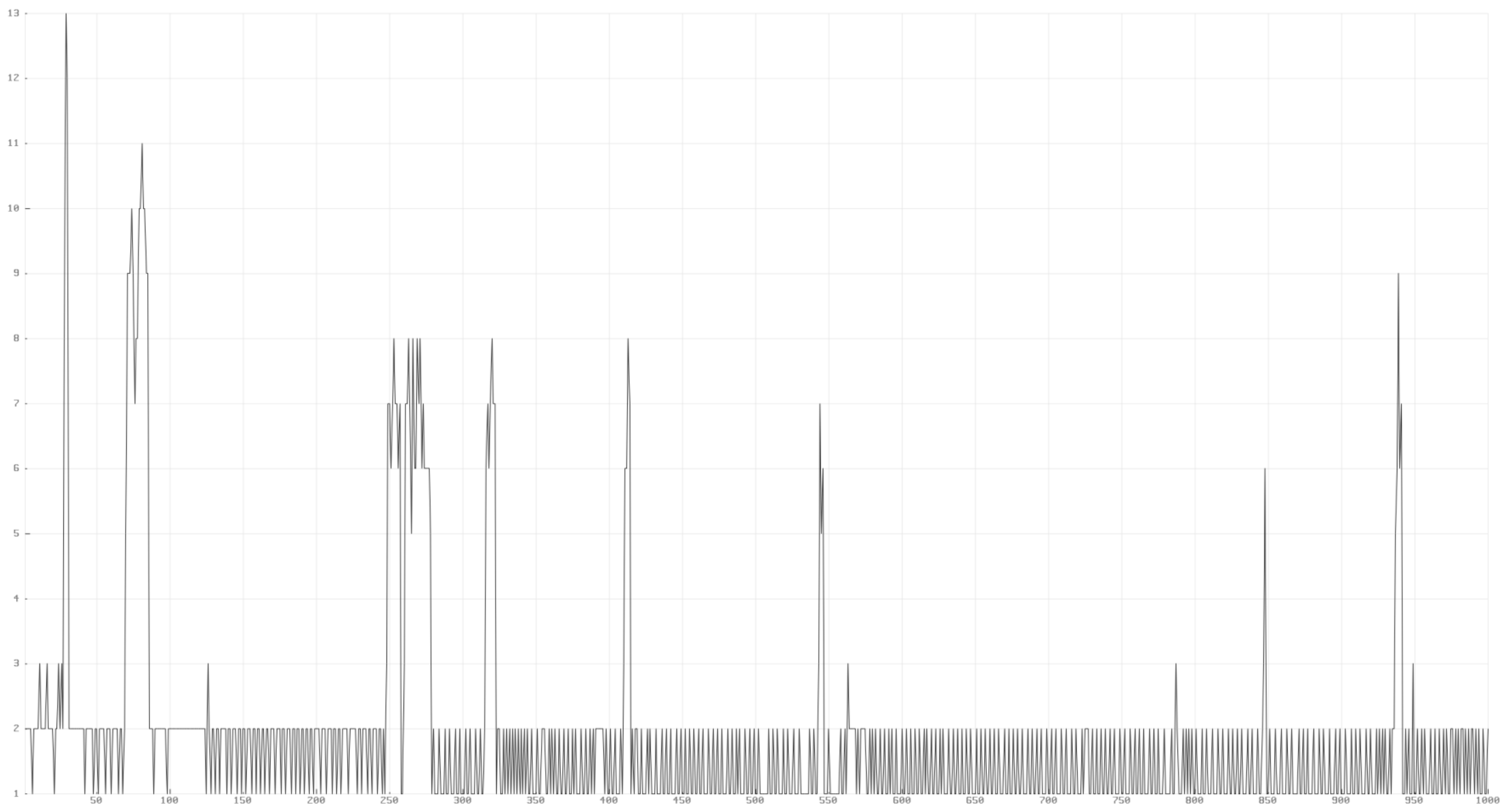
- 4 threads



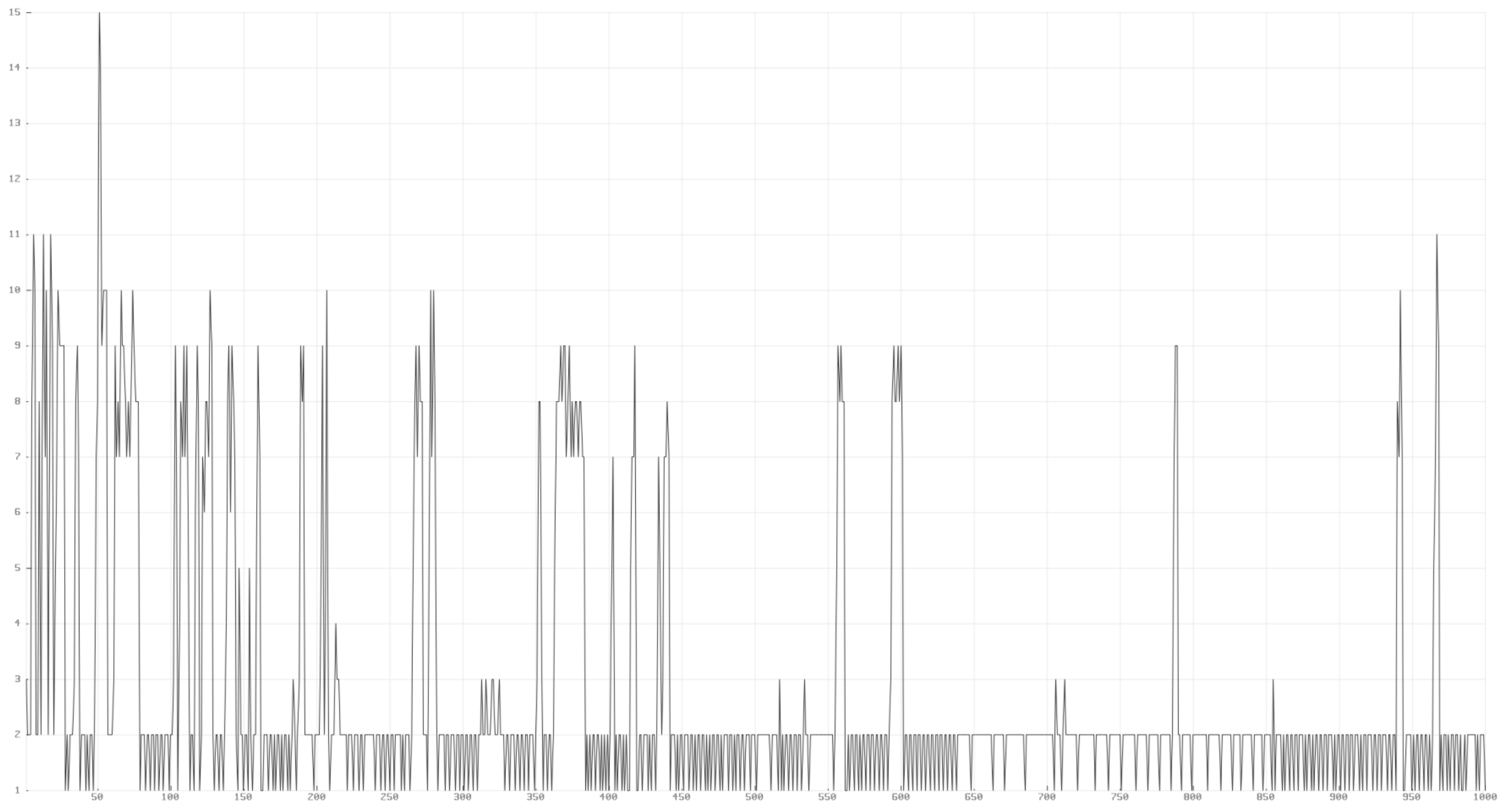
- 8 threads



- 16 threads



- 32 threads



7. Conclusions and Further Improvements

The goal of this project was to design and implement a basic benchmark program designed to run on modern personal computers. What I learned from this project was to read the PC performance parameters using C++ code and to make the connection between a Java application and several C++ files, mainly to run processes from the Java application.

Some further improvements of the Benchmark Application could be:

- reading the parameters of the GPU
- writing more tests for testing the memory speed, the computing speed and the multithreading

8. **Bibliography**

[1] https://docs.microsoft.com/en-us/windows/win32/sysinfo/getting-hardware-information
[2] https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-getsysteminfo
[3] https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/ns-sysinfoapi-system_info
[4] https://docs.microsoft.com/ro-ro/windows/win32/api/sysinfoapi/nf-sysinfoapi-getphysicallyinstalledsystemmemory?redirectedfrom=MSDN
[5] https://docs.microsoft.com/en-us/windows/win32/power/processor-power-information-str
[6] https://docs.microsoft.com/en-us/windows/win32/api/powerbase/nf-powerbase-callntpowerinformation
[7] http://www.trytoprogram.com/cpp-examples/cplusplus-program-encrypt-decrypt-string/#rsa
[8] https://github.com/InductiveComputerScience/pbPlots/tree/v0.1.7.2/Cpp