Technical University of Cluj-Napoca

Structure of Computer Systems

Laboratory Project

Set of benchmark programs to evaluate the facilities of a multicore processor

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Laboratory Assistant:

Neagu Mădălin

Student:

Enoiu Diana-Cristina

Contents

1 Objective

The objective of this project is to develop a set of benchmark programs in order to evaluate the performance of a multicore processor. In order to fully utilize all the physical and logical cores of a processors, multi-threading should be used in the development of the application.
This software program can be used by people who want to measure the performance of their multicore processors and it can also be a great tool analyze the processing time in case the program is run on all the cores compared to when is run on only one core of the processor.

2 Bibliographic study

2.1 Benchmarks

To understand the functionalities of this application we must first know what a benchmark is:

- According to Wikipedia, *a benchmark is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it*.
- In the "Structure of computer systems" by Baruch is noted that, *one of the most utilized measures of computer performance is the time of executing a set of representative programs on that computer. This time can be the total execution time of the programs from the set, the arithmetic mean or the geometric mean of the execution times, or other similar measure. A set of actual programs that are representative for a particular computing environment can be used for performance evaluation. Such programs are called benchmarks and are run by the user on the computer being evaluated.*

We should be aware that measuring and assessing the performance of a system is not a simple task because some computers or CPUs perform better for some tests and worse for others. Performance should be a weighted average of a number of specific tests. For example, let us consider a situation where is difficult to measure the performance of two machines.

	Computer A	**Computer B**
Program 1 (s)	1	10
Program 2 (s)	1000	100
Total time (s)	1001	110

For program '1', the computer A is 10 times faster than computer B, but for the program '2' the computer B is also 10 times faster then computer A. Therefore, the relative performance of both of this computers is unclear.

A simple way to summarize the relative performance is to by using total execution time of both of the programs. Now, we can conclude that computer B is 1001/110 which is 9.1 faster that the computer A for programs '1' and '2'. There are also other ways for comparing the execution time of different computers, as arithmetic mean, weighted arithmetic mean, geometrical mean or normalized geometrical mean.

2.2 Multi-core processor

It is also important to note what is a multicore processor and what is multithreading and hyper-threading.

- According to Wikipedia, *a multi-core processor is a computer processor on a single integrated circuit with two or more separate processing units, called cores, each of which reads and executes program instructions*, I would add, in parallel.

The concept of a multicore processor should not be unfamiliar to us since nowadays multicore processors are widely use. Our PC, mobile phone or laptop most likely has a multicore processor. In order for us to utilize all the cores of a processor in parallel, we need something that is called multithreading.

- The official Intel webpage notes that, *multithreading is a form of parallelization or dividing up work for simultaneous processing. Instead of giving a large workload to a single core, threaded programs split the work into multiple software threads. These threads are processed in parallel by different CPU cores to save time.*

Usually, a single thread is run on a single core, but this is not really the case for a hyper-threaded CPU .

- The official Intel webpage also notes that*, hyper-threading is a hardware innovation that allows more than one thread to run on each core. Physical cores are the number of physical cores, actual hardware components. Logical cores are the number of physical cores times the number of threads that can run on each core through the use of hyper-threading.*

Hyper-threading allows for two threads to run on a single core. This means that if a processor has 8 cores, if it is hyper-threaded, the operating system will actually see 16 logical cores on the processor.

3 Analysis

There are different types of benchmark programs:

- Real programs
  - word processing software
  - user's application software

- Micro-benchmarks
  - Designed to measure the performance of a very small and specific piece of code.

- Kernel
  - contains codes that perform a specific basic operation
  - normally abstracted from actual program
  - popular kernel: Livermore loops (every loop is a mathematical operation)
  - Linpack benchmark (contains basic linear algebra subroutines)
  - results are represented in MFLOPS

- Component Benchmarks/ micro-benchmarks
    - programs designed to measure performance of a computer's basic components
    - automatic detection of computer's hardware parameters like number of registers, cache size, memory latency

- Synthetic Benchmarks
    - Procedure for programming synthetic benchmark:
        - take statistics of all types of operations from many application programs
        - get proportion of each operation
        - write program based on the proportion above
    - Types of Synthetic Benchmark are:
        - Dhrystone – integer arithmetic
        - Whetstone – integer and floating point arithmetic

- Other benchmarks
    - I/O benchmarks
    - Database benchmarks: to measure the throughput and response times of database management systems (DBMS')
    - Parallel benchmarks: used on machines with multiple cores, processors or systems consisting of multiple machines

The set of programs implemented in this application will be a Component Benchmarks programs, since the target is to measure the performance of a CPU. They will also be parallel benchmarks as the type of processor that we want to measure is a multi-core processor and we will make use of the multithreading to analyze the performance.

The set of benchmark programs will mainly consist of some high intensive tasks that will require a lot processing power. Those tasks will be: finding the prime numbers among 30 000 000 numbers, calculating the pi number using Gregory-Leibniz Series with an N as big as possible and sort an array of size 20 000 000 using merge sort. These three tasks - prime number finder, pi calculation and merge sort - will be implemented using multithreading in order to see how multi-core processors will handle this tasks more effectively since they can run threads in parallel.

- In opposition to a multi-core processor, a single-core processor will not be able to run multiple threads in parallel, because it can only run instruction one by one i.e. concurrently. This means some time will be wasted with thread synchronization. The advantage of the multi-core processors is that they can run those multiple threads in parallel, reducing the time taken to finish a task.

When running an intensive task using on only one thread we are not fully utilizing the hole multi-core processor. In fact we use up to one core for that task, as if we had a single core processor. The optimization of utilizing the hole processor to handle that intensive task in order to reduce the time taken for that task is up to the programmer, by making his task multithreaded.

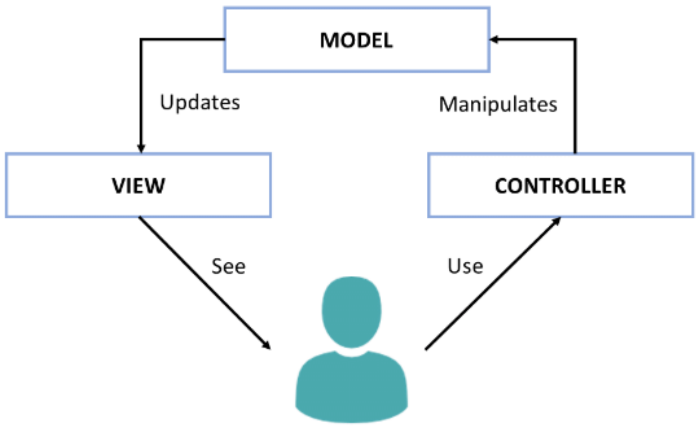
4 Design

4.1 Architecture

The application has a graphical user interface made in JavaFX, with the help of Scene Builder.

For the design of this application it was used, also, an architectural pattern, the MVC pattern, (Model-View-Controller pattern), which is used to separate application's concerns.

The Model–View–Controller is a software design pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements.



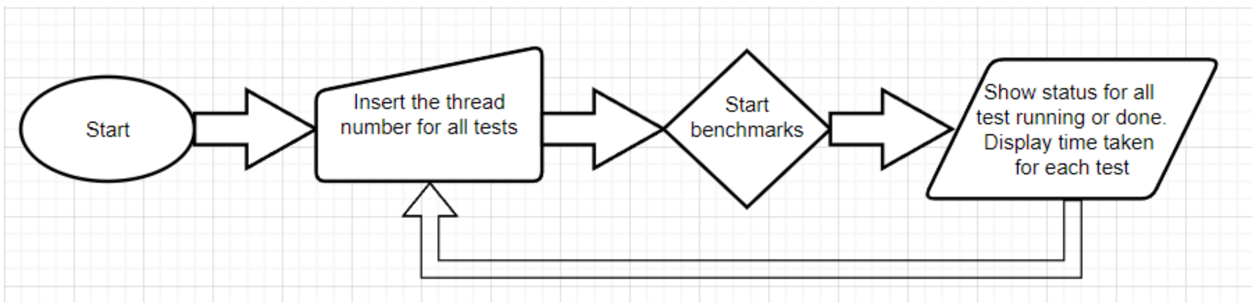
The model has the main tests that will be run for benchmarking.

The view contains .fxml files used for displaying the actual interface;

The controller has a class ViewFxmlController which processes the input form the user and run the necessary tests with the given parameters.

4.2 Use cases and flow chart

We will be able to insert the thread number that we want to use for each test and then start the benchmarks. The application will tell use the CPU utilization at every moment even when tests are not running. It will also tell us which tests are running or done, and the time taken for each test.


Start
Insert the thread number for all tests
Start benchmarks
Show status for all test running or done. Display time taken for each test

5 Implementation

5.1 Measuring CPU utilization

To measure the utilization of the CPU I have used the package OperatingSystemMXBean. This measurements is handled in the MeasureCpuStats class and it will run on another thread to get the cpu utilization in real time.

- The package to import

```
import com.sun.management.OperatingSystemMXBean;
```

- Instantiation

```
OperatingSystemMXBean osBean =
ManagementFactory.getPlatformMXBean(OperatingSystemMXBean.class);
```

- While running the thread update GUI with CPU utilization percent

```
String cpuUsage = (Math.floor(osBean.getCpuLoad() * 10000)/100) + "%";
System.out.println(cpuUsage);
Platform.runLater(() -> updateGUI(cpuUsage));
try {
    Thread.sleep(500);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

5.2 Prime Finder Test

For this test we will try to find all the prime numbers between 0 and 30 000 000. To make this process multithreaded we will assign each thread a number to check if it is prime by making a modulo between the number and the number of threads.

- Splitting work among threads

```
for(int i=0; i < StartTests.MAX; i++) {
    if(i % threadNumber == ID)
        if(isPrime(i))
            m.addPrime(i);
}
```

- Check if prime

```
public static boolean isPrime(int n) {
    if (n == 2 || n == 3 || n == 5) return true;
    if (n <= 1 || (n&1) == 0) return false;

    for (int i = 3; i*i <= n; i += 2)
        if (n % i == 0) return false;

    return true;
}
```

- Add the found prime numbers to an array

```
static class Monitor {
    public synchronized void addPrime(int n) {
         StartTests.addPrime(n);}}
```

5.3 Pi Calculation Test

Maybe there are more efficient ways to calculate the pi number but we have to use something more intensive for the processor to analyze the performance. We will use the Gregory-Leibniz Series to calculate pi/4 and then we can multiply by 4.

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots = \frac{\pi}{4}.$$

- The splitting among threads is similar to prime finder:

```
for (long i = 0; i <= n; i++) {
    if (i % threadNumber == threadID) {
        sum += Math.pow(-1, i) / (2 * i + 1);
    }
}
```

- We also have to give a big ‘n’ number to get a more precise result.

```
int n = 300000000;
```

5.4 Merge Sort Test

Merge Sort is kind of intended to be parallelized since if we want to have a parallel sorting we have to merge the results at the end.

Each thread should take a equal part of the array to sort if possible. If array size does not split exactly by the number of threads, then we split by no. of threads – 1 and the last thread takes the remaining.

- Splitting among threads

```
final int length = list.length;
boolean exact = length%threads_no == 0;
int maxlim = exact? length/threads_no: length/(threads_no-1);
maxlim = Math.max(maxlim, threads_no);
final ArrayList<SortThreads> threads = new ArrayList<>();
for(int i=0; i < length; i+=maxlim){
    int beg = i;
    int remain = (length)-i;
    int end = remain < maxlim? i+(remain-1): i+(maxlim-1);
    final SortThreads t = new SortThreads(list, beg, end);
    // Add the thread references to join them later
    threads.add(t);
}
```

- Merge everything at the end

```
for(int i=0; i < length; i+=maxlim){
    int mid = i == 0? 0 : i-1;
    int remain = (length)-i;
    int end = remain < maxlim? i+(remain-1): i+(maxlim-1);
    merge(list, 0, mid, end);
}
```

- The 2-way merge

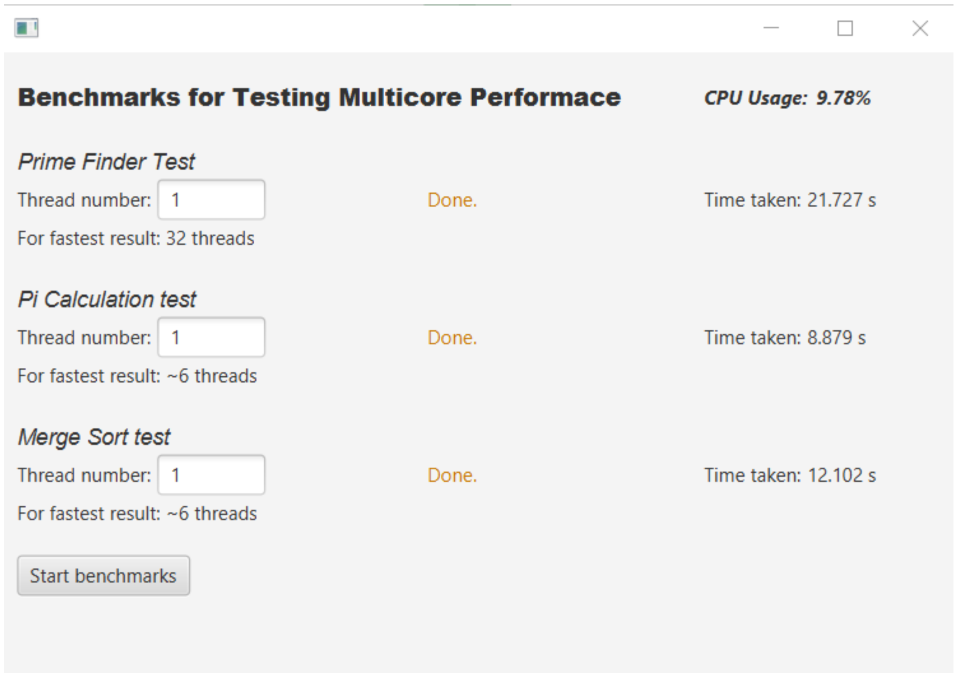
```
//Typical 2-way merge
public static void merge(Integer[] array, int begin, int mid, int end){
    Integer[] temp = new Integer[(end-begin)+1];

    int i = begin, j = mid+1;
    int k = 0;

    // Add elements from first half or second half based on whichever is
lower,
    // do until one of the list is exhausted and no more direct one-to-one
comparison could be made
    while(i<=mid && j<=end){
        if (array[i] <= array[j]){
            temp[k] = array[i];
            i+=1;
        }else{
            temp[k] = array[j];
            j+=1;
        }
        k+=1;
    }

    // Add remaining elements to temp array from first half that are left
over
    while(i<=mid){
        temp[k] = array[i];
        i+=1; k+=1;
    }

    // Add remaining elements to temp array from second half that are left
over
    while(j<=end){
        temp[k] = array[j];
        j+=1; k+=1;
    }

    for(i=begin, k=0; i<=end; i++,k++){
        array[i] = temp[k];
    }
}
```

Call merge sort for each chunk of the array processed by a thread, since we already have the ‘merge’ function we can use it

```
public static void mergeSort(Integer[] array, int begin, int end){
    if (begin<end){
        int mid = (begin+end)/2;
        mergeSort(array, begin, mid);
        mergeSort(array, mid+1, end);
        merge(array, begin, mid, end);
    }
}
```

6 Results

The time taken to run with only one thread is not very small. If only using one thread the CPU utilization barely reaches 12% which means we use mainly one core to perform the test. This test shoes the performance we will have with a single core processor.



Benchmarks for Testing Multicore Performace *CPU Usage: 9.78%*

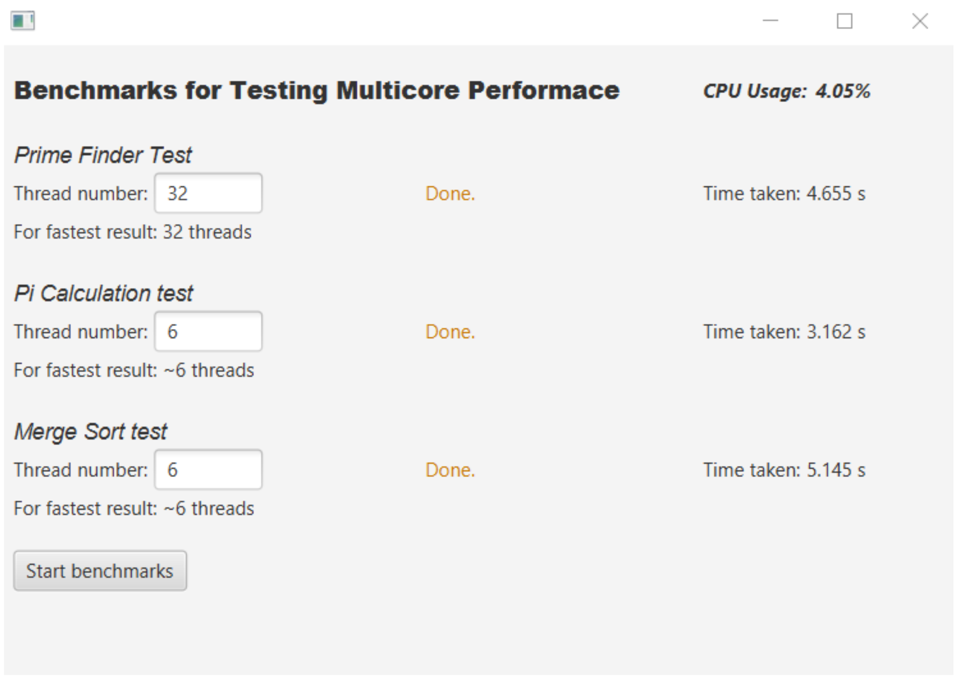
Prime Finder Test
Thread number: 1 Done. Time taken: 21.727 s
For fastest result: 32 threads

Pi Calculation test
Thread number: 1 Done. Time taken: 8.879 s
For fastest result: ~6 threads

Merge Sort test
Thread number: 1 Done. Time taken: 12.102 s
For fastest result: ~6 threads

Start benchmarks

Giving more threads to the program means using more of the multicore CPU thus obtaining faster results. In some moments we can also reach 100% utilization of the CPU which means we take advantage of the hole CPU.



Benchmarks for Testing Multicore Performace *CPU Usage: 4.05%*

Prime Finder Test
Thread number: 32 Done. Time taken: 4.655 s
For fastest result: 32 threads

Pi Calculation test
Thread number: 6 Done. Time taken: 3.162 s
For fastest result: ~6 threads

Merge Sort test
Thread number: 6 Done. Time taken: 5.145 s
For fastest result: ~6 threads

Start benchmarks

7 Conclusions

This assignment helped me understand better how a multi-core processor works, what are the benefits of using a multi-core processor, explaining why they are so used nowadays. It also showed the importance of task parallelization when it comes to performance.

Creating a benchmark program was an interesting idea that made me discover the computational power of a CPU and also made me more aware to the necessity to make programs that exploit the potential of a multi-core processor.

8 Bibliography

https://en.wikipedia.org/wiki/Multi-core_processor
https://en.wikipedia.org/wiki/Benchmark_(computing)
https://users.utcluj.ro/~baruch/book_ssce/SSCE-Benchmark.pdf
https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html