Катедра: „Информатика и софтуерни науки”
Дисциплина: „Паралелна обработка на информацията”

К У Р С О В П Р О Е К Т

Тема:

Създаване на многонишкова програма за сумирането на дължимите суми от фактури, намиращи се в голям файл, представляващ единствен ресурс

Изследване на време, скорост, ускорение и ефективност при различен брой нишки посредством паралелно изпълнение

Десислава Емилова Милушева
ИСН, курс III, гр. 77,
Фак. № 471219007

Разработил: ……

/ Десислава Милушева /

Проверил: ……

/ доц. д-р инж. Десислава Иванова /
/ ас. Емануил Дончев /

София, 2022 г.

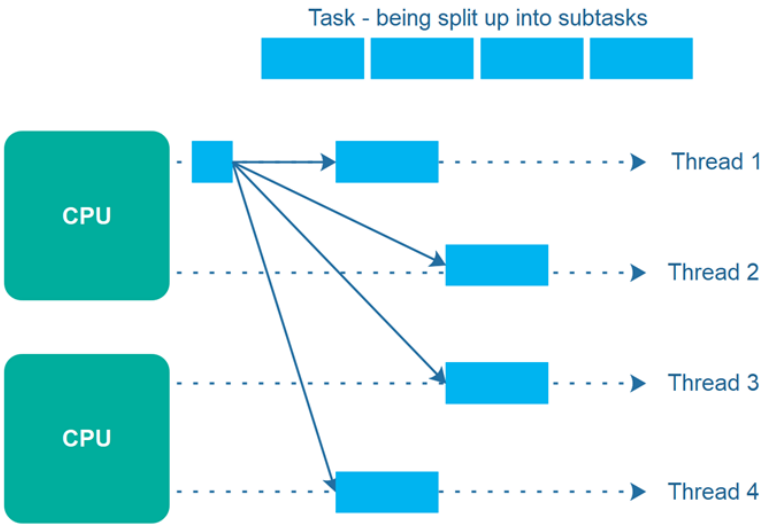
Съдържание

Част 1 - Въведение и приложение

1. Въведение

Изпълнението на множество задачи едновременно (т.н. multithreading) е способността на всяка операционна система да има повече от една програма, работеща в един и същи момент от времето. Например, можем да принтираме, докато редактираме или изтегляме имейла си, или да слушаме музика от музикалният плеър, докато разглеждаме снимки от галерията. В днешно време всеки компютър има повече от един процесор, но броят на едновременно изпълняваните процеси не е ограничен от броя на процесорите. Операционната система заделя отрязъци от време за всеки процес, създавайки впечатление за паралелна обработка. Многонишковите програми разширяват идеята за изпълнение на много процеси едновременно, като я спускат с едно ниво по-ниско: отделните програми ще изглежда да изпълняват множество задачи едновременно.

Езиците за програмиране поддържат различни подходи свързани със паралелното програмиране, включително програмиране с нишки, стартиране на подпроцеси и други различни трикове. В този курсов проект ще бъдат редставени и разгледани някои концепции, свързани с различни аспекти на едновременното програмиране, включително техники за програмиране с обща нишка и подходи свързани с това как едно приложение може да паралелизира изпълнението на една задача - обикновено чрез разделяне на задчата на подзадчи, които могат да бъдат изпълнени паралелно.



2. Приложение

Обхватът на приложение на multithreading-а е изключително голям и почти всяко приложение или програма използва повече от една нишка, по време на своето изпълнение, като така се оптимизира и самото действие, бързина и работа. Например, нека разгледаме случай на приложение, при който имаме компютър с едноядрен процесор. Това означава, че задачите, които трябва да бъдат изпълнени като част от приложение, не могат да постигнат напредък по едно и също време, тъй като процесорът е в състояние да работи само върху една задача в даден момент. Изпълнението на множество задачи едновременно означава, че процесорът извършва превключване на контекста, така че няколко задачи да могат да се изпълняват едновременно.

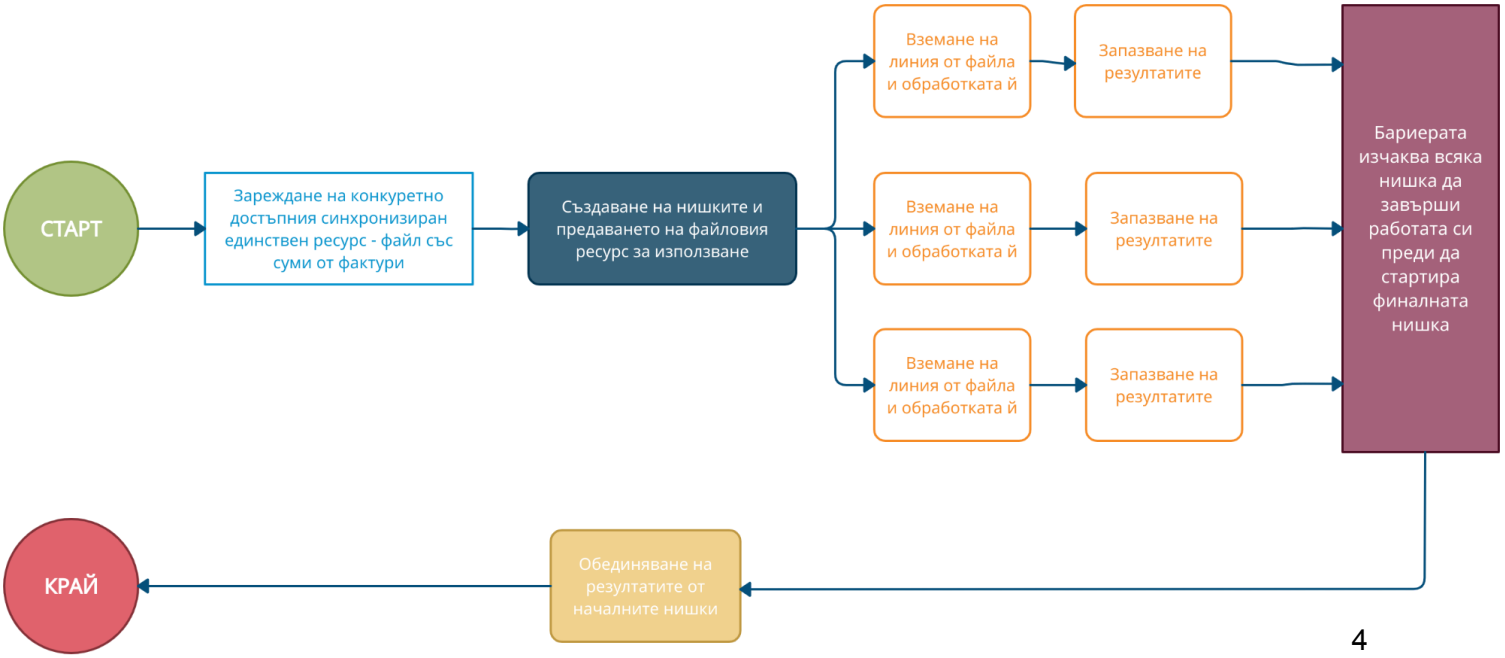
Част 2 - Описание на заданието

1. Описание на заданието

Да се създаде многонишкова програма за сумирането на дължимите суми от фактури, намиращи се в голям файл, представляващ единствен ресурс. Резултатите от извършването на сумирането от всяка отделна нишка трябва да бъдат сумирани и изведени, след като всяка от нишките е завършила изпълнението си.

2. Модел на паралелните изчисления

Визуално, процесът на програмата може да бъде репрезентиран по следния начин:



След стартиране на програмата и основната (main) нишка, първо се зарежда конкурентно достъпният файл, който е и единствен ресурс, а именно файлът със суми от фактури. Веднага след това се създават и останалите нишки (FileLineProcessingThreads), които ще работят върху файловия ресурс. Всяка нишка взема линия от файла и я обработва, като след това запазва резултата. Създадена бариера (CyclicBarrier) изчаква всяка от нишките (FileLineProcessingThreads) да завърши работата си. Накрая се стартира финална нишка (ResultFinalizationThread), която да обедини резултатите получени от всички нишки и да изведе реалният сбор от всички суми по фактури.

Част 3 - Имплементация на Java

1. Код на Java

https://github.com/desi109/invoices-sum-calculator---multithreading-java-app

InvoicesSumCalculatorMultithreading class

```
package com.invoices.sum.calculator;

import java.io.File;
import java.io.FileNotFoundException;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CyclicBarrier;
import utils.csv.CsvFileReader;
import utils.processes.FileLineProcessingThread;
import utils.processes.ResultFinalizationThread;
import utils.watcher.Watcher;

public class InvoicesSumCalculatorMultithreading {

   private static final String FILE_PATH = new File(Paths.get(".").toString(),
"resources/invoices.csv").getAbsolutePath();
   private static final int NUM_THREADS = 3;
   private static CyclicBarrier barrier;
   private static List<Float> results = new ArrayList<>(NUM_THREADS);

   private static List<Thread> threads = new ArrayList<>(NUM_THREADS);

   public static void main(String[] args) {

       // 1. Start the program and the main thread
       try {
           // initialize a CyclicBarrier to wait for all FileLineProcessingThread to finish, before start
the ResultConsolidationThread
           barrier = new CyclicBarrier(NUM_THREADS, new ResultFinalizationThread(results));
           long beforeUsedMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();

           // 2. Load and process the file invoices.csv
           CsvFileReader csvFileReader = new CsvFileReader(FILE_PATH);
           Watcher watcher = new Watcher();
           watcher.startTimeNanos();
           processPostsByLineMultithreading(csvFileReader);
           watcher.endTimeNanos();

           long afterUsedMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();

           System.out.println("Reading took: " + watcher.timeMillis() + " ms");
           System.out.println("Memory used for the for the whole multithreading program: " +
((afterUsedMemory - beforeUsedMemory) / 1024.0) + " MB");

       } catch (FileNotFoundException ex) {
           System.out.println(FILE_PATH + " does not exists!");
       }
   }

   private static void processPostsByLineMultithreading(CsvFileReader csvFileReader) {
       for (int i = 1; i <= NUM_THREADS; ++i) {
           FileLineProcessingThread thread = new FileLineProcessingThread("Thread #" + i, csvFileReader,
barrier, results);
           thread.start();
           threads.add(thread);
       }
```

```
    }
}
```

```
package utils.csv;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class CsvFileReader implements AutoCloseable {

   private FileReader fr = null;
   private StringBuilder sb = new StringBuilder();
   private int i;

   public CsvFileReader(String fileLocation) throws FileNotFoundException {
       fr = new FileReader(fileLocation);
   }

   public synchronized List<String> getCsvLine() throws IOException {
       sb.setLength(0);
       List<String> fileLine = new ArrayList<>();

       // read every line, split its elements by comma, and put them in fileLine ArrayList
       while ((i = fr.read()) != -1) {
           char c = (char) i;

           if (c == 10) {  // 10 -> NEW LINE (\n)
               for (String element : sb.toString().split(",")) {
                   fileLine.add(element);
               }
               sb.setLength(0);
               return fileLine;
           } else {
               sb.append(c);
           }
       }

       if (sb.length() != 0) {
           for (String element : sb.toString().split(",")) {
               fileLine.add(element);
           }
           sb.setLength(0);
           return fileLine;
       }
       return null;
   }

   @Override
   public void close() { }
}
```

```
package utils.processes;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import utils.csv.CsvFileReader;
import utils.watcher.Watcher;

public class FileLineProcessingThread extends Thread {
   private String threadName;
   private CsvFileReader csvFileReader;
   private CyclicBarrier barrier;
   private List<Float> results;

   public FileLineProcessingThread(String threadName, CsvFileReader csvFileReader, CyclicBarrier barrier,
List<Float> results) {
       this.threadName = threadName;
       this.csvFileReader = csvFileReader;
       this.barrier = barrier;
       this.results = results;
   }

   @Override
   public void run() {
       Watcher watcher = new Watcher();
       watcher.startTimeNanos();
       int fileLinesSize = 0;
       float sumOfAllInvoicesForCurrentThread = 0.0f;
       List<String> fileLine = new ArrayList<>();

       try {
           fileLine = csvFileReader.getCsvLine();

           while (fileLine != null) {
               ++fileLinesSize;

               // check if the sixth element is numeric (the invoice amount)
               if (isNumeric(fileLine.get(5))) {
                   float invoiceAmount = Float.parseFloat(fileLine.get(5));
                   if (isNumeric(fileLine.get(4))){
                       float invoiceQuantity = Float.parseFloat(fileLine.get(4));
                       sumOfAllInvoicesForCurrentThread += invoiceAmount * invoiceQuantity;
                   } else {
                       sumOfAllInvoicesForCurrentThread += invoiceAmount;
                   }

                   //simulate more complicated computational work
                   // Thread.sleep(1);
               } else {
                   if (fileLine.size() < 5) {
                       String lineContent = "[";
                       int elementNumber = 0;
                       for (String element : fileLine) {
                           if ((fileLine.size() - 1) == elementNumber) {
                               lineContent += element.trim() + "]";
                           } else {
                               lineContent += element.trim() + ", ";
                           }
```

```
                            elementNumber++;
                        }

                        System.out.println("Warning: inconsistent line: " + fileLinesSize + "! Content: " +
lineContent);
                        continue;
                    }
                }

                fileLine = csvFileReader.getCsvLine();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        results.add(sumOfAllInvoicesForCurrentThread);

        watcher.endTimeNanos();
        System.out.println("Execution time of thread " + threadName + ": " + watcher.timeMillis() + " ms");
        System.out.println("Sum of all invoices of thread " + threadName + ": " +
sumOfAllInvoicesForCurrentThread);
        System.out.println("File lines size processed by thread " + threadName + ": " + fileLinesSize);

        try {
            // the CyclicBarrier will wait for all FileLineProcessingThread to finish, before start the
ResultConsolidationThread
            barrier.await();
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
    }

    private static boolean isNumeric(String strNum) {
        if (strNum == null) {
            return false;
        }
        try {
            float f = Float.parseFloat(strNum);
        } catch (NumberFormatException nfe) {
            return false;
        }
        return true;
    }
}
```

```java
package utils.processes;

import java.util.List;

public class ResultFinalizationThread extends Thread {
   private List<Float> results;

   public ResultFinalizationThread(List<Float> results) {
       this.results = results;
   }

   @Override
   public void run() {
       System.out.println("Result Finalization Thread started!");
       float sum = 0.0f;

       for (Float result : results) {
           sum += result;
       }

       System.out.println("Invoices sum: " + sum);
   }
}
```

```java
package utils.watcher;

public class Watcher {
   private long startTime = -1;

   public void startTimeNanos() {
       this.startTime = System.nanoTime();
   }

   public long endTimeNanos() {
       return System.nanoTime() - this.startTime;
   }

   public double timeMillis() {
       return this.endTimeNanos() / 1000000.0;
   }
}
```

InvoicesSumCalculatorSingleThreaded class

```
package com.invoices.sum.calculator;

import utils.csv.CsvFileReader;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.List;
import utils.watcher.Watcher;

public class InvoicesSumCalculatorSingleThreaded {

   private static final String FILE_PATH = new File(Paths.get(".").toString(),
"resources/invoices.csv").getAbsolutePath();

   public static void main(String[] args) {

       // 1. Start the program and the main thread
       try {
           long beforeUsedMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();

           // 2. Load and process the file invoices.csv
           CsvFileReader csvFileReader = new CsvFileReader(FILE_PATH);
           Watcher watcher = new Watcher();
           watcher.startTimeNanos();
           processPostsByLineSingleThreaded(csvFileReader);
           watcher.endTimeNanos();

           long afterUsedMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();

           System.out.println("Reading took: " + watcher.timeMillis() + " ms");
           System.out.println("Memory used from a single thread: " + ((afterUsedMemory - beforeUsedMemory)
/ 1024.0) + " MB");

       } catch (FileNotFoundException ex) {
           System.out.println(FILE_PATH + " does not exists!");
       }
   }

   private static void processPostsByLineSingleThreaded(CsvFileReader csvFileReader) {
       Watcher watcher = new Watcher();
       watcher.startTimeNanos();
       int fileLinesSize = 0;
       float sumOfAllInvoicesForCurrentThread = 0.0f;
       List<String> fileLine = new ArrayList<>();

       try {
           fileLine = csvFileReader.getCsvLine();

           while (fileLine != null) {
               ++fileLinesSize;

               // check if the sixth element is numeric (the invoice amount)
               if (isNumeric(fileLine.get(5))) {
                   float invoiceAmount = Float.parseFloat(fileLine.get(5));
                   if (isNumeric(fileLine.get(4))) {
                       float invoiceQuantity = Float.parseFloat(fileLine.get(4));
                       sumOfAllInvoicesForCurrentThread += invoiceAmount * invoiceQuantity;
```

```
                    } else {
                        sumOfAllInvoicesForCurrentThread += invoiceAmount;
                    }
                    //simulate more complicated computational work
                    // Thread.sleep(1);
                } else {
                    if (fileLine.size() < 5) {
                        String lineContent = "[";
                        int elementNumber = 0;
                        for (String element : fileLine) {
                            if ((fileLine.size() - 1) == elementNumber) {
                                lineContent += element.trim() + "]";
                            } else {
                                lineContent += element.trim() + ", ";
                            }
                            elementNumber++;
                        }

                        System.out.println("Warning: inconsistent line: " + fileLinesSize + "! Content: " +
lineContent);
                        continue;
                    }
                }

                fileLine = csvFileReader.getCsvLine();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        watcher.endTimeNanos();
        System.out.println("Execution time for a single thread: " + watcher.timeMillis() + " ms");
        System.out.println("File lines size processed by a single thread: " + fileLinesSize);
        System.out.println("Invoices sum:  " + sumOfAllInvoicesForCurrentThread);
    }

    private static boolean isNumeric(String strNum) {
        if (strNum == null) {
            return false;
        }
        try {
            float f = Float.parseFloat(strNum);
        } catch (NumberFormatException nfe) {
            return false;
        }
        return true;
    }
}
```

2. Експериментална технологична рамка

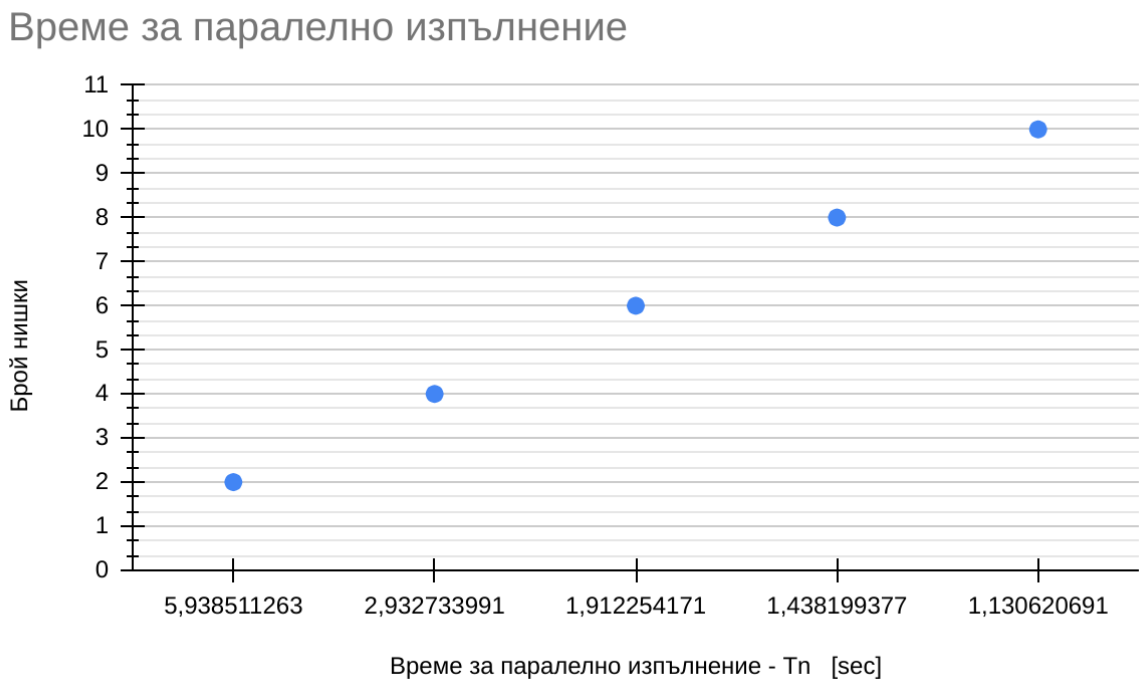
С цел изследване на поведението, бързината, скоростта и ефективността на програмата бяха проведени няколко експеримента с различен брой нишки с цел изчисляване на най-добрият вариант.

Атрибут	Обозначение	Формула	Единица
Размер на машината	**n**	-	бр. ядра
Тактова честота	**f**	-	MHz
Работен товар	**W**	-	Mflops
Време за последователно изпълнение	**T1**	-	sec
Време за паралелно изпълнение	**Tn**	-	sec
Скорост	**Pn**	Pn = W / Tn	Mflops/sec
Ускорение	**Sn**	Sn = T1 / Tn	-
Ефективност	**En**	En = Sn / n	-

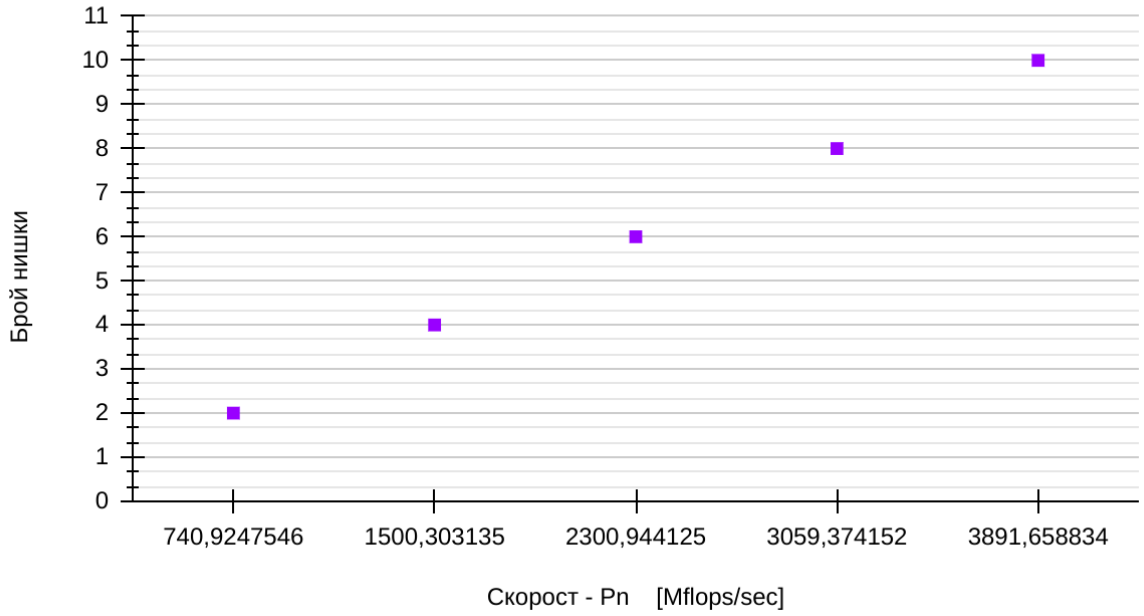
Експериментът беше проведен на машина с процесор Intel® Core(TM) i7-8750H CPU @ 2.20GHz. Физическите ядра на този процесор са 6, но той използва технологията на Intel® Hyper-Threading, чрез която процесорът разделя физическите си ядра на виртуални ядра, които се третират така, сякаш всъщност са физически ядра от операционната система. В конкретнипт случей този процесор е с 6 ядра и използват Hyper-Threading-а за създаване на 12 виртуални ядра. Така че ще приемем, че размерът на машината е n = 12 броя ядра. Всички 12 процесорни ядра имат работният товар 4399.99 Mflops. Процесорът има тактова честота 2.20 GHz, което е равно на 2200 MHz.

Атрибут	Обозначение	Стойност	Единица
Размер на машината	**n**	12	бр. ядра
Тактова честота	**f**	2200	MHz
Работен товар	**W**	4399.99	Mflops

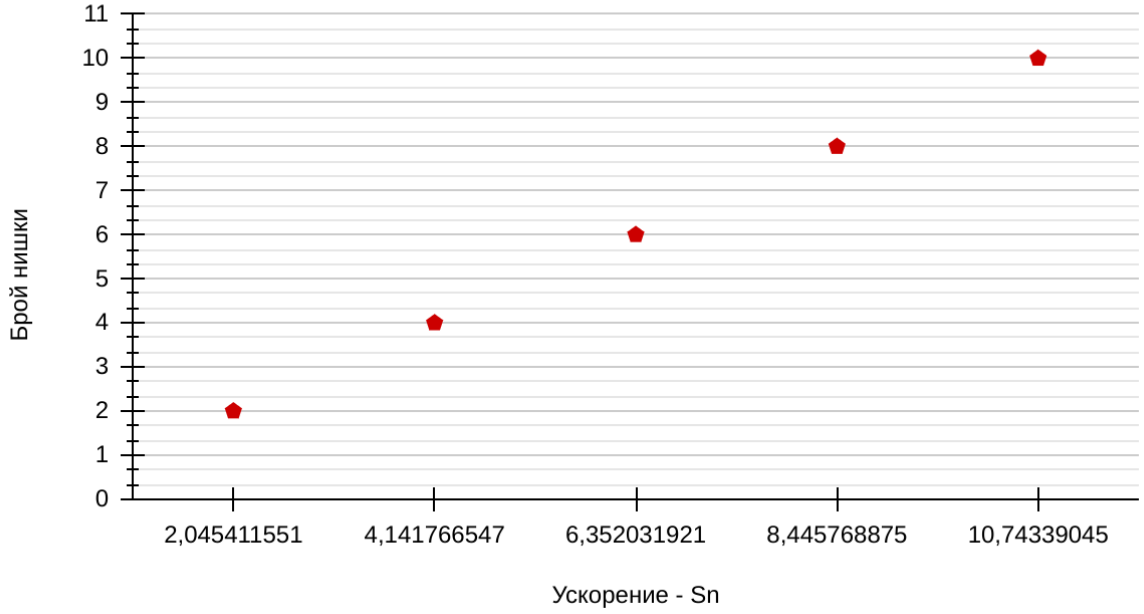
Работен товар - W [Mflops]	4399,99				
Време за последователно изпълнение - T1 [sec]	12,14669954				
Брой нишки	2	4	6	8	10
Време за паралелно изпълнение - Tn [sec]	5,938511263	2,932733991	1,912254171	1,438199377	1,130620691
Скорост - Pn [Mflops/sec]	740,9247546	1500,303135	2300,944125	3059,374152	3891,658834
Ускорение - Sn	2,045411551	4,141766547	6,352031921	8,445768875	10,74339045
Ефективност - En	1,022705776	1,035441637	1,058671987	1,055721109	1,074339045

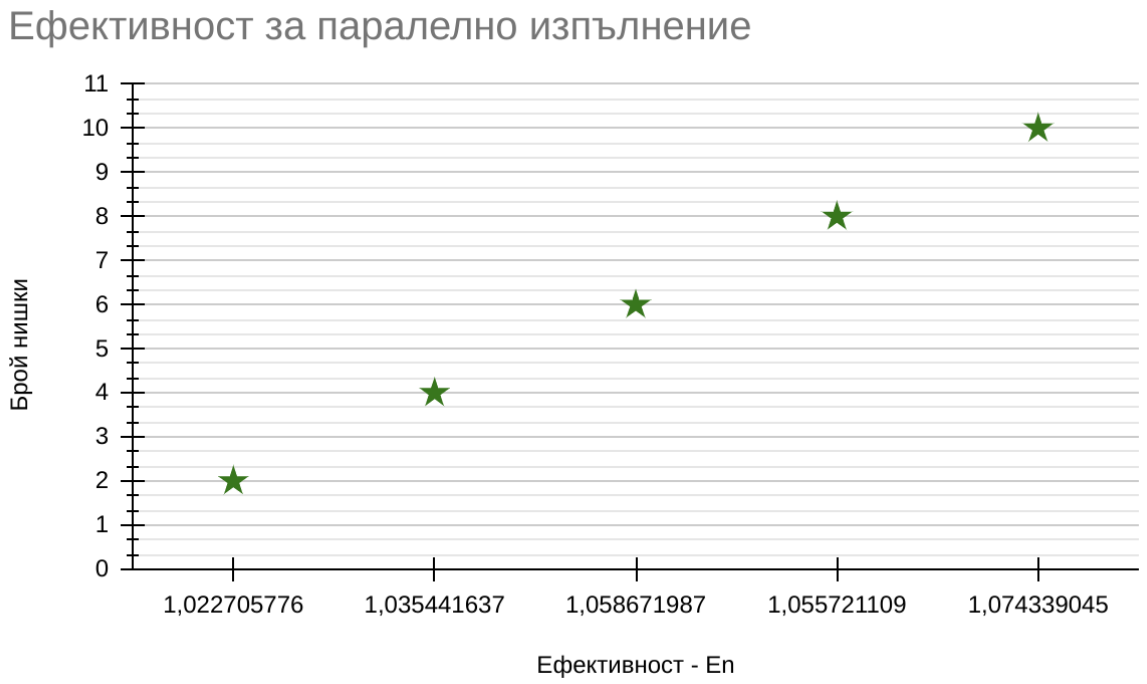


Скорост за паралелно изпълнение



Ускорение за паралелно изпълнение





Част 5 - Резултати от експеримента

```
1 thread - InvoicesSumCalculatorSingleThreaded.java
-----------------------------------------------------
Execution time for a single thread: 12146.699535 ms
File lines size processed by a single thread: 10000
Invoices sum:  2656788.2
Reading took: 12176.839503 ms
Memory used from a single thread: 43302.390625 MB
```

```
2 threads - InvoicesSumCalculatorMultithreading.java
-----------------------------------------------------
Reading took: 11.13696 ms
Memory used for the for the whole multithreading program: 491.5390625 MB
Execution time of thread Thread #1: 5939.425139 ms
Execution time of thread Thread #2: 5938.511263 ms
Sum of all invoices of thread Thread #2: 1331163.9
Sum of all invoices of thread Thread #1: 1325630.8
File lines size processed by thread Thread #2: 5001
File lines size processed by thread Thread #1: 4999
Result Finalization Thread started!
Invoices sum: 2656788.2
```

```
4 threads - InvoicesSumCalculatorMultithreading.java
-----------------------------------------------------
Reading took: 16.212648 ms
Memory used for the for the whole multithreading program: 1064.890625 MB
Execution time of thread Thread #1: 2932.733991 ms
Execution time of thread Thread #3: 2931.733848 ms
Execution time of thread Thread #4: 2931.502864 ms
Execution time of thread Thread #2: 2932.124885 ms
Sum of all invoices of thread Thread #1: 661921.8
Sum of all invoices of thread Thread #2: 662503.7
Sum of all invoices of thread Thread #4: 663492.1
Sum of all invoices of thread Thread #3: 668875.1
File lines size processed by thread Thread #3: 2500
File lines size processed by thread Thread #2: 2501
File lines size processed by thread Thread #1: 2500
File lines size processed by thread Thread #4: 2499
Result Finalization Thread started!
Invoices sum: 2656788.2
```

```
6 threads - InvoicesSumCalculatorMultithreading.java
-----------------------------------------------------
Reading took: 13.387816 ms
Memory used for the for the whole multithreading program: 1556.46875 MB
Execution time of thread Thread #3: 1910.341228 ms
Execution time of thread Thread #2: 1910.547334 ms
Execution time of thread Thread #1: 1912.254171 ms
Execution time of thread Thread #4: 1911.321964 ms
Execution time of thread Thread #6: 1911.113805 ms
Execution time of thread Thread #5: 1910.678237 ms
Sum of all invoices of thread Thread #6: 432005.88
Sum of all invoices of thread Thread #4: 447387.16
Sum of all invoices of thread Thread #3: 444153.97
Sum of all invoices of thread Thread #2: 441764.03
Sum of all invoices of thread Thread #5: 441018.1
Sum of all invoices of thread Thread #1: 450464.12
File lines size processed by thread Thread #6: 1661
File lines size processed by thread Thread #1: 1666
File lines size processed by thread Thread #3: 1668
File lines size processed by thread Thread #2: 1673
File lines size processed by thread Thread #4: 1662
File lines size processed by thread Thread #5: 1670
Result Finalization Thread started!
Invoices sum: 2656788.2
```

```
8 threads - InvoicesSumCalculatorMultithreading.java
-----------------------------------------------------
Reading took: 16.824377 ms
Memory used for the for the whole multithreading program: 3031.0703125 MB
Execution time of thread Thread #8: 1437.133038 ms
Execution time of thread Thread #7: 1437.670986 ms
Execution time of thread Thread #3: 1436.997481 ms
Execution time of thread Thread #6: 1436.932586 ms
Execution time of thread Thread #4: 1438.037244 ms
Execution time of thread Thread #1: 1437.094625 ms
Execution time of thread Thread #2: 1438.199377 ms
Execution time of thread Thread #5: 1438.009215 ms
Sum of all invoices of thread Thread #1: 339764.88
Sum of all invoices of thread Thread #2: 334265.56
Sum of all invoices of thread Thread #8: 324815.06
Sum of all invoices of thread Thread #3: 338719.12
Sum of all invoices of thread Thread #4: 332552.62
Sum of all invoices of thread Thread #6: 335678.06
Sum of all invoices of thread Thread #7: 335720.66
Sum of all invoices of thread Thread #5: 315277.06
File lines size processed by thread Thread #8: 1243
File lines size processed by thread Thread #7: 1253
File lines size processed by thread Thread #6: 1248
File lines size processed by thread Thread #1: 1252
File lines size processed by thread Thread #2: 1250
File lines size processed by thread Thread #3: 1251
File lines size processed by thread Thread #5: 1247
File lines size processed by thread Thread #4: 1256
Result Finalization Thread started!
Invoices sum: 2656788.2
```

```
10 threads - InvoicesSumCalculatorMultithreading.java
-----------------------------------------------------
Reading took: 13.635819 ms
Memory used for the for the whole multithreading program: 3112.90625 MB
Execution time of thread Thread #6: 1130.132833 ms
Execution time of thread Thread #8: 1128.921756 ms
Execution time of thread Thread #4: 1129.655975 ms
Execution time of thread Thread #3: 1129.722024 ms
Execution time of thread Thread #9: 1129.694425 ms
Execution time of thread Thread #5: 1130.269559 ms
Execution time of thread Thread #7: 1130.231167 ms
Execution time of thread Thread #2: 1130.620691 ms
Execution time of thread Thread #10: 1129.809512 ms
Execution time of thread Thread #1: 1129.831789 ms
Sum of all invoices of thread Thread #7: 269425.97
Sum of all invoices of thread Thread #9: 262524.75
Sum of all invoices of thread Thread #3: 273199.34
Sum of all invoices of thread Thread #4: 261790.38
Sum of all invoices of thread Thread #6: 277591.75
Sum of all invoices of thread Thread #8: 260602.7
Sum of all invoices of thread Thread #10: 261876.3
Sum of all invoices of thread Thread #1: 263846.84
Sum of all invoices of thread Thread #2: 259826.84
Sum of all invoices of thread Thread #5: 266107.7
File lines size processed by thread Thread #6: 1002
File lines size processed by thread Thread #7: 1005
File lines size processed by thread Thread #9: 989
File lines size processed by thread Thread #2: 1006
File lines size processed by thread Thread #10: 1000
File lines size processed by thread Thread #5: 1002
File lines size processed by thread Thread #4: 1002
File lines size processed by thread Thread #3: 1008
File lines size processed by thread Thread #8: 997
File lines size processed by thread Thread #1: 989
Result Finalization Thread started!
Invoices sum: 2656788.2
```

Част 6 - Заключение

Програмните езици поддържат различни подходи свързани с нишковото програмиране, като в този проект бяха разгледани някои основни концепции, както и тяхната същност и имплементация. Програмирането с нишки е изключително важно за всяко по-мащабно приложение, особено, ако то обработва, запазва и създава данни. Всеки програмист трябва да знае как да оптимизира създадените от него програми, така че да работят максимално надеждно, бързо и оптимално.