**Project #1**





과목명 | 멀티코어 컴퓨팅

담당교수 | 손봉수 교수님

학과 | 소프트웨어학과

학번 | 20184806

이름 | 이상은

제출일 | 2021. 04. 26

**Environment**

**CPU** : Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz 2.90 GHz

**OS** : Windows 10 Pro

**Number Of Cores :** 6

**Number of Logical Processors :** 6

**Memory Size** : RAM 16.0GB

**GitHub** : <https://github.com/HaruToy/MulticoreComputing.git>

**Problem 1. Count Prime Number**

The problem of obtaining a prime number up to N

**IDEA**

**Static Load Balancing**

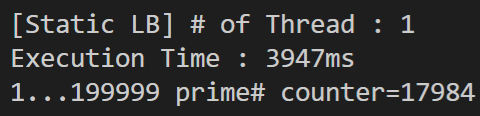
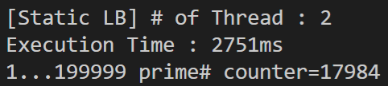
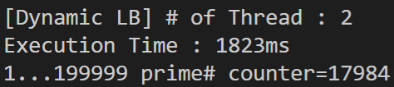
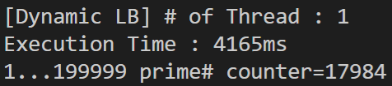
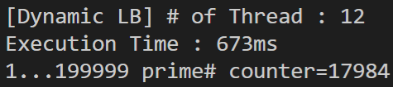
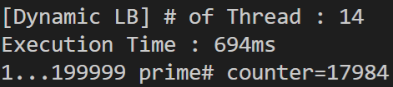
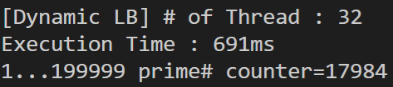
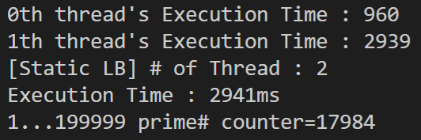
1. For each thread, Determine whether the number within a certain range is a prime number and hand over the result.

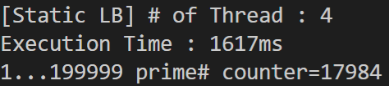
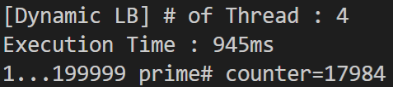
Navigation range of the i-th thread : (N/(NUM\_THREAD))\*i+1 ~ (N/(NUM\_THREAD))\*(i+1)

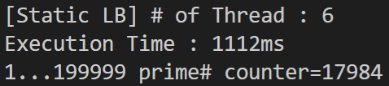
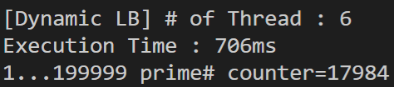
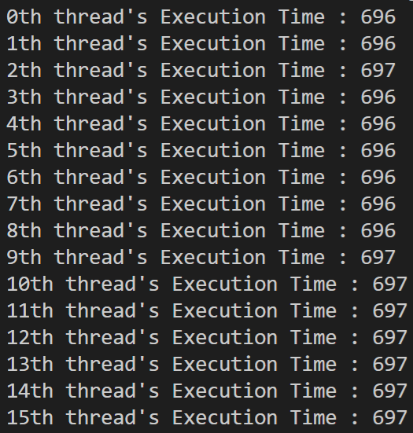
(For the last thread, the range is (N/(NUM\_THREAD))\*i+1 ~ N)

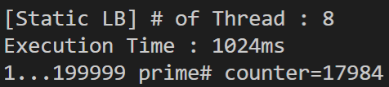
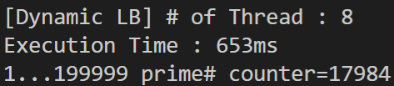
1. Output the sum of the results returned by the thread.

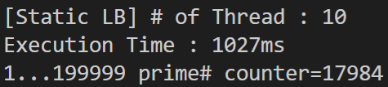
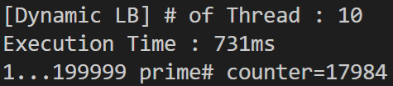
**Dynamic Load Balancing**

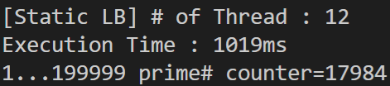
1. Each Thread checks if the value of the variable, atomic integer, is a prime number or not, Increment the value of the variable which other threads check.
2. Terminate Program when the value of the variable, Atomic integer, is same with N.
3. Output the sum of the results returned by the thread.
4. **Result**
5. 

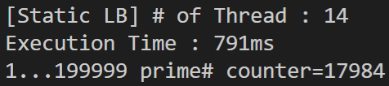


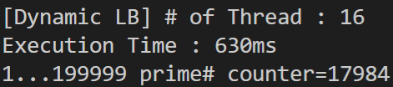


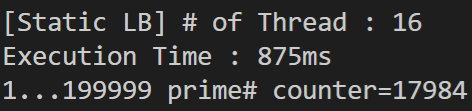
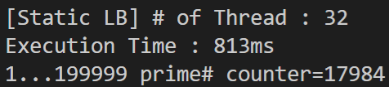












**[image 1-2]** Result of prime number

**[image 1-1]** Result of prime number

The result was 17984, which was the same for all threads. Although not mentioned in Image 1-1, each output outputs execution time on the thread, as shown in 1-2.

1. **Average Execution Time of each thread**

(단, 소수점 아래 반올림)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Exec time(ms) | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 32 |
| **Static** | 3621 | 1838 | 934 | 651 | 604 | 503 | 570 | 598 | 500 | 634 |
| **Dynamic** | 4604 | 2003 | 976 | 820 | 785 | 702 | 756 | 687 | 686 | 689 |

**[table 1]** Average execution time of each thread

As the number of threads increased, the average Execution time of each thread decreased. The total amount of operations is fixed, and because several threads are divided, the execution time decreases as there are more threads.

In Static Load Balancing, there was a large difference in running time for each thread, which was not efficient. We believe that allocating the remaining numbers to threads of each number will mitigate the difference in execution time.

1. **Execution Time & Performance when using all threads**

**[graph 1]** Execution Time and Performance per number of threads

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Exec time(ms)** | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 32 |
| **Static** | 3624 | 2708 | 1567 | 1111 | 1000 | 863 | 871 | 886 | 781 | 770 |
| **Dynamic** | 4927 | 1828 | 1030 | 700 | 745 | 721 | 714 | 764 | 701 | 695 |

(단, 소수점 아래 반올림)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Performance** | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 32 |
| **Static** | 2.7e-4 | 3.6e-4 | 6.3e-4 | 9.0e-4 | 0.001 | 0.0011 | 0.0011 | 0.0011 | 0.0012 | 0.0012 |
| **Dynamic** | 2.0e-4 | 5.4e-4 | 9.7e-4 | 0.0014 | 0.0013 | 0.0013 | 0.0014 | 0.0013 | 0.0014 | 0.0014 |

(단, 소수점 아래 다섯 번째자리서 반올림)

**[table 2]** Execution Time and Performance per number of threads

As the number of threads increased, performance improved and execution time decreased. Overall, Dynamic Load Balancing performs better than Static Load Balancing. I believe that Dynamic Load Balancing was more efficient due to unequal workload distribution in Static Load Balancing. There was no significant difference in execution time between Static Load Balancing and Dynamic Load Balancing.

The execution time was slightly larger than the execution time of the longest thread. Because each thread is processed in parallel.

1. **Source Code**

**Static Load Balancing**

package problem1;

public class pc\_static {

    private static int NUM\_END = 200000;

    static int NUM\_THREAD=32;

    public static void main(String[] args){

        int result;

        long startTime = System.currentTimeMillis();

        result=prime(NUM\_END);

        long endTime = System.currentTimeMillis();

        long timeDiff = endTime - startTime;

        System.out.println("[Static LB] # of Thread : "+NUM\_THREAD+"\nExecution Time : "+timeDiff+"ms");

        System.out.println("1..."+(NUM\_END-1)+" prime# counter="+result+"\n");

    }

    static int prime(int len){

        int ans=0;

        long[] ThreadTime = new long[NUM\_THREAD];

        PrimeThread[] pt = new PrimeThread[NUM\_THREAD];

        for(int i=0;i<NUM\_THREAD;i++){

            //For each thread, Determine whether the number within a certain range is a prime number and hand over the result.

            if(i==NUM\_THREAD-1)

            {

                pt[i] = new PrimeThread(((NUM\_END)/(NUM\_THREAD))\*i+1,NUM\_END);

            }

            else

            {

                pt[i] = new PrimeThread(((NUM\_END)/(NUM\_THREAD))\*i+1,((NUM\_END)/(NUM\_THREAD))\*(i+1));

            }

            long startThreadTime = System.currentTimeMillis();

            ThreadTime[i]=startThreadTime;

            pt[i].start();

        }

        try {

            for(int i=0; i < NUM\_THREAD; i++) {

              pt[i].join();

              long endThreadTime = System.currentTimeMillis();

              ThreadTime[i]=endThreadTime-ThreadTime[i];

              System.out.println(i+" "+ThreadTime[i]+" ");

              //Output the sum of the results returned by the thread.

              ans += pt[i].ans;

            }

          } catch (InterruptedException IntExp) {

          }

        return ans;

    }

}

class PrimeThread extends Thread{

    int lo;

    int hi;

    int ans=0;

    PrimeThread(int l, int h){

        lo=l; hi=h;

    }

    public void run(){

        //Check Prime Number in Thread

        for(int i=lo;i<hi+1;i++)

          if(isPrime(i))ans++;

    }

    private static boolean isPrime(int x){

        int i;

        if(x<=1) return false;

        for(i=2;i<x;i++){

            if((x%i==0)&&(i!=x)) return false;

        }

        return true;

    }

}

**Dynamic Load Balancing**

package problem1;

import java.util.concurrent.atomic.\*;

public class pc\_dynamic {

    private static int NUM\_END = 200000;

    static int NUM\_THREAD=32;

    public static void main(String[] args){

        int result;

        long startTime = System.currentTimeMillis();

        result=prime(NUM\_END);

        long endTime = System.currentTimeMillis();

        long timeDiff = endTime - startTime;

        System.out.println("[Dynamic LB] # of Thread : "+NUM\_THREAD+"\nExecution Time : "+timeDiff+"ms");

        System.out.println("1..."+(NUM\_END-1)+" prime# counter="+result+"\n");

    }

    static AtomicInteger number=new AtomicInteger(1);

    static int prime(int len){

        int ans=0;

        long[] ThreadTime = new long[NUM\_THREAD];

        PrimeDThread[] pt = new PrimeDThread[NUM\_THREAD];

            for(int i=0;i<NUM\_THREAD;i++){

                pt[i] = new PrimeDThread(number,NUM\_END);

                long startThreadTime = System.currentTimeMillis();

                ThreadTime[i]=startThreadTime;

                pt[i].start();

            }

        try {

            for(int i=0; i < NUM\_THREAD; i++) {

              pt[i].join();

              long endThreadTime = System.currentTimeMillis();

              ThreadTime[i]=endThreadTime-ThreadTime[i];

              System.out.println(ThreadTime[i]+" ");

              //Output the sum of the results returned by the thread.

              ans += pt[i].ans;

            }

          } catch (InterruptedException IntExp) {

          }

        return ans;

    }

}

class PrimeDThread extends Thread{

    AtomicInteger number;

    int ans=0;

    int end;

    PrimeDThread(AtomicInteger n,int e){

        number=n; end=e;

    }

    //Each Thread checks if the value of the variable, atomic integer, is a prime number or not until N, Increment the value of the variable which other threads check.

    public void run(){

        int n=number.incrementAndGet();

        while(n<end)

        {

            if(isPrime(n))ans++;

            n=number.incrementAndGet();

        }

    }

    private static boolean isPrime(int x){

        int i;

        if(x<=1) return false;

        for(i=2;i<x;i++){

            if((x%i==0)&&(i!=x)) return false;

        }

        return true;

    }

}

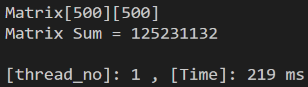
**Problem 2. Multiply Matrix**

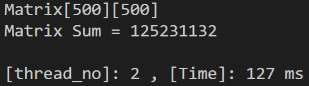
Problems multiplying matrices with threads

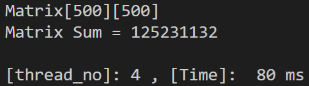
**IDEA**

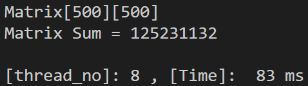
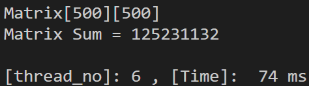
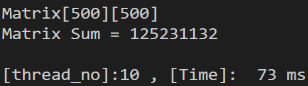
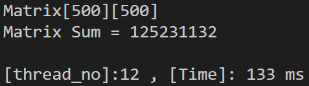
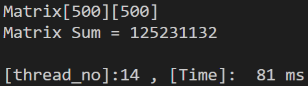
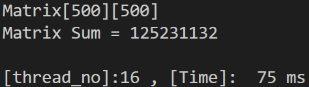
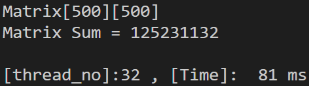
1. For each thread, Determine whether the number within a certain range is a prime number and hand over the result.
2. Navigation range of the i-th thread : (N/(NUM\_THREAD))\*i~ (N/(NUM\_THREAD))\*(i+1)

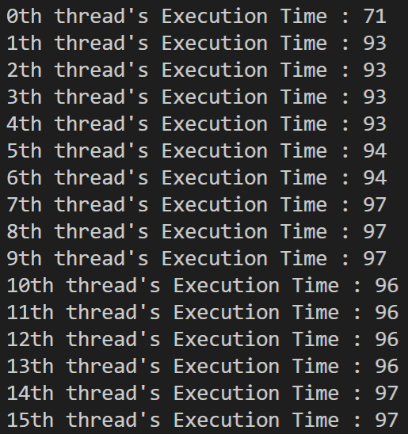
(For the last thread, the range is (N/(NUM\_THREAD))\*i~ N)

1. Each thread multiplies a matrix that corresponds to a range by a matrix.
2. **Result**









**[image 2-2]** execution time of each thread

**[image 2-1]** Sum of all elements in the resulting matrix

The result was 125231132, which was the same for all threads. Although not mentioned in Image 2-1, each output outputs execution time on the thread, as shown in 2-2.

1. **Average execution time of each thread**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 32 |
| Exec time(ms) | 213 | 122 | 77 | 70 | 72 | 103 | 79 | 122 | 60 | 64 |

**[table 3]** Average execution time of each thread

1. **execution time when using all threads**

**[graph 2]** Execution Time and Performance per number of threads

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 32 |
| Exec time | 223 | 123 | 114 | 126 | 100 | 122 | 98 | 114 | 91 | 85 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 32 |
| Performance | 0.0044 | 0.0081 | 0.0087 | 0.0079 | 0.01 | 0.0081 | 0.0102 | 0.0087 | 0.0109 | 0.0117 |

**[table 4]** Execution Time and Performance per number of threads

As the number of threads increased, performance improved and execution time decreased. The total amount of operations is fixed, and because several threads are divided, the execution time decreases as there are more threads.

The execution time was slightly larger than the execution time of the longest thread. Because each thread is processed in parallel.

1. **Source Code**
2. package problem2;
3. import java.util.\*;
4. import java.io.File;
5. import java.io.FileNotFoundException;
6. //import java.lang.\*;
7. // command-line execution example) java MatmultD 6 mat500.txt
8. // 6 means the number of threads to use
9. // mat500.txt means the file that contains two matrices is given as standard input
10. public class MatmultThread
11. {
12. static Scanner sc = new Scanner(System.in);
13. static int thread\_no=0;
14. static int n;
15. static int p;
16. static int ans[][];
17. public static void main(String [] args)
18. {
19. String path = System.getProperty("user.dir");
20. File file = new File(path+"\\Project1\\problem2\\"+args[1]);
21. try{
22. sc = new Scanner(file);
24. if (args.length==2) thread\_no = Integer.valueOf(args[0]);
25. else thread\_no = 1;
26. int a[][]=readMatrix();
27. int b[][]=readMatrix();
28. long startTime = System.currentTimeMillis();
29. int[][] c=multMatrix(a,b);
30. long endTime = System.currentTimeMillis();
31. //printMatrix(a);
32. //printMatrix(b);
33. printMatrix(c);
34. //System.out.printf("thread\_no: %d\n" , thread\_no);
35. //System.out.printf("Calculation Time: %d ms\n" , endTime-startTime);
36. System.out.printf("[thread\_no]:%2d , [Time]:%4d ms\n", thread\_no, endTime-startTime);
37. }
38. catch(FileNotFoundException e){
39. e.printStackTrace();
40. }
41. }
42. public static int[][] readMatrix() {
43. int rows = sc.nextInt();
44. int cols = sc.nextInt();
45. int[][] result = new int[rows][cols];
46. for (int i = 0; i < rows; i++) {
47. for (int j = 0; j < cols; j++) {
48. result[i][j] = sc.nextInt();
49. }
50. }
51. return result;
52. }
53. public static void printMatrix(int[][] mat) {
54. System.out.println("Matrix["+mat.length+"]["+mat[0].length+"]");
55. int rows = mat.length;
56. int columns = mat[0].length;
57. int sum = 0;
58. for (int i = 0; i < rows; i++) {
59. for (int j = 0; j < columns; j++) {
60. //System.out.printf("%4d " , mat[i][j]);
61. sum+=mat[i][j];
62. }
63. //System.out.println();
64. }
65. //System.out.println();
66. System.out.println("Matrix Sum = " + sum + "\n");
67. }
68. public static int[][] multMatrix(int a[][], int b[][]){//a[m][n], b[n][p]
69. long[] ThreadTime = new long[thread\_no];
70. multiMThread[] mt = new multiMThread[thread\_no];
71. if(a.length == 0) return new int[0][0];
72. if(a[0].length != b.length) return null; //invalid dims
73. n = a[0].length;
74. int m = a.length;
75. p = b[0].length;
76. ans = new int[m][p];
77. for(int i=0;i<thread\_no;i++){
78. if(i==thread\_no-1)
79. {
80. mt[i] = new multiMThread(((m)/(thread\_no))\*i,m,ans,a,b,p,n);
81. }
82. else
83. {
84. mt[i] = new multiMThread(((m)/(thread\_no))\*i,((m)/(thread\_no))\*(i+1),ans,a,b,p,n);
85. }
86. long startThreadTime = System.currentTimeMillis();
87. ThreadTime[i]=startThreadTime;
88. mt[i].start();
89. }
90. try {
91. for(int i=0; i < thread\_no; i++) {
92. mt[i].join();
93. long endThreadTime = System.currentTimeMillis();
94. ThreadTime[i]=endThreadTime-ThreadTime[i];
95. System.out.println(i+"th thread's Execution Time : "+ThreadTime[i]+" ");
96. }
97. } catch (InterruptedException IntExp) {
98. }
99. return ans;
100. }
101. }
102. class multiMThread extends Thread{
103. int lo;
104. int hi;
105. int[][] ans;
106. int[][] a;
107. int[][] b;
108. int p;
109. int n;
110. multiMThread(int l, int h,int[][] an,int[][] ad,int[][] bd,int pd,int nd){
111. lo=l; hi=h; ans=an; a=ad; b=bd; p=pd; n=nd;
112. }
113. public void run(){
114. for(int i = lo;i < hi;i++){
115. for(int j = 0;j < p;j++){
116. for(int k = 0;k < n;k++){
117. ans[i][j] += a[i][k] \* b[k][j];
118. }
119. }
120. }
121. }
122. }