Project #1

Multicore Computing

Problem 1

Date	May 10, 2020
Instructor	Bongsoo Sohn
Std. Name	Junhyuck Woo
Std. ID	20145337

INDEX

ENVIRONMENT

Hardware

- MacBook Pro (15-inch, 2017)
- Processor: 2.8 GHz Quad-Core Intel Core i7
- Memory: 16GB 2133 MHz LPDDR3

Operating System

- macOS Catalina, ver: 10.15.4

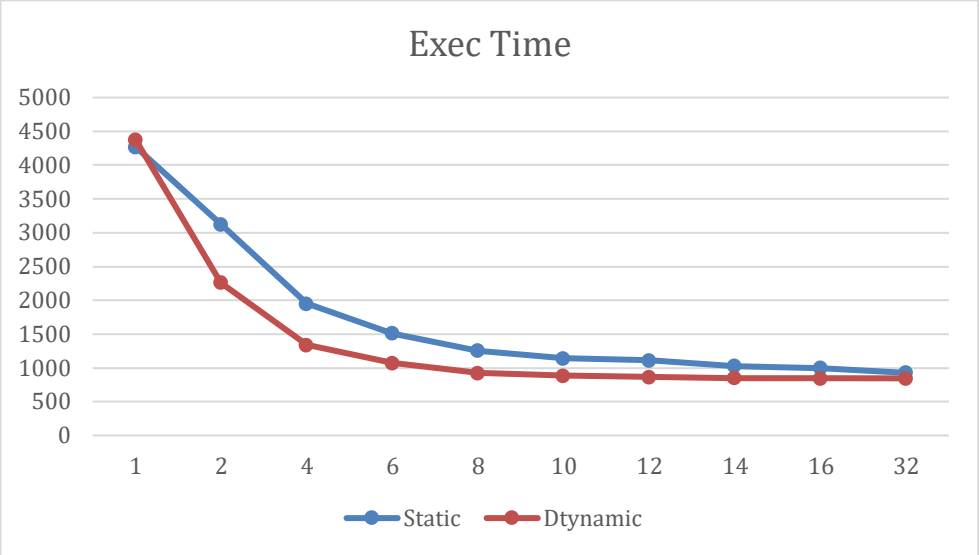
IDE (Integrated Development Environment)

- IntelliJ IDEA 2019.3.4 (Ultimate Edition)
- Java version “11.0.6”

Testing Environment

- iTerm2
- Build 3.3.9
- openjdk 14.0.1 2020-04-14
  OpenJDK Runtime Environment (build 14.0.1+7)
  OpenJDK 64-Bit Server VM (build 14.0.1+7, mixed mode, sharing)

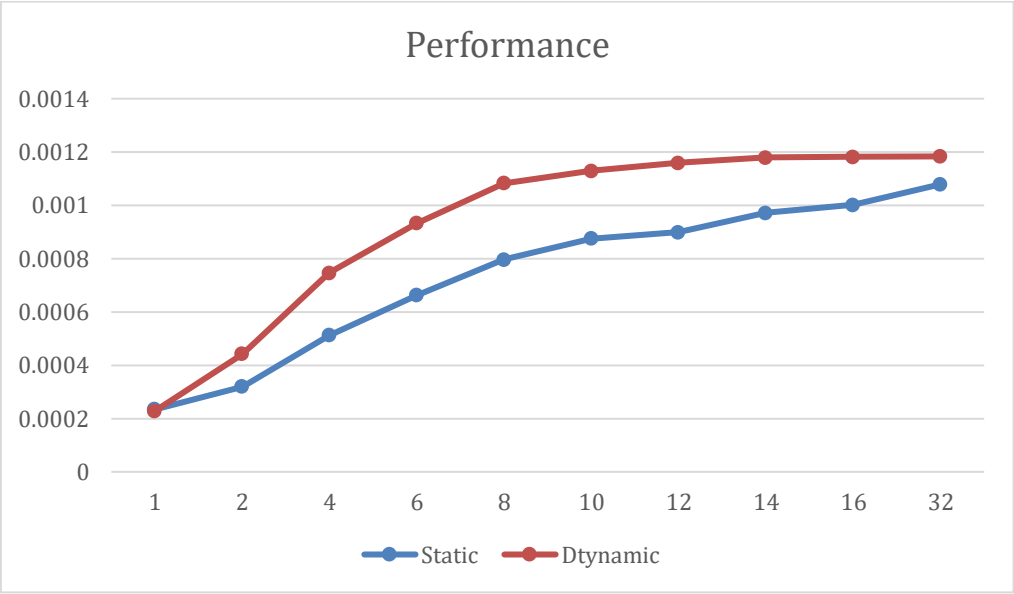
TABLE & GRAPH



▲ Fig. 1. Exec Time of multi-thread programming with static and dynamic load balancing approach

Exec Time	1	2	4	6	8	10	12	14	16	**32**
Static	4271	3124	1949	1510	1256	1142	1113	1029	998	927
Dynamic	4377	2261	1339	1072	923	886	863	848	846	845

▲ TABLE 1



▲ Fig. 2. Performance of multi-thread programming with static and dynamic load balancing approach

Performance **(1/ Exec Time)**	1	2	4	6	8	10	12	14	16	**32**
Static	0.000 23414	0.000 3201	0.000 51308	0.000 66225	0.000 79618	0.000 87566	0.000 89847	0.000 97182	0.001 002	0.001 07875
Dynamic	0.000 22847	0.000 44228	0.000 74683	0.000 93284	0.001 08342	0.001 12867	0.001 15875	0.001 17925	0.001 18203	0.001 18343

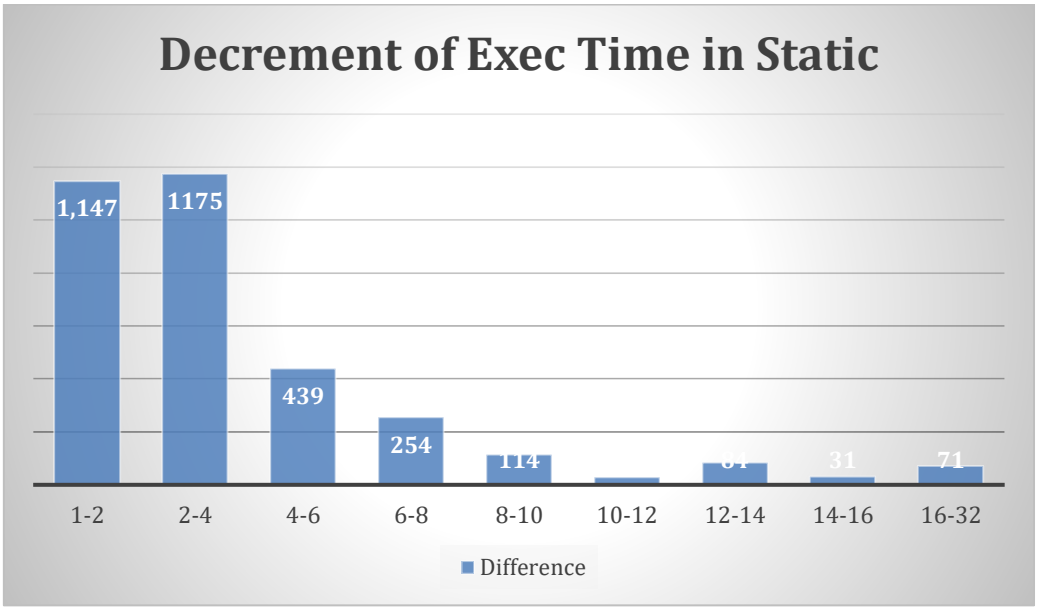
▲ TABLE 2

EXPLANATION ON RESULT

In this problem, the given work is counting prime number. The way checking whether number is prime or not is to divide every numbers which under the given number. It means that the bigger number consumes more time than smaller one. When I wrote the code using the static balancing approach, I assigned the work sequentially.

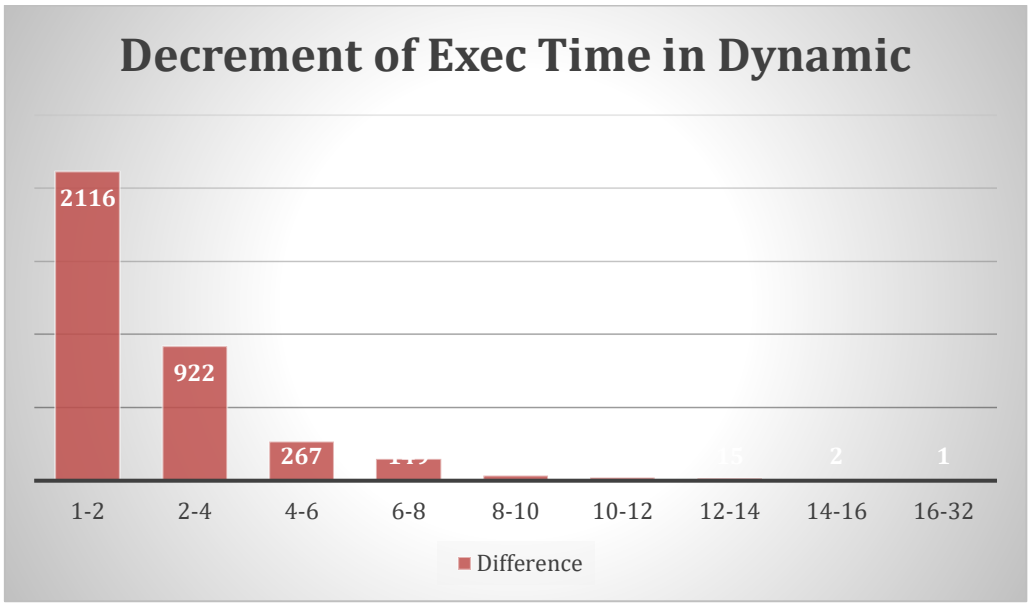
1. Exec Time

The Fig. 1 shows the decrement tendency. To see the detail, I make 3 more figures.



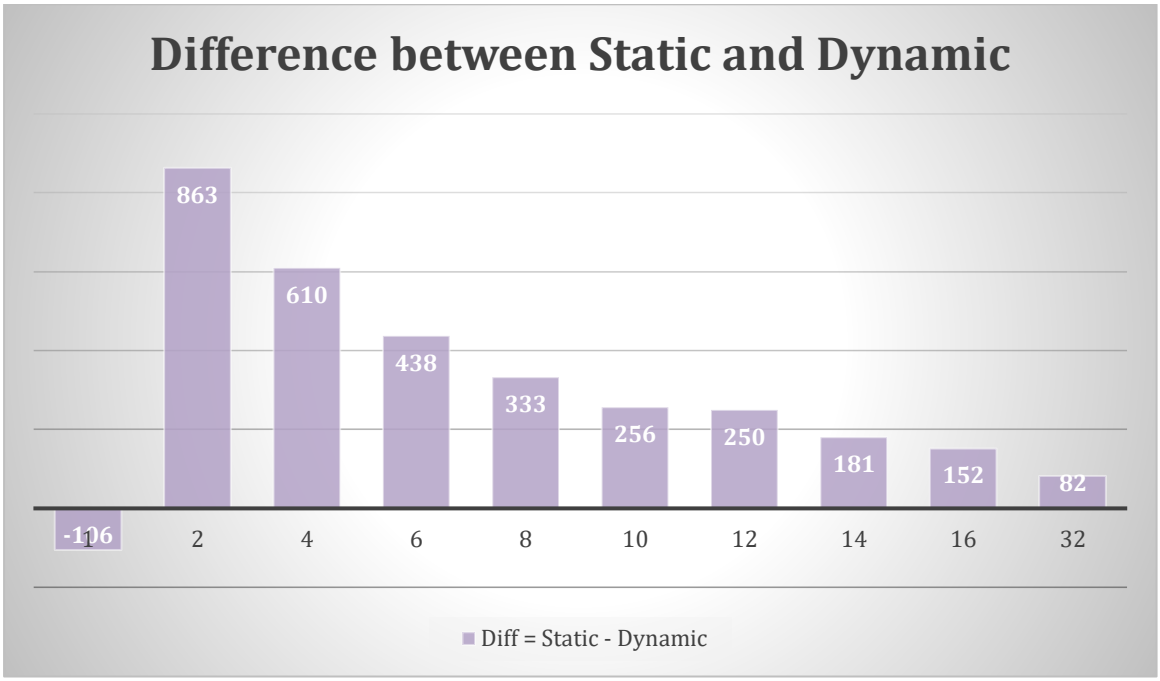
▲ Fig. 3. Decrement of execution time in static load balancing approach

The Fig. 3 shows how the execution time has decreased as the thread increases at the static load balancing approach. The significant results appear to be available when increasing from 1 to 8. As mentioned earlier, since the work was assigned sequentially, how few large numbers were allocated seems to have had the greatest impact. For 8–32, although the tendency is decreasing, it is not considered to be a significant achievement. I think the work has been distributed enough for each thread to cover.



▲ Fig. 4. Decrement of execution time in dynamic load balancing approach

Fig. 4 shows how the execution time has decreased as the thread increases. The significant results appear to be available when increasing from 1 to 8. What's unusual is that unlike static load balancing, the reduction time is roughly halved. Because dynamic load balancing draws work from the work queue, it is thought that a relatively similar amount of work is allocated, so execution time is determined in proportion to the increase and decrease of the thread. For 8–32, the trend is decreasing, but it does not appear to be a significant achievement at all. I think it's because each thread is assigned too little work.



▲ Fig. 5. Difference between static and dynamic load balancing approach

If you look at the fig5, you can see that the overall "dynamic load balancing approach" has a shorter running time. I think it's because load balancing went well because we brought work from one work queue.

2. Performance

The performance evaluation criterion is 1/exe time, so if the run time is reduced, the performance is higher. Figure 2 shows that in the case of static, it appears to be a relative primary polynomial, and dynamic is shaped like a curve. It appears to be the result of the amount of work and the allocation method mentioned earlier.

SOURCE CODE

1. pc_static.java

```
// Writer: Junhyuck Woo
// Lecture: Multicore Computing
// Organization: Chung-Ang University
// Deadline: May 10, 2020
// Project #1
//  - problem 1-1

import java.awt.desktop.SystemEventListener;

class pc_static {
    // Given Variable
    private static final int NUM_END = 200000;
    private static final int NUM_THREAD = 1;

    // Main Method
    public static void main(String[] args) {
        PrimeOperator1 po = new PrimeOperator1(NUM_END,
NUM_THREAD);
        po.run();
    }
}

class PrimeOperator1 {
    // Variable for getting a number of thread and end of number
    int num_thread = 0;
    int num_end = 0;

    // Constructor
    public PrimeOperator1(final int NUM_END, final int NUM_THREAD) {
        num_end = NUM_END;
        num_thread = NUM_THREAD;
    }

    // Class runner
    public void run() {
        // Variables
        Prime1[] pn = new Prime1[num_thread];
        int[] start_num = new int[num_thread];
        int[] end_num = new int[num_thread];
        int prime_num = 0;
        long total_time = 0;

        // Split the number as same as
        for (int i=0; i<num_thread; i++) {
            int gap = num_end / num_thread;
            start_num[i] = gap * i;
            end_num[i] = start_num[i] + gap;
            if (i == num_thread-1) {
                end_num[i] = num_end;
            }
        }

        // Set timer
        long startTime = System.currentTimeMillis();

        // Run
        for (int i=0; i<num_thread; i++) {
            pn[i] = new Prime1(start_num[i], end_num[i]);
            pn[i].start();
        }

        // Wait the work is done
        for (int i=0; i<num_thread; i++) {
            try {
                pn[i].join();
                prime_num += pn[i].getPrimeNum();
            }
            catch (InterruptedException e) {}
        }

        // Finish timer
        long endTime = System.currentTimeMillis();
        long runTime = endTime - startTime;

        // Visualize the execution time of each thread
        for (int i=0; i<num_thread; i++) {
            System.out.println(pn[i].getName() + ": " + pn[i].getRunTime()
+"ms");
        }

        // Visualize the total execution time
        System.out.println("Total Execution Time: "+ runTime +"ms");

        // Visualize the total number of prime number
        System.out.println("Total number of prime number: " + prime_num);
    }
}

class Prime1 extends Thread {
    // Variables
    private long runTime = 0;
    private int prime_num = 0;
    private int start = 0;
    private int end = 0;

    // Constructor
    public Prime1(int start_num, int end_num) {
        start = start_num;
        end = end_num;
    }

    // Runner
    public void run() {
        // Set a timer
        long startTime = System.currentTimeMillis();
        // Check the number is prime or not
        for (int i=start; i<end; i++) {
            if (isPrime(i)) {
                count();
            }
        }
        // Check the run time
        long endTime = System.currentTimeMillis();
        runTime = endTime - startTime;
    }

    // Getter - runt time
    public long getRunTime() {
        return runTime;
    }

    // Getter - Prime number
    public long getPrimeNum() {
        return prime_num;
    }

    // Count the prime number with synchronization
    public synchronized void count() {
        prime_num++;
    }

    // Check whether the number is prime
    // If it is prime, then return True
    // Else return false
    private 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;
    }
}
```

2. pc_dynamic.java

```
// Writer: Junhyuck Woo
// Lecture: Multicore Computing
// Organization: Chung-Ang University
// Deadline: May 10, 2020
// Project #1
//  - problem 1-2

import java.awt.desktop.SystemEventListener;

class pc_dynamic {
    // Given Variables
    private static final int NUM_END = 200000;
    private static final int NUM_THREAD = 1;

    // Main Method
    public static void main(String[] args) {
        PrimeOperator2 po = new PrimeOperator2(NUM_END,
NUM_THREAD);
        po.run();
    }
}

class PrimeOperator2 {
    // Variable for getting a number of thread and end of number
    int num_end = 0;
    int num_thread = 0;
    private int[] work_queue = new int[1];

    // Constructor
    public PrimeOperator2(final int NUM_END, final int NUM_THREAD) {
        work_queue[0] = 0;
        num_end = NUM_END;
        num_thread = NUM_THREAD;
    }

    public void run() {
        // Variables
        Prime2[] pn = new Prime2[num_thread];
        int prime_num = 0;

        // Set timer
        long startTime = System.currentTimeMillis();

        // Runner
        for (int i=0; i<num_thread; i++) {
            pn[i] = new Prime2(work_queue, num_end);
            pn[i].start();
        }

        // Wait the work is done
        for (int i=0; i<num_thread; i++) {
            try {
                pn[i].join();
                prime_num += pn[i].getPrimeNum();
            }
            catch (InterruptedException e) {}
        }

        // Finish timer
        long endTime = System.currentTimeMillis();
        long runTime = endTime - startTime;

        // Visualize the execution time of each thread
        for (int i=0; i<num_thread; i++) {
            System.out.println(pn[i].getName() + ": " + pn[i].getRunTime()
+"ms");
        }

        // Visualize the total execution time
        System.out.println("Total Execution Time: "+ runTime +"ms");

        // Visualize the total number of prime number for debugging the result
        System.out.println("Total number of prime number: " + prime_num);
    }

}

class Prime2 extends Thread {
    // Variable
    private long runTime = 0;
    private int prime_num = 0;
    private int start = 0;
    private int end = 0;
    private int[] work_queue = new int[1];

    // Constructor
    public Prime2(int[] work_queue, int end_num) {
        end = end_num;
        this.work_queue = work_queue;
    }

    // Runner
    public void run() {
        // Set a timer
        long startTime = System.currentTimeMillis();

        // Check the number is prime or not with synchronization
        while(true) {
            int work = 0;
            synchronized (this.work_queue) {
                work = getWork();
                if (work >= end){
                    break;
                }
            }
            if(isPrime(work)) {
                count();
            }
        }

        // Check the run time
        long endTime = System.currentTimeMillis();
        runTime = endTime - startTime;
    }

    // Getter - work from queue with synchronization
    public synchronized int getWork() {
        this.work_queue[0]++;
        return (this.work_queue[0]-1);
    }

    // Getter - runt time
    public long getRunTime() {
        return runTime;
    }

    // Getter - Prime number
    public long getPrimeNum() {
        return prime_num;
    }

    // Count the prime number with synchronization
    public synchronized void count() {
        prime_num++;
    }

    // Check whether the number is prime
    // If it is prime, then return True
    // Else return false
    private boolean isPrime(final 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;
    }
}
```

OUTPUT

1. pc_static.java

- Thread #1

```
oeComputing/Project1/src  master  java pc_static
Thread-0: 4269ms
Total Execution Time: 4271ms
Total number of prime number: 17984
```

- Thread #2

```
oeComputing/Project1/src  master ●  java pc_static
Thread-0: 1170ms
Thread-1: 3124ms
Total Execution Time: 3124ms
Total number of prime number: 17984
```

- Thread #4

```
oeComputing/Project1/src  master ●  java pc_static
Thread-0: 386ms
Thread-1: 944ms
Thread-2: 1458ms
Thread-3: 1948ms
Total Execution Time: 1949ms
Total number of prime number: 17984
```

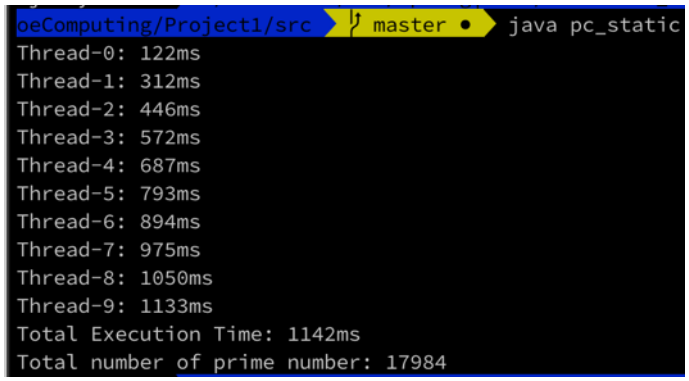
- Thread #6

```
oeComputing/Project1/src  master ●  java pc_static
Thread-0: 216ms
Thread-1: 553ms
Thread-2: 837ms
Thread-3: 1069ms
Thread-4: 1291ms
Thread-5: 1509ms
Total Execution Time: 1510ms
Total number of prime number: 17984
```

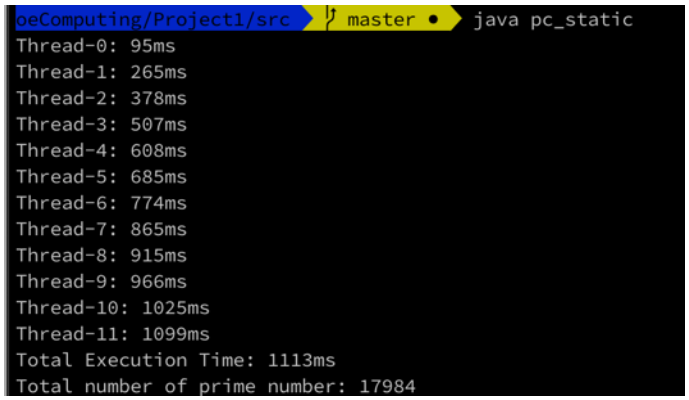
- Thread #8

```
oeComputing/Project1/src  master ●  java pc_static
Thread-0: 146ms
Thread-1: 369ms
Thread-2: 563ms
Thread-3: 740ms
Thread-4: 879ms
Thread-5: 1010ms
Thread-6: 1142ms
Thread-7: 1255ms
Total Execution Time: 1256ms
Total number of prime number: 17984
```

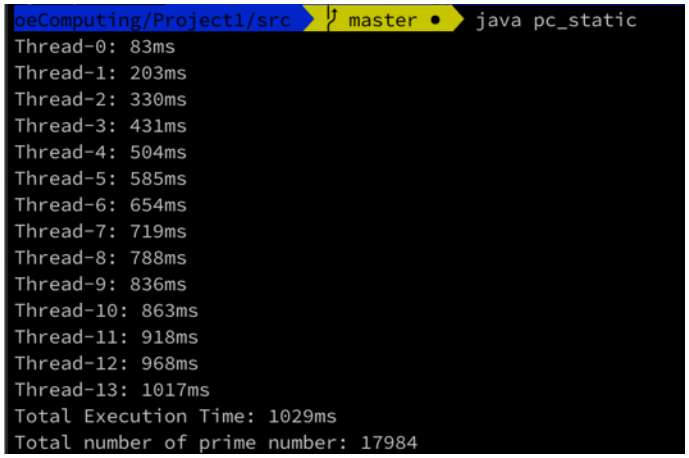
- Thread #10



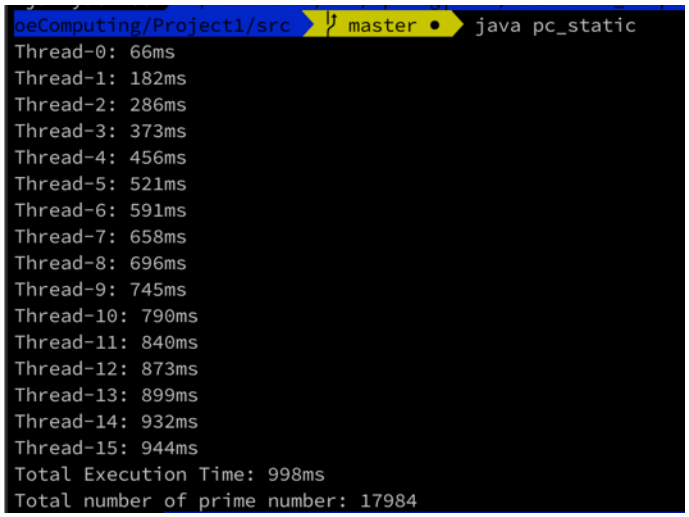
- Thread #12



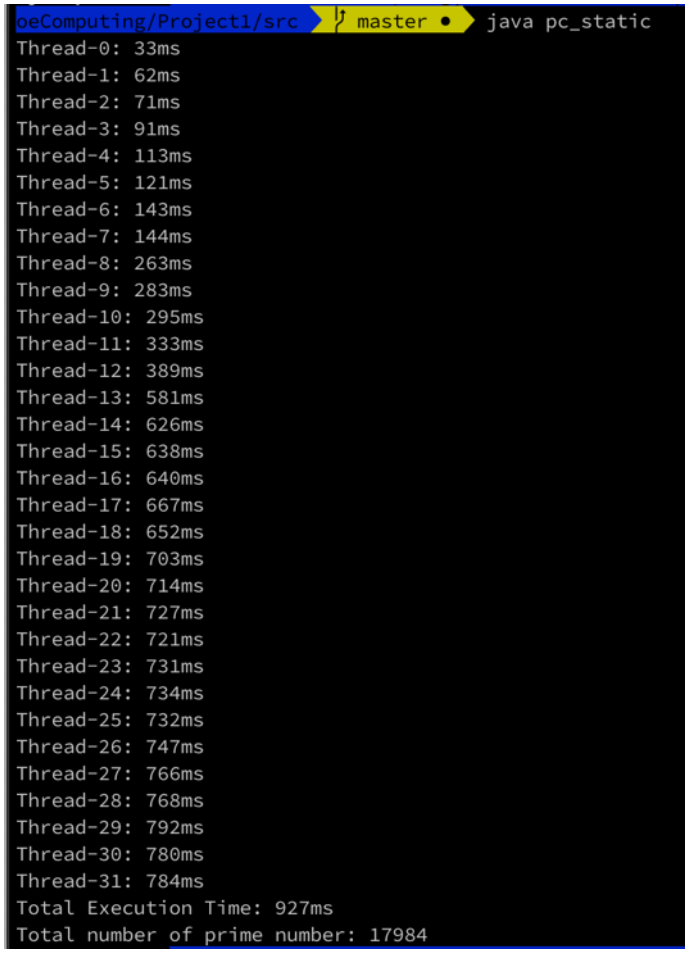
- Thread #14



- Thread #16



- Thread #32

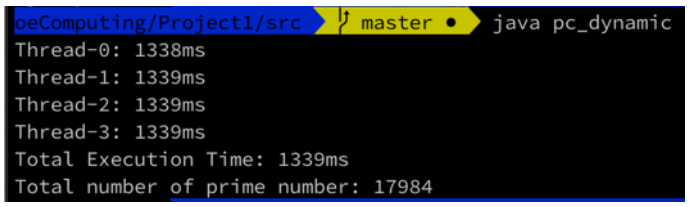


2. pc_dynamic.java

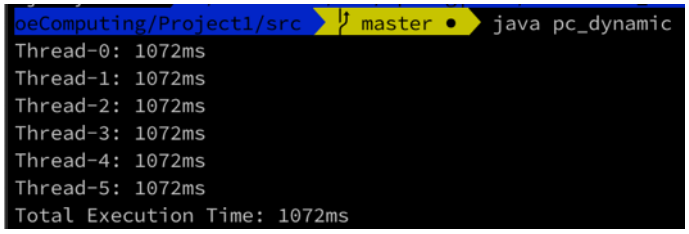
- Thread #1

- Thread #2

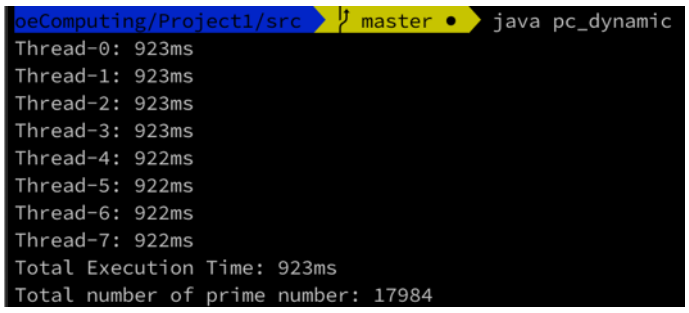
- Thread #4



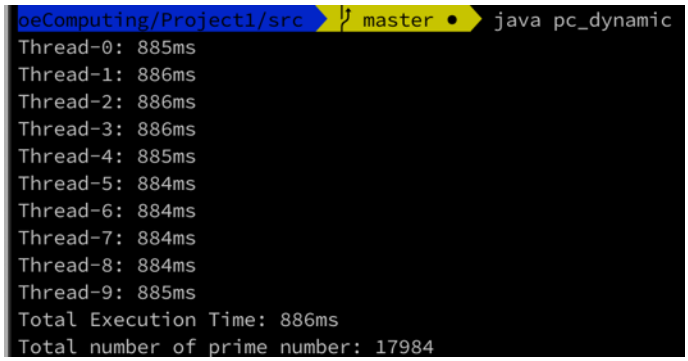
- Thread #6



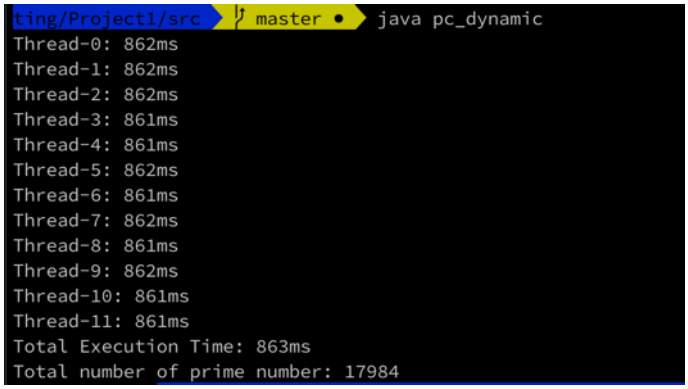
- Thread #8



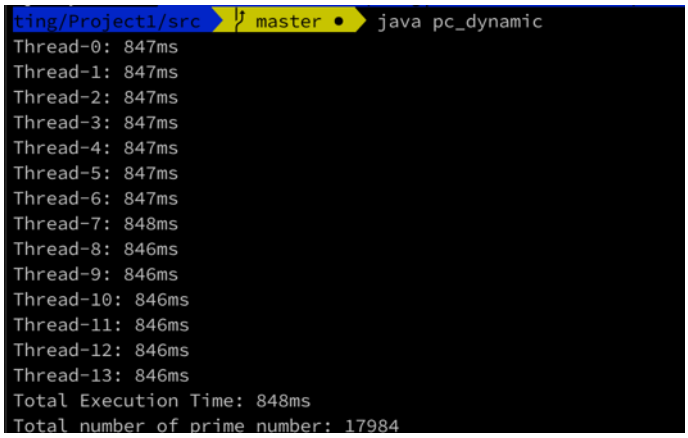
- Thread #10



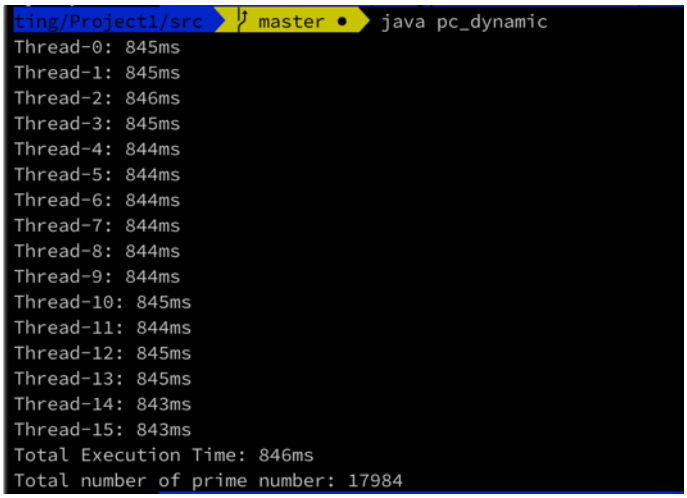
- Thread #12



- Thread #14



- Thread #16



- Thread #32