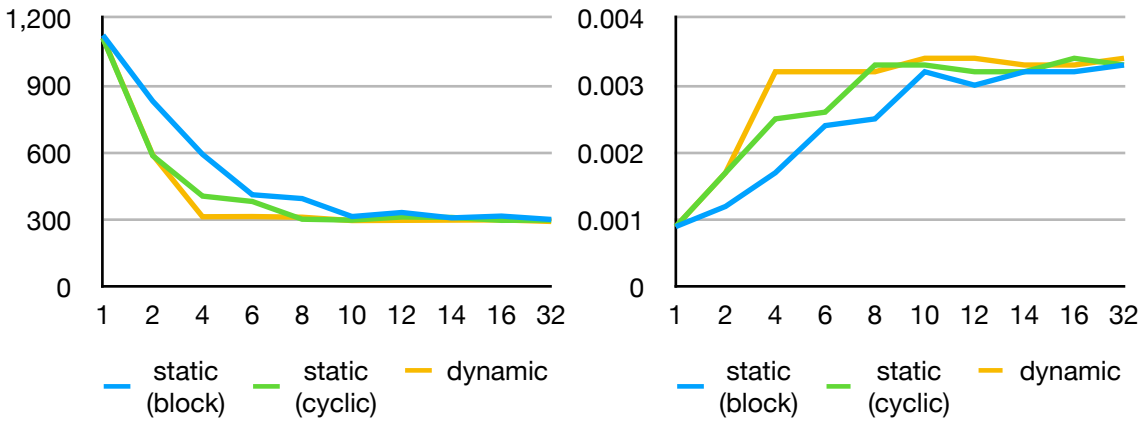
멀티코어컴퓨팅(Spring, 2023)
20171248 안재형

Multicore Project #1 - Problem01

Environment

Macbook Air(M1, 2020)
CPU : Apple M1 Chip(8-core CPU with 4 performance cores and 4 efficiency cores)
Memory : 8GB
OS : MacOS Ventura 13.2.1
IDE : IntelliJ

Execution Result



(unit : ms), task size of static(cyclic) and dynamic : 10

exec_time	1	2	4	6	8	10	12	14	16	32
static (block)	1123	829	592	412	395	315	333	309	317	302
static (cyclic)	1109	586	406	382	303	299	313	311	298	299
dynamic	1111	589	314	315	312	297	298	299	300	293

perfor-mance	1	2	4	6	8	10	12	14	16	32
static (block)	0.0009	0.0012	0.0017	0.0024	0.0025	0.0032	0.0030	0.0032	0.0032	0.0033
static (cyclic)	0.0009	0.0017	0.0025	0.0026	0.0033	0.0033	0.0032	0.0032	0.0034	0.0033
dynamic	0.0009	0.0017	0.0032	0.0032	0.0032	0.0034	0.0034	0.0033	0.0033	0.0034

Analysis

My computer's CPU - Apple M1 chip - has 4 performance cores and 4 efficient cores.

1~4 threads

It is expected that the performance will increase significantly, since the threads will be assigned to performance cores.

4~8 threads

It is expected that performance will increase slightly, since 4 threads will be assigned to performance cores and the others will be assigned to efficient cores.

8~ threads

Because there are only 8 physical processors, it is not expected to increase performance for more than 8 threads.

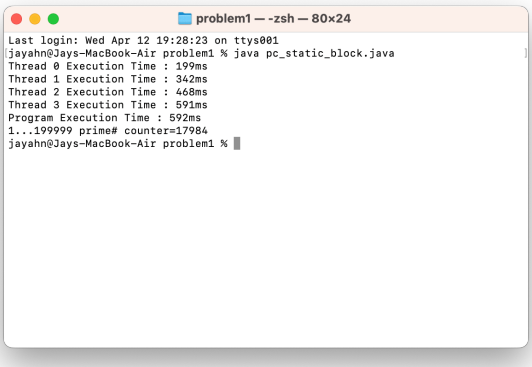
Even if I have more cpu cores, I don't expect that performance would increase infinitely because of the Amdahl's Law.

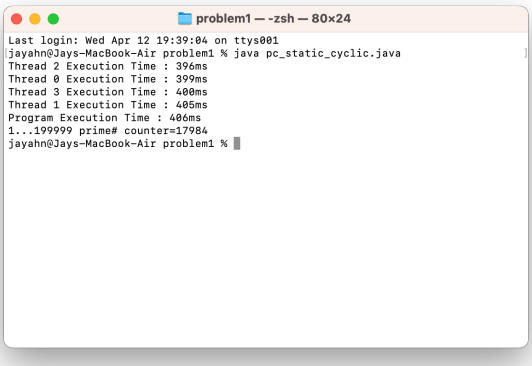
Compared to problem2, problem1 doesn't need a lot of synchronization. In my opinion, this is why the performance graph is shown almost linearly - the ideal line.

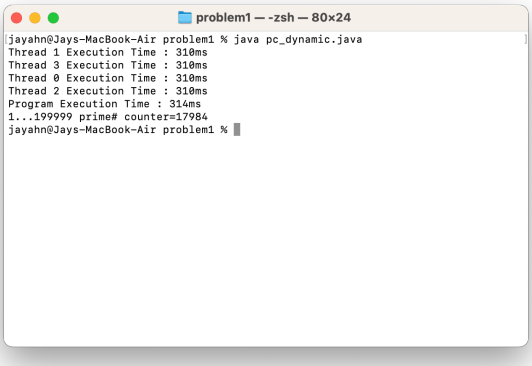
Execution

How to execute : java pc_static_block.java
java pc_static_cyclic.java
java pc_dynamic.java

Execution Results :







Code

pc_static_block.java

```
package problem1;

public class pc_static_block {
    private static final int NUM_START = 0;
    private static int NUM_END = 200000;
    private static int NUM_THREADS = 4;

    public static void main (String[] args) {
        if (args.length == 2) {
            NUM_THREADS = Integer.parseInt(args[0]);
            NUM_END = Integer.parseInt(args[1]);
        }
        pc_static_block_counter counter = new pc_static_block_counter();
        int i;
        long startTime = System.currentTimeMillis();

        // Thread creation
        pc_static_block_thread[] threads = new pc_static_block_thread[NUM_THREADS];
        for (i=0; i<NUM_THREADS; i++) {
            int thread_num_start = (int) (Math.ceil(NUM_START + ((NUM_END - NUM_START) * ((dou-
ble) i / NUM_THREADS))));
            int thread_num_end = (int) (Math.ceil(NUM_START + ((NUM_END - NUM_START) * ((double)
(i + 1) / NUM_THREADS))));
            threads[i] = new pc_static_block_thread(i, thread_num_start, thread_num_end,
counter);
            threads[i].start();
        }
        for (i=0; i<NUM_THREADS; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException ignored) {
                System.out.println("Thread joining failed.");
            }
        }
        long endTime = System.currentTimeMillis();
        long timeDiff = endTime - startTime;
        System.out.println("Program Execution Time : " + timeDiff + "ms");
        System.out.println("1..." + (NUM_END-1) + " prime# counter=" + counter.num_of_prime_num-
bers);
    }
}

class pc_static_block_thread extends Thread {
    int thread, num_start, num_end;
    pc_static_block_counter counter;
    public pc_static_block_thread(int thread, int num_start, int num_end, pc_static_block_counter
counter) {
        this.thread = thread;
        this.num_start = num_start;
        this.num_end = num_end;
        this.counter = counter;
    }

    @Override
    public void run() {
        long startTime = System.currentTimeMillis();
        for (int i=num_start; i < num_end; i++) {
            if (isPrime(i)) counter.addCount();;
        }
        long endTime = System.currentTimeMillis();
        long timeDiff = endTime - startTime;
        System.out.println("Thread " + thread + " Execution Time : " + timeDiff + "ms");
    }

    private static boolean isPrime(int x) {
```

```
        int i;
        if (x<=1) return false;
        for (i=2; i<x; i++) {
            if (x%i == 0) return false;
        }
        return true;
    }
}

class pc_static_block_counter {
    int num_of_prime_numbers = 0;
    synchronized void addCount() {
        this.num_of_prime_numbers += 1;
    }
}
```

pc_static_cyclic.java

```
package problem1;

public class pc_static_cyclic {
    private static final int NUM_START = 0;
    private static int NUM_END = 200000;
    private static int NUM_THREADS = 4;
    private static final int SIZE_OF_TASK = 10;

    public static void main (String[] args) {
        if (args.length == 2) {
            NUM_THREADS = Integer.parseInt(args[0]);
            NUM_END = Integer.parseInt(args[1]);
        }
        pc_static_cyclic_counter counter = new pc_static_cyclic_counter();
        int i;
        long startTime = System.currentTimeMillis();

        // Thread creation
        pc_static_cyclic_thread[] threads = new pc_static_cyclic_thread[NUM_THREADS];
        for (i=0; i<NUM_THREADS; i++) {
            threads[i] = new pc_static_cyclic_thread(i, NUM_THREADS, SIZE_OF_TASK, NUM_START,
NUM_END, counter);
            threads[i].start();
        }
        for (i=0; i<NUM_THREADS; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException ignored) {
                System.out.println("Thread joining failed.");
            }
        }
        long endTime = System.currentTimeMillis();
        long timeDiff = endTime - startTime;
        System.out.println("Program Execution Time : " + timeDiff + "ms");
        System.out.println("1..." + (NUM_END-1) + " prime# counter=" + counter.num_of_prime_num-
bers);
    }
}

class pc_static_cyclic_thread extends Thread {
    int thread, num_of_threads, size_of_task, num_start, num_end;
    pc_static_cyclic_counter counter;
    public pc_static_cyclic_thread(
            int thread,
            int num_of_threads,
            int size_of_task,
            int num_start,
            int num_end,
            pc_static_cyclic_counter counter
    ) {
        this.thread = thread;
```

```
        this.num_of_threads = num_of_threads;
        this.size_of_task = size_of_task;
        this.num_start = num_start;
        this.num_end = num_end;
        this.counter = counter;
    }

    @Override
    public void run() {
        long startTime = System.currentTimeMillis();
        for (int i=0; (i * size_of_task * num_of_threads) < num_end; i++) {
            for (int j=0; j < size_of_task; j++) {
                int number_to_check = (i * num_of_threads * size_of_task) + (size_of_task *
thread) + j;
                if (number_to_check < num_end) {
                    if (isPrime(number_to_check)) counter.addCount();
                } else {
                    break;
                }
            }
        }
        long endTime = System.currentTimeMillis();
        long timeDiff = endTime - startTime;
        System.out.println("Thread " + thread + " Execution Time : " + timeDiff + "ms");
    }

    private static boolean isPrime(int x) {
        int i;
        if (x<=1) return false;
        for (i=2; i<x; i++) {
            if (x%i == 0) return false;
        }
        return true;
    }
}

class pc_static_cyclic_counter {
    int num_of_prime_numbers = 0;
    synchronized void addCount() {
        this.num_of_prime_numbers += 1;
    }
}
```

pc_dynamic.java

```
package problem1;

import javax.swing.text.html.Option;
import java.util.OptionalInt;

public class pc_dynamic {
    private static final int NUM_START = 0;
    private static int NUM_END = 200000;
    private static int NUM_THREADS = 4;
    private static final int SIZE_OF_TASK = 10;

    public static void main (String[] args) {
        if (args.length == 2) {
            NUM_THREADS = Integer.parseInt(args[0]);
            NUM_END = Integer.parseInt(args[1]);
        }
        pc_dynamic_counter counter = new pc_dynamic_counter();
        int i;
        long startTime = System.currentTimeMillis();

        // Thread creation
        pc_dynamic_task_stack task_stack = new pc_dynamic_task_stack(NUM_START, NUM_END,
SIZE_OF_TASK);
        pc_dynamic_thread[] threads = new pc_dynamic_thread[NUM_THREADS];
```

```
        for (i=0; i<NUM_THREADS; i++) {
            threads[i] = new pc_dynamic_thread(i, NUM_END, SIZE_OF_TASK, task_stack, counter);
            threads[i].start();
        }
        for (i=0; i<NUM_THREADS; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException ignored) {
                System.out.println("Thread joining failed.");
            }
        }
        long endTime = System.currentTimeMillis();
        long timeDiff = endTime - startTime;
        System.out.println("Program Execution Time : " + timeDiff + "ms");
        System.out.println("1..." + (NUM_END-1) + " prime# counter=" + counter.num_of_prime_num-
bers);
    }
}

class pc_dynamic_thread extends Thread {
    final int thread, num_end, size_of_task;
    pc_dynamic_counter counter;
    pc_dynamic_task_stack task_stack;
    public pc_dynamic_thread(
            int thread,
            int num_end,
            int size_of_task,
            pc_dynamic_task_stack task_stack,
            pc_dynamic_counter counter
    ) {
        this.thread = thread;
        this.num_end = num_end;
        this.size_of_task = size_of_task;
        this.task_stack = task_stack;
        this.counter = counter;
    }

    @Override
    public void run() {
        long startTime = System.currentTimeMillis();
        while (true) {
            OptionalInt get_task = task_stack.getTask();
            if (get_task.isEmpty()) break;
            int task = get_task.getAsInt();
            for (int i = task; i < Math.min(task + size_of_task, num_end); i++) {
                if (isPrime(i)) counter.addCount();
            }
        }
        long endTime = System.currentTimeMillis();
        long timeDiff = endTime - startTime;
        System.out.println("Thread " + thread + " Execution Time : " + timeDiff + "ms");
    }

    private static boolean isPrime(int x) {
        int i;
        if (x<=1) return false;
        for (i=2; i<x; i++) {
            if (x%i == 0) return false;
        }
        return true;
    }
}

class pc_dynamic_counter {
    int num_of_prime_numbers = 0;
    synchronized void addCount() {
        this.num_of_prime_numbers += 1;
    }
}
```

```
class pc_dynamic_task_stack {
    final int num_start, num_end, size_of_task;
    int current_task;

    public pc_dynamic_task_stack(int num_start, int num_end, int size_of_task) {
        this.num_start = num_start;
        this.num_end = num_end;
        this.size_of_task = size_of_task;
        this.current_task = num_start;
    }

    synchronized OptionalInt getTask() {
        if (current_task >= num_end) return OptionalInt.empty();
        current_task += size_of_task;
        return OptionalInt.of(current_task - size_of_task);
    }
}
```