Load Balancing Problem 1 report

submission date: 20.04.2025

Student ID	50251141
Student name	Hamidreza Rahimian

Contents

1 Introduction 1

1.1 Motivation 1

1.2 Goal.....1

1.3 Approach 1

2 Methodology 2

3 Load balancing methods..... 4

3.1 Static load balancing with block method 4

3.2 Static load balancing with cyclic method.....7

3.3 Dynamic load balancing 8

4 Conclusion..... Error! Bookmark not defined.

1 Introduction

1.1 Motivation

This project is a project to see usage of Load Balancing in real java project, given is a java script called pc_serial.java that has count all the prime numbers from 1 to 200000.

1.2 Goal

The Goal of the project is to see and compare the runtime of different Load balancing methods and see which of was the best solution for this project, we can also check and observe with what number of Threads has the program its perfect and ideal runtime and what makes more sense.

There are 3 different Load balancing methods that we wanna use there:

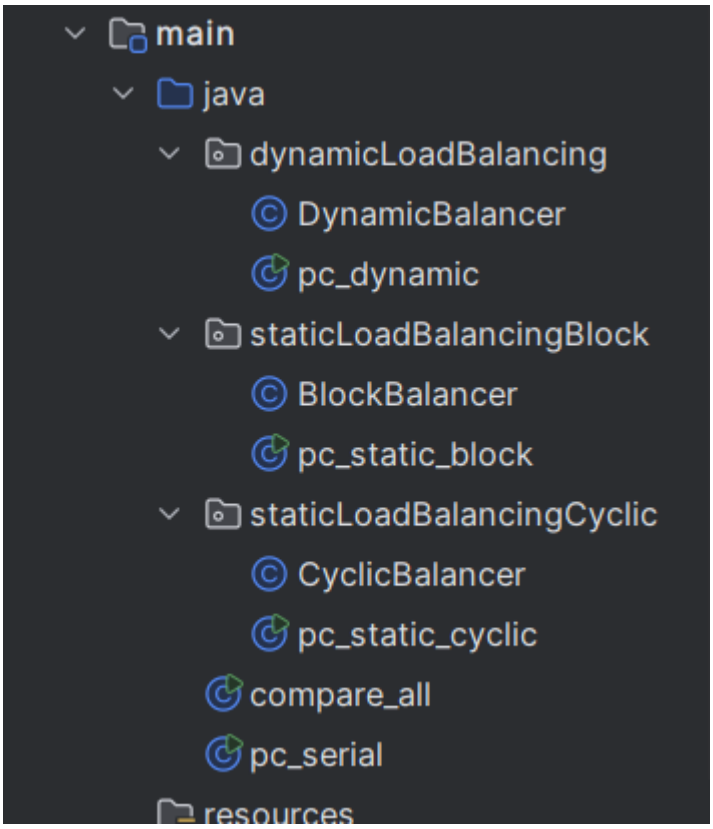
1. **Static load balancing based on block decomposition**
2. **Static load balancing based on cyclic decomposition**
3. **Dynamic load balancing**

1.3 Approach

Through this project, I aim make 3 java directories for 3 methods, each directory has a class for Load balancing Method in general and a java class that has the values and run the method for example for dynamic load balancing there are classes: **DynamicBalancer.java** with the general dynamic balancing function and **class pc_dynamic.java** with the actual code that runs the program. Basically class pc_dynamic.java called the methods and functions of class dynamicbalance.java and run the program with them. In the end I made another class which called **compare_all.java** that runs all of 3 methods and compare them to give a easy and understandable view of all the runtimes

2 Methodology

First, I started to make a reasonable structure for code files and directories. I made 3 directories and I put 2 code file in each directory which are representing the code that run the program and the code that include the Load balancing method. And I also added another code script next to pc_serial.java which has will be used in the end and run all the methods and compare them. The photo below shows how the java classes are organized here. I used IntelliJ as the programming surface cause I am familiar with implementation in that platform.



Then in the first step I copied the pc_serial code into the file and ran it. the Number of threads is 1. and as supposed it counts prime number between 1 and 200000.

```java
public class pc_serial {
  private static int NUM_END = 200000;   // default input
  private static int NUM_THREADS = 1;    // default number of threads
  public static void main (String[] args) {
    if (args.length==2) {
      NUM_THREADS = Integer.parseInt(args[0]);
      NUM_END = Integer.parseInt(args[1]);
    }
    int counter=0;
    int i;
    long startTime = System.currentTimeMillis();
    for (i=0;i<NUM_END;i++) {
      if (isPrime(i)) counter++;
    }
    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);
  }
  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;
  }
}
```

The result after running the code is: simply just number of counted prime numbers and the execution time:

```
2:33:28 PM: Executing ':pc_serial.main()'...

> Task :compileJava UP-TO-DATE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE

> Task :pc_serial.main()
Program Execution Time: 2312ms
1...199999 prime# counter=17984

Deprecated Gradle features were used in this build, making it incompatible with Gradle 9.0.

You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own scripts or p

For more on this, please refer to https://docs.gradle.org/8.5/userguide/command_line_interface.html#sec:command_line_warnings in t

BUILD SUCCESSFUL in 2s
2 actionable tasks: 1 executed, 1 up-to-date
2:33:31 PM: Execution finished ':pc_serial.main()'.
```

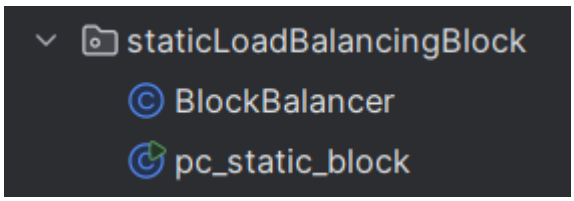
Runtime is 2312ms and it found 17984 prime number with NUM_THREADS = 1

so now we are going to start programming the Load Balancing Methods , for that we going to use same names as here in pc_serial like : NUM_THREADS , counter , count , isPrime and….

3 Load balancing methods

3.1 Static load balancing with block method

As I mentioned before I made 2 java classes in each directory, one of them has the general function (methods) for the type of load balancing that going to be used in that directory. For example, here there are 2 classes:



BlockBalancer has the code specifically designed for static load balancing with block method. This class going to be called and used when we run **pc_static_block.java** .

This is how pc_static_block.java looks like:

```java
package staticLoadBalancingBlock;

public class pc_static_block {  Alex *

    // Just your friendly launcher 🚀
    public static void main(String[] args) throws InterruptedException {  Alex *
        int NUM_THREADS = 4;         // Default number of threads (can be overridden via args)
        int NUM_END = 200000;        // Count primes up to this number

        // And now we call the general static block logic to do the real work
        BlockBalancer.run(NUM_THREADS, NUM_END);
    }
}
```

Basically, we just say how many threads and what is the biggest number (1 up to that number is our searching range). The we run the function BlockBlancer with the values we defined, when we run it. this will take us to the BlockBalancer class.

This method gets 2 int inputs from the other one which are number of threads and biggest number , then we made an array out of threads an array for counting the prime numbers and call it localCounts. Afterwards we want to check the start time and end time of program so that we can measure the runtime .

```
public class BlockBalancer {  1 usage   Alex *

    // Our main logic to count prime numbers using Block Static Load Balancing
    public static void run(int NUM_THREADS, int NUM_END) throws InterruptedException {  1 usage   Alex *
        int primeCount = 0;

        Thread[] threads = new Thread[NUM_THREADS];   // army of worker threads
        int[] localCounts = new int[NUM_THREADS];     // to collect how many primes each thread found

        long programStart = System.currentTimeMillis(); // stopwatch start 🚀
```

second thing we need isPrime function here as well so we just copy it from pc_serial . this will only help us to check if a number is prime or not.

```
    // The classic "is it prime?" function.
    static boolean isPrime(int num) {  1 usage   Alex *
        if (num <= 1) return false;
        for (int i = 2; i <= Math.sqrt(num); i++) {
            if (num % i == 0) return false;
        }
        return true;
    }
```

Now we need to define some for loops here so that they can actually start calculating and dividing the threads and tasks of each thread so than we can run multiple threads at the same time and make sure each thread has almost same tasks.

To do that we make a for-loop starting form 0 up to the Threads number (which is here 4) , inside loop we define integer threadIndex , which is literally the thread that we are currently working on in this loop . we have an array of threads and the thread we work on has a specific threadIndex . we start another timer to measure the time that this thread takes .afterwards we need to specify the size of each block. Each thread gonna count the prime numbers separately so we will define a variable local form type of integer which is the number of prime numbers in each block.

```
for (int i = 0; i < NUM_THREADS; i++) {
    final int threadIndex = i;

    threads[i] = new Thread(() -> {
        long threadStart = System.currentTimeMillis(); // stopwatch per thread
        int local = 0;

        // This is where the "block" idea kicks in. Each thread takes a big fat chunk.
        int blockSize = NUM_END / NUM_THREADS;
        int start = threadIndex * blockSize;
        int end = (threadIndex == NUM_THREADS - 1) ? NUM_END : start + blockSize;

        // Now let's go through the assigned block and count some primes like math detectives 🕵️
        for (int j = start; j < end; j++) {
            if (isPrime(j)) local++;
        }

        localCounts[threadIndex] = local;

        long threadEnd = System.currentTimeMillis();
        System.out.println("Thread " + threadIndex + " took " + (threadEnd - threadStart) + "ms");
    });

    threads[i].start(); // go go go little thread!
}
```

For each block the is the for-loop of isPrime , which count the prime numbers and add up the var local. In the end they will be save in array localCounts and index of each block will be the index of thread of that block.

In this point the for loop done what it supposedly should do , so we check the end time and print out the time that threads took , it should be a simple minus calculation.

Now that threads is defined so we can **start** the thread (still inside the loop because each time a different blocked will be called)

In the end we will wait for all threads to finish their task using **join**.

```
// Wait for all threads to finish their prime-counting business
for (Thread t : threads) t.join();
```

Now that all threads are finished we can sum up the number of prime number in each thread, for that we use

```
// Gather all the local prime counts from each thread
for (int count : localCounts) {
    primeCount += count;
}
```

In this point we basically did what we wanted to so we will use timer to get the current time and name it endtime and we count the time for start and end to measure the time that the whole time of application.

Then there is only one thing left and that's to print out the time , performance and the number of prime numbers.

```
// Print the juicy results
System.out.println("Total time: " + totalTime + "ms  using static block load balancing");
System.out.println("Performance: " + (1000.0 / totalTime) + "   using static block load balancing");
System.out.println("Total prime numbers: " + primeCount + "   using static block load balancing");
```

3.2 Static load balancing with cyclic method

This one has the same base as the Block method the only difference is that instead of going for big blocks like 50000 at once we go through the whole numbers (200000) in a loop which means for example P1 check 1 P2 checks 2 P3checks 3 P4 checks 4 and P1 checks 5 P2 checks 6 and….. , I believe this is a better way to do that here cause the bigger the numbers are more challenging is it to calculate and find the prime between them and with this load balancing method we divide the tasks better between processors.

```
for (int i = 0; i < NUM_THREADS; i++) {
    final int threadIndex = i;

    threads[i] = new Thread(() -> {
        long threadStart = System.currentTimeMillis(); // timer for this thread
        int local = 0;

        // Here's the CYCLIC trick: thread i works on i, i+NUM_THREADS, i+2*NUM_THREADS, etc.
        for (int j = threadIndex; j < NUM_END; j += NUM_THREADS) {
            if (isPrime(j)) local++;
        }

        localCounts[threadIndex] = local;

        long threadEnd = System.currentTimeMillis();
        System.out.println("Thread " + threadIndex + " took " + (threadEnd - threadStart) + "ms");
    });

    threads[i].start(); // thread goes brrrr
}
```

3.3 Dynamic load balancing

Here are all processor working together , and tasks will be given to the next free processor . there are some special things about dynamic load balancing for example instead of normal variable we use atomic ones.

```
int TASK_SIZE = 10;

AtomicInteger nextTaskStart = new AtomicInteger( initialValue: 0);
AtomicInteger primeCount = new AtomicInteger( initialValue: 0);
```

This will prevent race condition and errors , so no 2 or more methods or thread

s changes a value or variable , or work on them at the same time.

```java
for (int i = 0; i < NUM_THREADS; i++) {
    final int threadIndex = i;

    threads[i] = new Thread(() -> {
        long threadStart = System.currentTimeMillis();
        int localCount = 0;

        while (true) {
            int start = nextTaskStart.getAndAdd(TASK_SIZE);
            if (start >= NUM_END) break;

            for (int j = start; j < start + TASK_SIZE && j < NUM_END; j++) {
                if (isPrime(j)) localCount++;
            }
        }

        primeCount.addAndGet(localCount);
        long threadEnd = System.currentTimeMillis();
        System.out.println("Thread " + threadIndex + " took " + (threadEnd - threadStart) + "ms");
    });

    threads[i].start();
}
```

The other difference of dynamic with other is that , here we have a **TASK_SIZE** example 10 tasks , and **while** a thread is free a task will be given to them.

That's the biggest difference between dynamic and static load balancing.

4 System information

The system is powered by an Intel **Core i7-12700H processor**, featuring a hybrid architecture with **14 cores and 20 threads**. It operates at a base clock speed of **2.3 GHz** and can boost up to 4.7 GHz. Hyperthreading is enabled, ensuring efficient multitasking. The system runs on both **Windows 11** Pro and Linux Ubuntu 22.04, offering flexibility for different environments. For development purposes, it uses **OpenJDK 17** as the Java runtime.

5 Results and data comparation

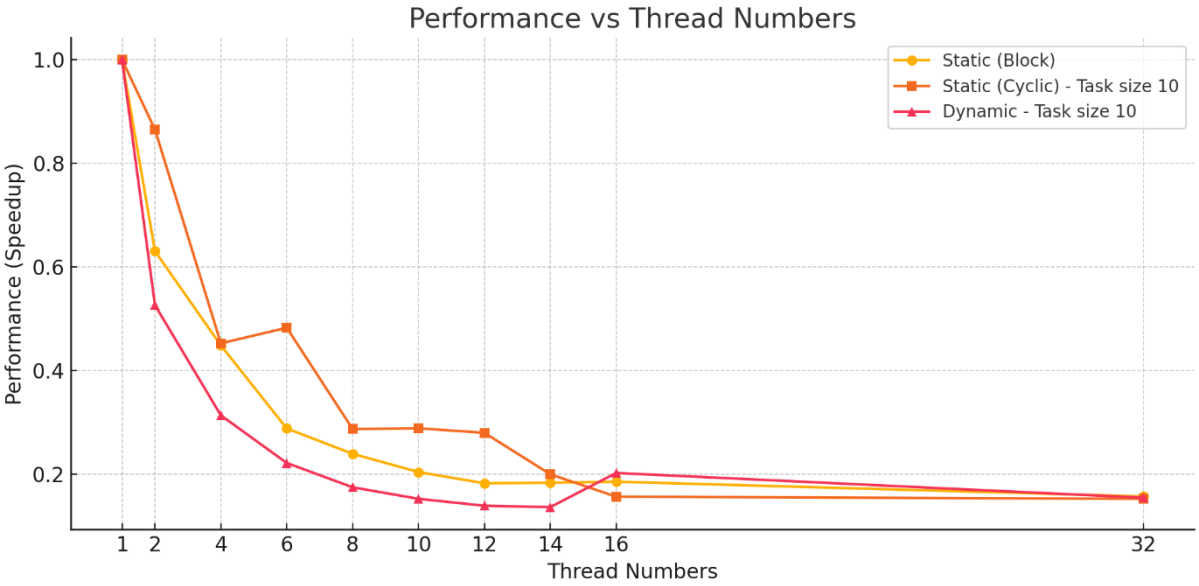
after executing the program with different Thread number and measuring it each time we will have such a table with all the execute times, the time is in milliseconds.

Thread count	1	2	4	6	8	10	12	14	16	32
Static (Block)	5225	3302	2345	1508	1251	1067	954	959	970	821
Static (cyclic) task size:10 numbers	2597	2246	1177	1253	745	749	726	518	406	395
Dynamic task size:10 numbers	2242	1184	704	498	392	342	312	306	454	347

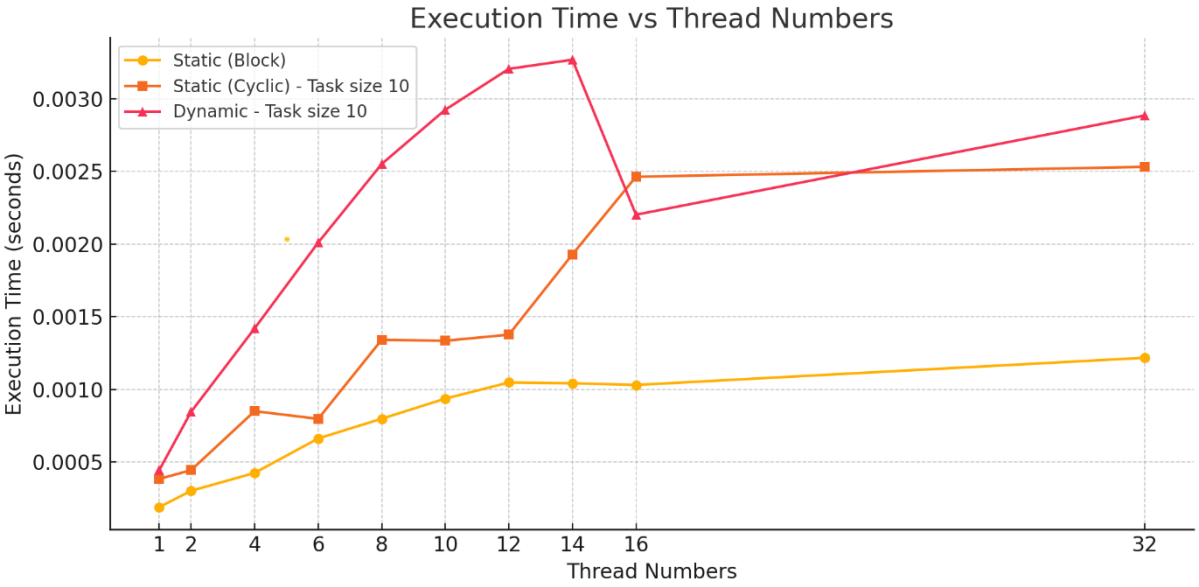
Performance is simply 1/exec time , attached blew is the table of performance of the all 3 load balancing .

Thread numbers	**1**	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**32**
Static (Block)	0.000191	0.000303	0.000426	0.000663	0.000799	0.000937	0.001048	0.001043	0.001031	0.001218
Static (Cyclic) - Task size 10	0.000385	0.000445	0.000851	0.000798	0.001342	0.001335	0.001377	0.001929	0.002463	0.002531
Dynamic - Task size 10	0.000445	0.000846	0.00142	0.00201	0.002551	0.002924	0.003205	0.003268	0.002202	0.002884

As we observe the execution time will clearly decrease in static block and dynamic , has better decreasing trend than static cyclic one , so probably here they are a better solution for solving this type of problem. The performance in dynamic and static, this was in the beginning clear however at some point the increasing trend slowed down.



The execution graph is exact opposite of performance , because the higher performance is means basically less exec time :D . as we see in execution graph wih increasing the threads the time was decreased very sharp in the begging however after some point it slowed down. Which means the effect of thread numbers is very big until some point after that the change is not that eye catching.



So after testing different load balancing strategies for counting prime numbers, we saw that the **dynamic method with small tasks (10 numbers)** clearly performed the best

as thread count increased. The **static cyclic** approach also scaled well, especially compared to **static block**, which also has a good trend growing but still not as good as other methods. It doesn't mean that static load balancing with block method was not good just means that the improvement and performance of cyclic or dynamic load balancing are way better than static load balancing with block method , so here they are good choices for solving this issue.