

**Load Balancing Problem 1 report**

submission date: 20.04.2025

Student ID 50251141

Student name Hamidreza Rahimian

Contents

[1 Introduction 1](#_Toc195538777)

[1.1 Motivation 1](#_Toc195538778)

[1.2 Goal 1](#_Toc195538779)

[1.3 Approach 1](#_Toc195538780)

[2 Methodology 2](#_Toc195538781)

[3 Load balancing methods 4](#_Toc195538782)

[3.1 Static load balancing with block method 4](#_Toc195538783)

[3.2 Static load balancing with cyclic method 5](#_Toc195538784)

[3.3 Dynamic load balancing 5](#_Toc195538785)

[4 Conclusion 6](#_Toc195538786)

# Introduction

## 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.

## 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**

## 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

# 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.

A screenshot of a computer

AI-generated content may be incorrect.

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.

A computer code with black text

AI-generated content may be incorrect.

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

A screenshot of a computer program

AI-generated content may be incorrect.

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….

# Load balancing methods

## 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:

A screenshot of a computer

AI-generated content may be incorrect.

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:

A screen shot of a computer program

AI-generated content may be incorrect.

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 .

A screenshot of a computer code

AI-generated content may be incorrect.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.

A computer screen shot of a program code

AI-generated content may be incorrect.

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.

A screen shot of a computer program

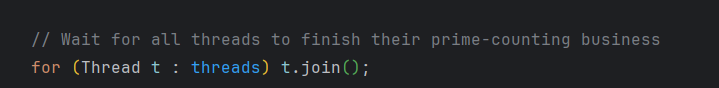
AI-generated content may be incorrect.

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**.



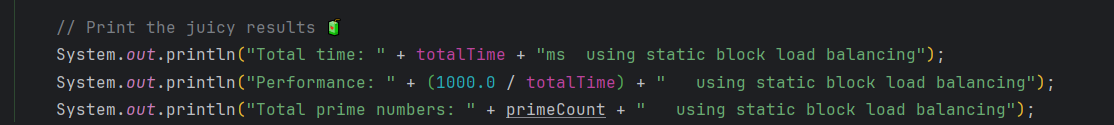
Now that all threads are finished we can sum up the number of prime number in each thread, for that we use

A screen shot of a computer code

AI-generated content may be incorrect.

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.



## 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.

A screen shot of a computer code

AI-generated content may be incorrect.

## 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.

A screen shot of a computer

AI-generated content may be incorrect.

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.

A computer screen shot of a program code

AI-generated content may be incorrect.

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.

# 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.

# 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) | 1055 | 1016 | 1005 | 1005 | 1079 | 1046 | 1268 | 1202 | 1284 | 1154 |
| 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 .

A grid of numbers and symbols

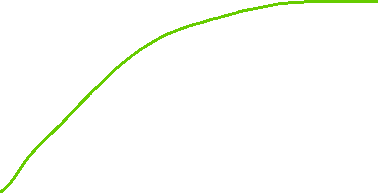
AI-generated content may be incorrect.

As we observe the execution time will clearly decrease in static cyclic and dynamic , however the difference in the static load balancing with block method is not that much ,its even kind of increased in some point that can be because of some random things like CPU usage in moment of execution or…

The performance in dynamic and static cyclic , this was in the beginning really clear however at some point the increasing trend slowed down.

A graph showing the difference between a number of people

AI-generated content may be incorrect.



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.

A graph with different colored lines

AI-generated content may be incorrect.

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 pretty well, especially compared to **static block**, which didn’t benefit much from more threads and sometimes even got worse.