CS424 – Introduction to Parallel Computing
Semester II, 2024-2025

(Group Project)

Prime Number Sequence Calculator

Prepared by:

Afnan Alsahli - 4252717

Jawaher Alluhaybi - 4251109

Bashayr Almutairi - 4257551

Prepared for:

Dr. Huda Abbas

April 24, 2025

Project Repository

Source Code + Output

HTTPS://DRIVE.GOOGLE.COM/DRIVE/FOLDERS/1RCT2Q43RQAOR0H1NYEKGAUGNT6ZA0XR1?USP=SHARING

Table of Contents

Problem Overview

Prime numbers, while simple to define, present a significant computational challenge when dealing with large ranges. A prime number is any natural number greater than 1 that has no divisors other than 1 and itself. These numbers play a critical role in various fields, including cryptography, mathematics, and computer science.

The objective of this project is to compute all prime numbers up to a given number N. While this may seem like a straightforward task, the complexity increases substantially as N grows. Performing this task sequentially can become inefficient and time-consuming, especially with larger input sizes.

To address this challenge, we explore the use of parallel computing. By dividing the workload among multiple threads or systems, we aim to improve performance and reduce the time required for computation.

A sequential implementation in Java processes tasks using a single thread

Two parallel implementations:

- One using MPI (Message Passing Interface) for distributed memory systems, where multiple machines communicate and coordinate the task
- Another using Java multithreading for shared memory systems, where multiple threads within the same machine work concurrently

Sequential Implementation (Java)

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class SequentialSolution {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String continueChoice;

        do {
            int N = -1;
            // Input validation loop
            while (true) {
                System.out.print("Enter the value of N (or type 'exit' to quit): ");
                String input = scanner.nextLine().trim();

                if (input.equalsIgnoreCase("exit")) {
                    System.out.println("See you next time...!");
                    scanner.close();
                    return; // Exit the program
                }
                // Handle different types of invalid inputs
                if (input.isEmpty()) {
                    System.out.println("You did not type anything! Please try again.");
                } else if (input.matches("-?\\d+\\.\\d+")) {
                    System.out.println("Decimals are not allowed! Please enter a whole
number.");
                } else if (input.matches("[a-zA-Z]+")) {
                    System.out.println("Letters are not valid input! Please enter numbers
only.");
                } else if (!input.matches("-?\\d+")) {
                    System.out.println("Symbols or mixed characters are not allowed! Please
enter a valid number.");
                } else {
                    N = Integer.parseInt(input);
                    // Check for negative or too-small numbers
                    if (N < 0) {
                        System.out.println("Negative numbers are not allowed! Please enter a
positive whole number.");
                    } else if (N < 2) {
                        System.out.println("There are no prime numbers less than 2! Try a
bigger number.");
                    } else {
                        break; // Valid number entered
                    }
                }
            }
            // Start timing using nanoseconds
            long startTime = System.nanoTime();
            List<Integer> primes = computePrimesUpToN(N); // Compute primes
            long endTime = System.nanoTime();    // End timing
            // Output results
            System.out.println("Prime numbers up to " + N + ":");
            System.out.println(primes);
            System.out.printf("Execution time: %.4f milliseconds%n", (endTime - startTime) /
1_000_000.0);
```

```java
// Ask user if they want to repeat
            while (true) {
                System.out.print("Try another number? (yes/no): ");
                continueChoice = scanner.nextLine().trim();

                if (continueChoice.equalsIgnoreCase("yes") ||
continueChoice.equalsIgnoreCase("no")) {
                    break; // Valid input
                } else {
                    System.out.println("Invalid input! Please type ('yes' or 'no'):");
                }
            }

        } while (!continueChoice.equalsIgnoreCase("no") &&
!continueChoice.equalsIgnoreCase("exit"));

        // Exit message
        System.out.println("Thank you for using The Program. Hope you come back soon...!");
        scanner.close();
    }

     // Computes all prime numbers from 2 up to N
    public static List<Integer> computePrimesUpToN(int N) {
        List<Integer> primeList = new ArrayList<>();
        for (int number = 2; number <= N; number++) {
            if (isPrime(number)) {
                primeList.add(number); // Add if prime
            }
        }
        return primeList;
    }

     // Checks whether a number is prime
    public static boolean isPrime(int num) {
        if (num <= 1) return false; // Not prime
        for (int i = 2; i <= Math.sqrt(num); i++) {
            if (num % i == 0) return false; // Not prime if divisible
        }
        return true; // Prime number
    }
}
```

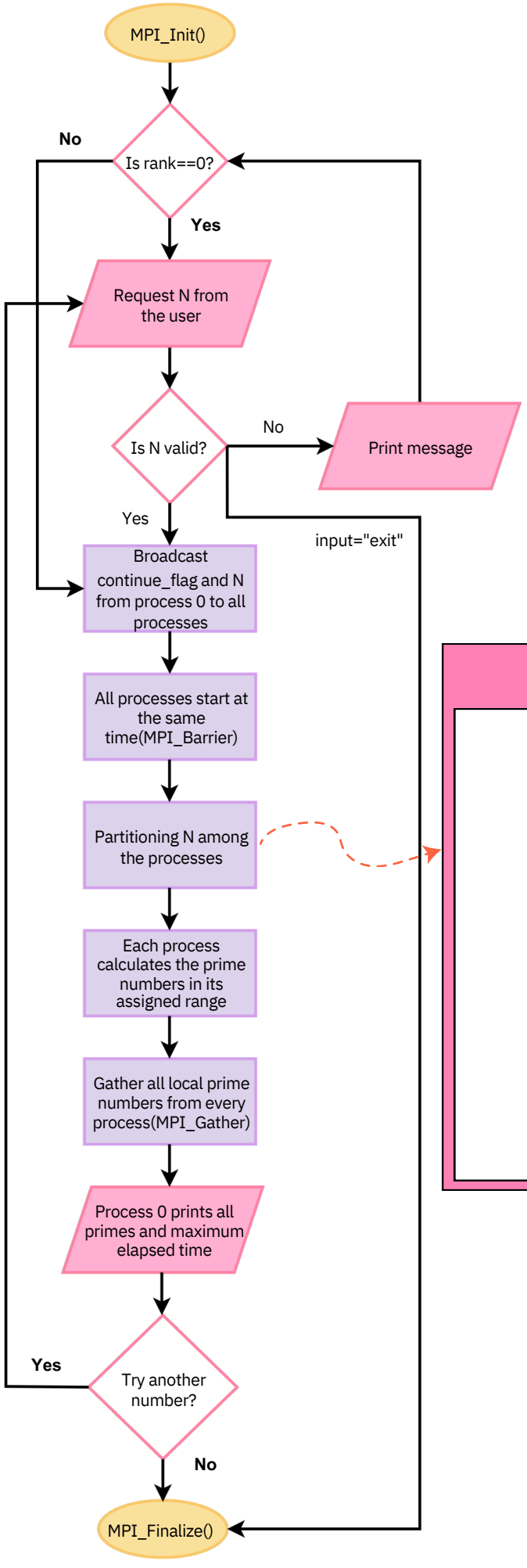
Sample Output (Java)

```
run:
Enter the value of N (or type 'exit' to quit):
You did not type anything! Please try again.
Enter the value of N (or type 'exit' to quit): 3.3
Decimals are not allowed! Please enter a whole number.
Enter the value of N (or type 'exit' to quit): HudaIsTheBest
Letters are not valid input! Please enter numbers only.
Enter the value of N (or type 'exit' to quit): #
Symbols or mixed characters are not allowed! Please enter a valid number.
Enter the value of N (or type 'exit' to quit): 0
There are no prime numbers less than 2! Try a bigger number.
Enter the value of N (or type 'exit' to quit): 1
There are no prime numbers less than 2! Try a bigger number.
Enter the value of N (or type 'exit' to quit): -3
Negative numbers are not allowed! Please enter a positive whole number.
Enter the value of N (or type 'exit' to quit): 10
Prime numbers up to 10:
[2, 3, 5, 7]
Execution time: 0.1074 milliseconds
Try another number? (yes/no): yes
Enter the value of N (or type 'exit' to quit): 100
Prime numbers up to 100:
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
Execution time: 0.0423 milliseconds
Try another number? (yes/no): yes
Enter the value of N (or type 'exit' to quit): exit
See you next time...!
```

```
Try another number? (yes/no): no
Thank you for using The Program. Hope you come back soon...!
```

#	Input	Explanation
1	empty input	The user pressed Enter without typing anything. The program prompts them to try again.
2	3.3	A decimal number was entered. The program only accepts whole numbers (integers).
3	HudaIsTheBest	Letters are not valid. The program requires numeric input.
4	#	A symbol (or mixed characters) was entered. Only numbers are allowed.
5	0	There are no prime numbers less than 2, so the input is invalid.
6	1	Similar to case 5: 1 is not considered a prime number.
7	-3	Negative numbers are not allowed. The program requires a positive whole number.
8	10	Valid small number; shows prime output for a short range.
9	100	Valid larger number; shows prime output for a bigger range.
10	exit	The user exits the program.

Parallel Algorithm Design (MPI)



Example of block data partitioning for input=10 and processes=3

1	2	3	4	5	6	7	8	9	10

process 0

range_start = 2 + 0 * 9 / 3 = 2 + 0 = 2
range_end = 2 + 1 * 9 / 3 - 1 = 2 + 3 - 1 = 4

process 1

range_start = 2 + 1 * 9 / 3 = 2 + 3 = 5
range_end = 2 + 2 * 9 / 3 - 1 = 2 + 6 - 1 = 7

process 2

range_start = 2 + 2 * 9 / 3 = 2 + 6 = 8
range_end = 2 + 3 * 9 / 3 - 1 = 2 + 9 - 1 = 10

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <ctype.h>

// Compares two strings
int my_strcasecmp(const char* s1, const char* s2) {
    while (*s1 && *s2) {
        char c1 = tolower((unsigned char)*s1);
        char c2 = tolower((unsigned char)*s2);
        if (c1 != c2)
            return c1 - c2;
        s1++;
        s2++;
    }
    return tolower((unsigned char)*s1) - tolower((unsigned char)*s2);
}

// Checks if the string contains digits only
int is_digits_only(const char* str) {
    if (str == NULL || *str == '\0') return 0;
    for (int i = 0; str[i]; i++) {
        if (!isdigit((unsigned char)str[i])) return 0;
    }
    return 1;
}

// Checks if the string contains letters only
int is_letters_only(const char* str) {
    if (str == NULL || *str == '\0') return 0;
    for (int i = 0; str[i]; i++) {
        if (!isalpha((unsigned char)str[i])) return 0;
    }
    return 1;
}

// Checks if the string represents a valid decimal number
int is_valid_decimal(const char* str) {
    int dot_count = 0;
    int digit_count = 0;
    for (int i = 0; str[i]; i++) {
        if (str[i] == '.') {
            dot_count++;
            if (i == 0 || str[i + 1] == '\0') return 0; // Starts or ends with dot
        }
        else if (isdigit((unsigned char)str[i])) {
            digit_count++;
        }
        else {
            return 0; // Invalid character
        }
    }
    return (dot_count == 1 && digit_count > 0);
}
```

Parallel Implementation (MPI) *cont.*

```
// Checks whether a number is prime.
int isPrime(int num) {
    if (num <= 1) return 0;
    for (int i = 2; i <= (int)sqrt(num); i++) {
        if (num % i == 0) return 0;
    }
    return 1;
}

int main(int argc, char** argv) {
    int rank, size;
    int N;
    char continue_flag[10] = "yes";

    double start_time, end_time, local_elapsed, max_elapsed;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    do {
        if (rank == 0) {
            if (my_strcasecmp(continue_flag, "yes") == 0) {
                char input[100];
                // Input validation loop
                while (1) {
                    printf("Enter the value of N (or type 'exit' to quit): ");
                    fflush(stdout);
                    fgets(input, sizeof(input), stdin);
                    input[strcspn(input, "\n")] = '\0';

                    if (my_strcasecmp(input, "exit") == 0) {
                        snprintf(continue_flag, sizeof(continue_flag), "exit");
                        break;
                    }
                    // Handle different types of invalid inputs
                    if (strlen(input) == 0) {
                        printf("You did not type anything! Please try again.\n");
                    }
                    else if (is_valid_decimal(input)) {
                        printf("Decimals are not allowed! Please enter a whole number.\n");
                    }
                    else if (is_letters_only(input)) {
                        printf("Letters are not valid input! Please enter numbers only.\n");
                    }
                    // Handle the case for negative numbers
                    else if (input[0] == '-' && is_digits_only(input + 1)) {
                        printf("Negative numbers are not allowed! Please enter a positive
whole number.\n");
                    }
                    else if (is_digits_only(input)) {
                        N = atoi(input);
```

```
                        // Handle the case for 0 or 1
                        if (N < 2) {
                            printf("There are no prime numbers less than 2! Try a bigger
number.\n");
                        }
                        else {
                            break; // Valid number entered
                        }
                    }
                    else {
                        printf("Symbols or mixed characters are not allowed! Please enter a
valid number.\n");
                    }
                }
            }
        }

        MPI_Bcast(continue_flag, 10, MPI_CHAR, 0, MPI_COMM_WORLD);
        if (my_strcasecmp(continue_flag, "exit") == 0) {
            if (rank == 0) printf("See you next time...!\n");
            MPI_Finalize();
            return 0; // Exit the program
        }

        // Broadcast the value of N from process 0 to all other processes
        MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);// Synchronize all processes
        start_time = MPI_Wtime();

        // Partitioning N among the processes
        int range_start = 2 + rank * (N - 1) / size;
        int range_end = 2 + (rank + 1) * (N - 1) / size - 1;
        if (range_end > N) range_end = N;

        int* local_primes = (int*)malloc((N - 1) * sizeof(int));
        int local_count = 0;

        // Each process calculates the prime numbers in its assigned range
        for (int i = range_start; i <= range_end; i++) {
            if (isPrime(i)) {
                local_primes[local_count++] = i;
            }
        }

        end_time = MPI_Wtime();
        local_elapsed = end_time - start_time;

        int* recv_counts = NULL;
        if (rank == 0) {
            recv_counts = (int*)malloc(size * sizeof(int));
        }
        // Gather all local prime numbers
        MPI_Gather(&local_count, 1, MPI_INT, recv_counts, 1, MPI_INT, 0, MPI_COMM_WORLD);

        int* displs = NULL;
        int total_primes = 0;
        int* all_primes = NULL;
```

```
        if (rank == 0) {
            displs = (int*)malloc(size * sizeof(int));
            displs[0] = 0;
            for (int i = 1; i < size; i++) {
                displs[i] = displs[i - 1] + recv_counts[i - 1];
            }
            total_primes = displs[size - 1] + recv_counts[size - 1];
            all_primes = (int*)malloc(total_primes * sizeof(int));
        }
        //Gather local primes in unequal quantities
        MPI_Gatherv(local_primes, local_count, MPI_INT,
            all_primes, recv_counts, displs, MPI_INT,
            0, MPI_COMM_WORLD);

        MPI_Reduce(&local_elapsed, &max_elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

        //Print max elapsed time and the list of prime numbers up to N
        if (rank == 0) {
            printf("\nPrime numbers up to %d:\n[", N);
            for (int i = 0; i < total_primes; i++) {
                printf("%d", all_primes[i]);
                if (i < total_primes - 1) printf(", ");
            }
            printf("]\n");
            printf("Max Elapsed time = %f seconds\n", max_elapsed);

            free(recv_counts);
            free(displs);
            free(all_primes);

            // Ask user if they want to try again
            printf("Try another number? (yes/no): ");
            while (1) {
                fflush(stdout);
                fgets(continue_flag, sizeof(continue_flag), stdin);
                continue_flag[strcspn(continue_flag, "\n")] = '\0';

                if (my_strcasecmp(continue_flag, "yes") == 0 ||
                    my_strcasecmp(continue_flag, "no") == 0) {
                    break;
                }
                else {
                    printf("Invalid input! Please type ('yes' or 'no'): ");
                }
            }
        }

        MPI_Bcast(continue_flag, 10, MPI_CHAR, 0, MPI_COMM_WORLD);
        free(local_primes);

    } while (my_strcasecmp(continue_flag, "no") != 0 && my_strcasecmp(continue_flag, "exit")
!= 0);

    // Exit message
    if (rank == 0) {
        printf("Thank you for using The Program. Hope you come back soon...!\n");
    }

    MPI_Finalize();
    return 0;
}
```

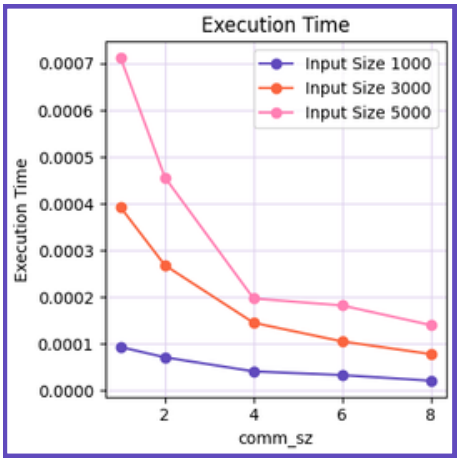
Performance Evaluation (MPI)

[In MPI, execution time is measured in seconds]

Execution Time			
comm_sz	Input Size		
	1000	3000	5000
1	0.000093	0.000392	0.000712
2	0.000071	0.000268	0.000456
4	0.000041	0.000145	0.000197
6	0.000033	0.000105	0.000182
8	0.000021	0.000078	0.000140

Execution time = max elapsed time across all processors

Lower is better



Overhead			
comm_sz	Input Size		
	1000	3000	5000
1	0.0	0.0	0.0
2	0.000024	0.000072	0.000100
4	0.000018	0.000047	0.000019
6	0.000017	0.000040	0.000063
8	0.000009	0.000029	0.000051

Overhead = $T_{parallel} - T_{serial}/P$

Extra time from parallelization

is caused by:

Communication

Load Imbalance

Hardware Limitations

Algorithm Design

Synchronization Delays

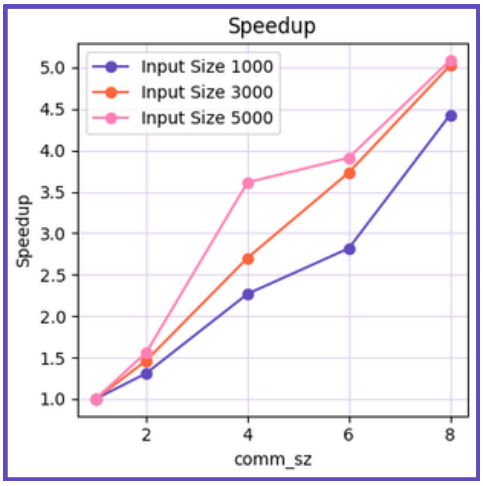
Performance Evaluation (MPI) *cont.*

Speedup			
comm_sz	Input Size		
	1000	3000	5000
1	1.0	1.0	1.0
2	1.310	1.463	1.561
4	2.268	2.703	3.614
6	2.818	3.733	3.912
8	4.429	5.026	5.086

Speedup = T_{serial}/$T_{parallel}$

How much faster than serial?

Higher is better

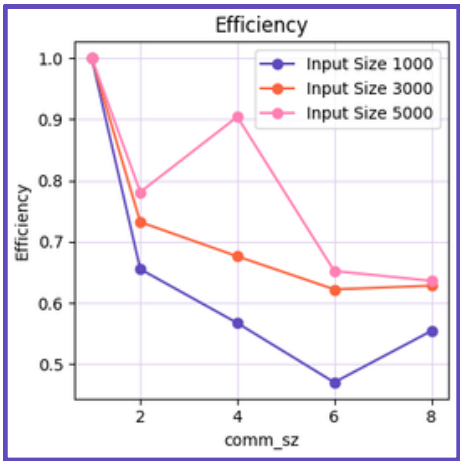


Efficiency			
comm_sz	Input Size		
	1000	3000	5000
1	1.0	1.0	1.0
2	0.655	0.732	0.781
4	0.567	0.676	0.904
6	0.470	0.622	0.652
8	0.554	0.628	0.636

Efficiency = **Speedup**/**P**

How well resources are used?

Perfect if =1 (100% utilization) ✔



Parallel Algorithm Design (Multithreading)

```
START
DO
  DISPLAY "Enter the value of N (or type 'exit' to quit):"
  INPUT user_input

  IF user_input IS "exit" THEN
    DISPLAY "See you next time...!"
    EXIT PROGRAM
  END IF

  IF user_input IS INVALID (empty, decimal, letters, symbols) THEN
    DISPLAY specific error message based on the type of invalid input
    CONTINUE TO NEXT LOOP
  END IF

  PARSE user_input TO INTEGER N

  IF N < 2 THEN
    DISPLAY "There are no prime numbers less than 2! Try a bigger number."
    CONTINUE TO NEXT LOOP
  END IF

  thread_count = number of threads
  primes = EMPTY LIST
  segment_size = N / thread_count

  START TIMER

  FOR i FROM 0 TO thread_count - 1 DO
    start = i * segment_size + 2
    IF i == thread_count - 1 THEN
      end = N
    ELSE
      end = (i + 1) * segment_size + 1
    END IF

    CREATE THREAD i TO:
      FOR j FROM start TO end DO
        IF j IS PRIME THEN
          ADD j TO primes WITH SYNCHRONIZATION
        END IF
      END FOR
  END FOR

  WAIT FOR ALL THREADS TO COMPLETE

  STOP TIMER

  SORT primes
  DISPLAY "Prime numbers up to N: " + primes
  DISPLAY "Execution Time (parallel): " + elapsed_time_in_milliseconds

  DO
    DISPLAY "Try another number? (yes/no):"
    INPUT continue_choice
  WHILE continue_choice IS NOT "yes" AND NOT "no"

WHILE continue_choice IS "yes"

DISPLAY "Thank you for using The Program. Hope you come back soon...!"
END
```

Example of the data partitioning process
n = 100 , Thread_count = 8

Thread Rank	start = i * m + 2	end = (i+1) * m +1	Range
0	0*12 + 2 =2	(0+1)*12+1 = 13	{2,13}
1	1*12 +2 = 14	(1+1)*12 +1 = 25	{14,25}
2	2*12 +2 = 26	(2+1)*12 +1= 37	{26,37}
3	3*12 + 2= 38	(3+1)*12 +1= 49	{38,49}
4	4*12 + 2 = 50	(4+1) * 12 +1 = 61	{50,61}
5	5*12 +2 = 62	(5+1) *12 +1 = 73	{62,73}
6	6*12 + 2 = 74	(6+1) * 12 +1 = 85	{74,85}
7	86	100	{86,100}

Parallel Implementation (Multithreading)

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Scanner;

public class ParallelSolution {
    public static void main(String[] args) throws InterruptedException {
        Scanner scanner = new Scanner(System.in);
        String continueChoice;

        do {
            int N = -1;
            // Input validation loop
            while (true) {
                System.out.print("Enter the value of N (or type 'exit' to quit): ");
                String input = scanner.nextLine().trim();

                if (input.equalsIgnoreCase("exit")) {
                    System.out.println("See you next time...!");
                    scanner.close();
                    return; // Exit the program
                }
                // Handle different types of invalid inputs
                if (input.isEmpty()) {
                    System.out.println("You did not type anything! Please try again.");
                } else if (input.matches("-?\\d+\\.\\d+")) {
                    System.out.println("Decimals are not allowed! Please enter a whole
number.");
                } else if (input.matches("[a-zA-Z]+")) {
                    System.out.println("Letters are not valid input! Please enter numbers
only.");
                } else if (!input.matches("-?\\d+")) {
                    System.out.println("Symbols or mixed characters are not allowed! Please
enter a valid number.");
                } else {
                    N = Integer.parseInt(input);
                    // Check for negative or too-small numbers
                    if (N < 0) {
                        System.out.println("Negative numbers are not allowed! Please enter a
positive whole number.");
                    } else if (N < 2) {
                        System.out.println("There are no prime numbers less than 2! Try a
bigger number.");
                    } else {
                        break; // Valid number entered
                    }
                }
            }
            // Get the number of threads
            int threadCount = Runtime.getRuntime().availableProcessors();
            // Shared list for collecting prime numbers
            List<Integer> primes = new ArrayList<>();
            // Start timing using nanoseconds
            long startTime = System.nanoTime();
            // Create and start threads
            Thread[] threads = new Thread[threadCount];
            int segmentSize = N / threadCount;

            for (int i = 0; i < threadCount; i++) {
                int start = i * segmentSize + 2;
                int end = (i == threadCount - 1) ? N : (i + 1) * segmentSize + 1;
                threads[i] = new Thread(new PrimeTask(start, end, primes));
                threads[i].start();
            }
```

```
            // Wait for all threads to complete
            for (Thread thread : threads) {
                thread.join();
            }
            long endTime = System.nanoTime();

            // Sort and print results
            Collections.sort(primes);
            System.out.println("Prime numbers up to " + N + ":");
            System.out.println(primes);
            System.out.printf("Execution time (parallel): %.4f milliseconds%n", (endTime -
startTime) / 1_000_000.0);

            // Ask user if they want to repeat
            while (true) {
                System.out.print("Try another number? (yes/no): ");
                continueChoice = scanner.nextLine().trim();

                if (continueChoice.equalsIgnoreCase("yes") ||
continueChoice.equalsIgnoreCase("no")) {
                    break; // Valid input
                } else {
                    System.out.println("Invalid input! Please type ('yes' or 'no'):");
                }
            }

        } while (!continueChoice.equalsIgnoreCase("no") &&
!continueChoice.equalsIgnoreCase("exit"));

        // Exit message
        System.out.println("Thank you for using The Program. Hope you come back soon...!");
        scanner.close();
    }
}

// Computes prime numbers in a specific range (used by each thread)
class PrimeTask implements Runnable {
    private final int start, end;
    private final List<Integer> sharedPrimes;

    public PrimeTask(int start, int end, List<Integer> sharedPrimes) {
        this.start = start;
        this.end = end;
        this.sharedPrimes = sharedPrimes;
    }
    @Override
    public void run() {
        for (int i = start; i <= end; i++) {
            if (isPrime(i)) {
                synchronized (sharedPrimes) {
                    sharedPrimes.add(i); // Safely add to shared list
                }
            }
        }
    }
    // Checks whether a number is prime
    private static boolean isPrime(int num) {
        if (num <= 1) return false;
        for (int i = 2; i <= Math.sqrt(num); i++) {
            if (num % i == 0) return false;
        }
        return true;
    }
}
```

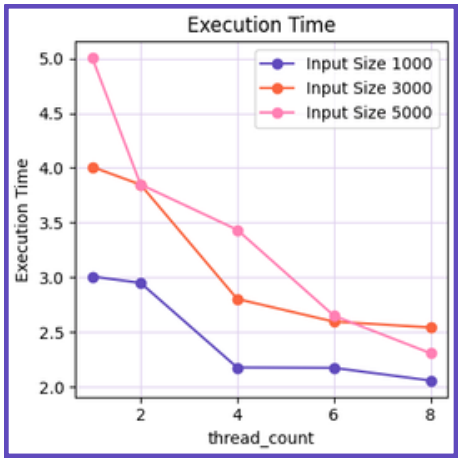
Performance Evaluation (Multithreading)

[In Multithreading, execution time is measured in Milliseconds]

Execution Time			
thread_count	Input Size		
	1000	3000	5000
1	3.008	4.009	5.005
2	2.950	3.847	3.849
4	2.177	2.803	3.435
6	2.174	2.596	2.651
8	2.058	2.543	2.308

Execution time = max elapsed time across all threads

again Lower is better



Overhead			
thread_count	Input Size		
	1000	3000	5000
1	0.0	0.0	0.0
2	1.446	1.842	1.346
4	1.425	1.800	2.183
6	1.672	1.928	1.816
8	1.682	2.041	1.682

Overhead = $T_{parallel}$ - T_{serial}/P

Extra time from parallelization

is caused by:

Contention

Thread Spawning

Scheduling Latency

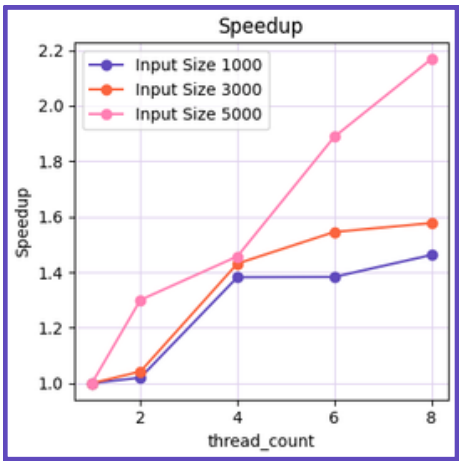
Synchronization Delays

Speedup			
thread_count	Input Size		
	1000	3000	5000
1	1.0	1.0	1.0
2	1.020	1.042	1.300
4	1.382	1.431	1.457
6	1.383	1.545	1.888
8	1.462	1.577	2.168

Speedup = $T_{serial}/T_{parallel}$

How much faster than serial?

again Higher is better



Efficiency			
thread_count	Input Size		
	1000	3000	5000
1	1.0	1.0	1.0
2	0.510	0.521	0.650
4	0.345	0.358	0.364
6	0.231	0.258	0.315
8	0.183	0.197	0.271

Efficiency = Speedup/P

How well resources are used?

Perfect if =1 (100% utilization)