Project 1 - Distributed Operating Systems

Durga Abhiram Gorle (3050881) - du.gorle@ufl.edu
Durga Sritha Dongla (54220803)- durgasrit.dongla@ufl.edu
GitHub URL: https://github.com/abhiramgorle/Class_project_1dosp

Problem Statement:

An interesting problem in arithmetic with deep implications for elliptic curve theory is finding perfect squares that are sums of consecutive squares. A classic example is the well-known Pythagorean identity:

$$3^2 + 4^2 = 5^2$$

This shows that the sum of the squares of 3 and 4 is itself a square. Another, more interesting, example is Lucas' Square Pyramid:

$$1^2 + 2^2 + \ldots + 24^2 = 70^2$$

In both examples, the sums of squares of consecutive integers form the square of another integer.

The goal of this project is to use Pony and the actor model to build a solution to this problem that performs efficiently on multi-core machines.

Requirements:

The input to your program (provided via command line, e.g., Lukas consists of two numbers: `N` and `k`.

The overall goal is to find all sequences of `k` consecutive numbers (starting at 1 or higher and up to `N`), such that the sum of squares forms a perfect square (i.e., the square of an integer).

Output : For each solution, print the first number in the sequence on independent lines.

Example 1: lukas 3 2

Output: 3

This indicates that a sequence of length 2 with a starting point between 1 and 3 contains the solution 3, 4 since:

$$3^2 + 4^2 = 5^2$$

Example 2: lukas 40 24

Output: 1

Indicates that a sequence of length 24 with a starting point between 1 and 40 contains the solution 1, 2,…, 24 since:

$$1^2 + 2^2 + \ldots + 24^2 = 70^2$$

Actor Model:

In this project, you are required to exclusively use the actor model in Pony. Projects that do not utilize multiple actors or use any other form of parallelism will not receive credit.

You may employ a model similar to the one discussed in class for summing large numbers. Specifically:

- Define worker actors that are assigned a range of sub-problems to solve.
- Use a boss actor to manage all tasks and handle job assignments to the workers.

README File:

In the `README` file, you must include the following:

- Work Unit Size :The size of the work unit you determined to achieve the best performance in your implementation. Explain how you arrived at this decision. The work unit size refers to the number of sub-problems a worker actor receives in a single request from the boss.
- Program Results: Report the output for running the program with the following command: lukas 1000000 4
- Running Time: Include the running time for the above command as reported by `time`: time lukas 1000000 4
- The ratio of CPU time to REAL TIME will help evaluate how effectively your program used the available cores. If the ratio is close to 1, it suggests minimal parallelism, which will result in point deductions
- Largest Problem Solved: Report the largest problem your program was able to solve successfully.

Code Variables and Functions Explained:

1. Main Actor

The Main actor is the entry point of the program, which initializes the solution by processing command-line arguments and invoking the WorkDistribution actor. If the input is invalid, it prints a usage message.

- **Input**: Command-line arguments N and k, where N is the upper bound of numbers and k is the number of consecutive numbers whose squares need to be summed.
- **Output**: If valid input is given, the WorkDistribution actor is created to handle the processing. Otherwise, it outputs a message explaining the correct input format.

2. WorkDistribution Actor

This actor is responsible for distributing the computational tasks among multiple worker actors, Answerfinder. It splits the problem into smaller **chunks** and assigns them to workers for parallel computation.

- **Attributes**:
    - _size_of_chunk: Determines the range of numbers that each worker processes. It is computed using the ceiling of the square root of N for better load balancing.
    - _workers: An array that holds instances of Answerfinder, responsible for calculating whether a sum of consecutive squares is a perfect square.
    - _total_number: The total number of tasks to be processed.
- **Workflow**:
    - The number of workers (num_ofSummers) is calculated by dividing N into chunks.
    - Workers are created and assigned ranges of numbers (start and end values) to process.
    - Each worker is given a range of numbers, and they calculate the sum of squares for sequences of length k within this range.

3. Math Class

This class provides utility functions for mathematical operations needed in the program.

- **square_root**: Implements a method to compute the square root of a number using the **Babylonian method** (also known as Heron's method). It returns the square root as a floating-point number.
- **ceil_function**: Returns the ceiling of a floating-point number, ensuring that any fractional part rounds up to the nearest integer.

4. Answerfinder Actor

This actor performs the actual computation for each range of numbers assigned by the WorkDistribution actor. It calculates the sum of squares for sequences of length k and checks if the sum is a perfect square.

- **Attributes**:
    - _work_queue: A queue that stores the squares of consecutive numbers being processed.
    - _sumans: Holds the sum of squares for the current sequence of length k.
    - _index: Keeps track of the starting index of the current sequence.
    - _checker: An instance of CheckSquareorNot, responsible for determining whether the sum is a perfect square.
- **Behavior**:
    - **chunk_calc**: For each number in the range, it calculates the square, adds it to the sum, and checks if the sum of the current sequence is a perfect square.
    - **add_square**: Updates the sum of squares by adding the square of the current number and removing the oldest number from the sum (if the sequence length exceeds k). Once the sum is calculated for k numbers, it calls the check_perfect_square method.

5. CheckSquareorNot Actor

This actor checks whether a given number is a perfect square and, if it is, prints the index of the sequence.

- **Attributes**:
    - _results: Stores the results of whether the sums calculated are perfect squares.
- **Behavior**:
    - **check_perfect_square**: Receives a sum and determines if it is a perfect square. If the sum is a perfect square, it prints the starting index of the sequence.
    - **is_perfect_square**: Implements a **binary search** to check if a number is a perfect square. It finds the middle point, squares it, and compares it with the input number. If it matches, the number is a perfect square.

Results:

```
(base) durgasrithadongla@Durgas-Laptop my_pony_project % /usr/bin/time ./my_pony_project lucas 1000000 4
        0.04 real         0.26 user         0.01 sys
(base) durgasrithadongla@Durgas-Laptop my_pony_project % /usr/bin/time ./my_pony_project lucas 100000000 4
        2.19 real        16.37 user         0.29 sys
(base) durgasrithadongla@Durgas-Laptop my_pony_project % /usr/bin/time ./my_pony_project lucas 100000000 4
        2.16 real        16.55 user         0.25 sys
(base) durgasrithadongla@Durgas-Laptop my_pony_project %
```

Cores:

No. of Cores used = User + System / Real

= 16.55 + 0.25 / 2.16

= 7.778