Distributed Operating Systems Project 1

Author: Sitanshu Sukhmandir Lamba, UFID: 5190-8991
Author: Darshan Dilip Kakwani, UFID: 5311-7117

Problem Definition:

Given an input N and k the program has to calculate all such numbers where $1^2 + 2^2 + 3^2 + \ldots + k^2$ = a perfect square. We will run this for k such consecutive numbers starting from 1 and going upto N. For example if N = 40 and k = 24 then one of the outputs will be 1 as $1^2 + 2^2 + 3^2 + 4^2 + 5^2 + \ldots + 24^2 = 70^2$. So we need to output the starting number of the series which gives a perfect square.

Languages Used:

We have used F# and Akka with .Net in order to run this code as F# is a functional programming language which helps in running processes in parallel on multicore machines. Akka here provides actors to F# which are agents that help in parallel execution and more utilisation of CPU time rather than real time.

The project is in the proj1.fsx and needs to be run on the command line for N = 1000000 and k = 4 using the command:
“dotnet fsi --langversion:preview proj1.fsx 1000000 4”

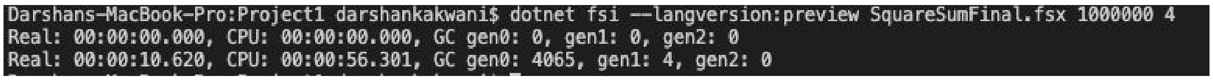
Algorithm/Approach used:

There is a boss actor which is called when the input is provided through the command line. This actor helps in providing work to its workers i.e other actors. After the input is received the input is divided into 10 parts and sent to each worker. For our code and our machines the most optimum solution was for 10/15 actors for us. We arrived at this number using trial and error. The sum of squares in each actor is calculated using a sliding window protocol which takes O(N) time to calculate the squares of numbers from 1 to N. Then each calculated square is passed to a square root function that calculates if the number is a perfect square using binary search in O(log N) time.

Inputs and Outputs:

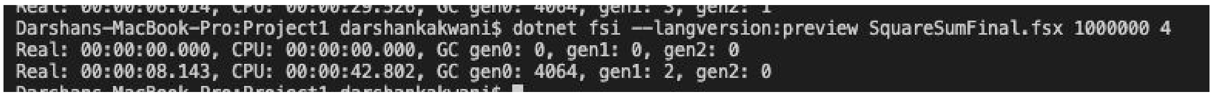
The following were the inputs and outputs for inputs with different sizes and different numbers of actors performed on a machine with 4 cores. The CPU to Real TIme ratio on an average is **5.00.** In order to check the CPU and Real time we used #time “on” in our code. For our work unit we decided that dividing the input into 10 parts and giving an equal chunk to each worker as a working unit.

Input: n = 1000000 (10^6), k = 4
Number of actors: 15
CPU Time: 56.301 seconds
Real Time: 10.620 seconds
Ratio: 5.30



```
Darshans-MacBook-Pro:Project1 darshankakwani$ dotnet fsi --langversion:preview SquareSumFinal.fsx 1000000 4
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
Real: 00:00:10.620, CPU: 00:00:56.301, GC gen0: 4065, gen1: 4, gen2: 0
```

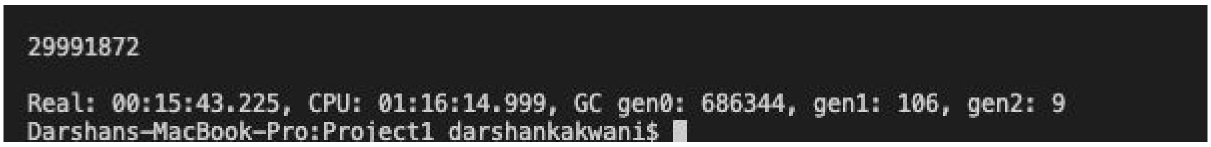
Input: n = 1000000 (10^6), k = 4
Number of Actors: 22
CPU Time: 42.802 seconds
Real Time: 8.143 seconds
Ratio: 5.25



```
Real: 00:00:06.014, CPU: 00:00:29.520, GC gen0: 4064, gen1: 3, gen2: 1
Darshans-MacBook-Pro:Project1 darshankakwani$ dotnet fsi --langversion:preview SquareSumFinal.fsx 1000000 4
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
Real: 00:00:08.143, CPU: 00:00:42.802, GC gen0: 4064, gen1: 2, gen2: 0
Darshans-MacBook-Pro:Project1 darshankakwani$
```

Largest Problem we managed to run:

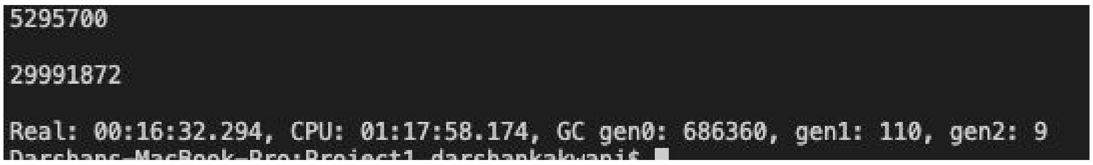
Input: 100000000 (10^8), k = 24
Number of Actors: 10
CPU Time: 1 hour 16 minutes 14.999 seconds
Real Time: 15 minutes 43.225 seconds
Ratio: 4.85



```
29991872

Real: 00:15:43.225, CPU: 01:16:14.999, GC gen0: 686344, gen1: 106, gen2: 9
Darshans-MacBook-Pro:Project1 darshankakwani$
```

Input: 100000000 (10^8), k = 24
Number of Actors: 16
CPU Time: 1 hour 17 minutes 58.174 seconds
Real Time: 16 minutes 32.294 seconds
Ratio: 4.71



```
5295700

29991872

Real: 00:16:32.294, CPU: 01:17:58.174, GC gen0: 686360, gen1: 110, gen2: 9
Darshans-MacBook-Pro:Project1 darshankakwani$
```

Conclusion:

Running tasks in parallel helps in increasing CPU time and reducing the real time in which a problem can be solved.