COP5612 – Fall 2020

Alin Dobra

September 10, 2020

- **Due Date:** September 21, Midnight
- One submission per group
- Submit using eLearning
- **What to include:**
  - `README` file including group members, other requirements specified below
  - `Lastnames.zip` the code for the project

1 Problem definition

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

$$3^2 + 4^2 = 5^2 \tag{1}$$

that reveals that the sum of squares of $3, 4$ is itself a square. A more interesting example is Lucas' *Square Pyramid*:

$$1^2 + 2^2 + ... + 24^2 = 70^2 \tag{2}$$

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

The goal of this first project is to use F# and the actor model to build a good solution to this problem that runs well on multi-core machines.

2 Requirements

Input: The input provided (as command line to your program, e.g. `my_app`) will be two numbers: N and k. The overall goal of your program is to find all k consecutive numbers starting at 1 and up to N, such that the sum of squares is itself a perfect square (square of an integer).

Output: Print, on independent lines, the first number in the sequence for each solution.

Example 1:

```
dotnet fsi proj1.fsx 3 2
3
```

indicates that sequences of length 2 with start point between 1 and 3 contain 3,4 as a solution since $3^2 + 4^2 = 5^2$.

Example 1:

```
dotnet fsi proj1.fsx 40 24
1
```

indicates that sequences of length 24 with start point between 1 and 40 contain 1,2,...,24 as a solution since $1^2 + 2^2 + ... + 24^2 = 70^2$.

Actor modeling: In this project you have to use exclusively the actor facility in F# (**projects that do not use multiple actors or use any other form of parallelism will receive no credit**). A model similar to the one indicated in class for the problem of adding up a lot of numbers can be used here, in particular define *worker* actors that are given a range of problems to solve and a *boss* that keeps track of all the problems and perform the job assignment.

README file In the README file you have to include the following material:

- Size of the *work unit* that you determined results in best performance for your implementation and an explanation on how you determined it. Size of the work unit refers to the number of sub-problems that a worker gets in a single request from the boss.
- The result of running your program for
  `dotnet fsi proj1.fsx 1000000 4`
- The running time for the above as reported by time for the above, i.e. run `time scala project1.scala 1000000 4` and report the time. The ratio of CPU time to REAL TIME tells you how many cores were effectively used in the computation. If your are close to 1 you have almost no parallelism (points will be subtracted).
- The largest problem you managed to solve.

3 BONUS – 15%

Use remote actors and run you program on 2+ machines. Use your solution to solve a really large instance such as: `dotnet fsi proj1.fsx 100000000 20`. To get the bonus points you have to give a demo to the instructor and explain your solution.

COP5615 – Fall 2020
Project 2 – Gossip Simulator

Alin Dobra

September 29, 2020

- **Due Date:** October 12, Midnight
- One submission per group
- Submit using eLearning/Canvas
- **What to include:**
  - `README` file including group members, other requirements specified below
  - `project2.tgz` or `project2.zip` the code for the project
  - `project2-bonus.tgz/zip` the code for the bonus part, if any

1 Problem definition

As described in class Gossip type algorithms can be used both for group communication and for aggregate computation. The goal of this project is to determine the convergence of such algorithms through a simulator based on actors written in F#. Since actors in F# are fully asynchronous, the particular type of Gossip implemented is the so called *Asynchronous Gossip*.

Gossip Algorithm for information propagation The Gossip algorithm involves the following:

- **Starting:** A participant(actor) it told/sent a roumor(fact) by the main process
- **Step:** Each actor selects a random *neighbor* and tells it the rumor
- **Termination:** Each actor keeps track of rumors and how many times it has heard the rumor. It stops transmitting once it has heard the rumor 10 times (10 is arbitrary, you can select other values).

Push-Sum algorithm for sum computation

- **State:** Each actor A_i maintains two quantities: s and w. Initially, $s = x_i = i$ (that is actor number i has value i, play with other distribution if you so desire) and $w = 1$
- **Starting:** Ask one of the actors to start from the main process.
- **Receive:** Messages sent and received are pairs of the form (s, w). Upon receive, an actor should add received pair to its own corresponding values. Upon receive, each actor selects a random neighboor and sends it a message.
- **Send:** When sending a message to another actor, half of s and w is kept by the sending actor and half is placed in the message.
- **Sum estimate:** At any given moment of time, the sum estimate is $\frac{s}{w}$ where s and w are the current values of an actor.
- **Termination:** If an actors ratio $\frac{s}{w}$ did not change more than 10^{-10} in 3 consecutive rounds the actor terminates. **WARNING: the values s and w independently never converge, only the ratio does.**

Topologies The actual network topology plays a critical role in the dissemination speed of Gossip protocols. As part of this project you have to experiment with various topologies. The topology determines who is considered a neighboor in the above algorithms.

- **Full Network** Every actor is a neighbor of all other actors. That is, every actor can talk directly to any other actor.
- **2D Grid:** Actors form a 2D grid. The actors can only talk to the grid neighbors.
- **Line:** Actors are arranged in a line. Each actor has only 2 neighbors (one left and one right, unless you are the first or last actor).
- **Imperfect 2D Grid:** Grid arrangement but one random other neighbor is selected from the list of all actors (4+1 neighbors).

2 Requirements

Input: The input provided (as command line to your `project2`) will be of the form:

```
project2 numNodes topology algorithm
```

Where `numNodes` is the number of actors involved (for 2D based topologies you can round up until you get a square), `topology` is one of `full, 2D, line, imp2D`, `algorithm` is one of `gossip, push-sum`.

Output: Print the amount of time it took to achieve convergence of the algorithm. Please measure the time using

```
... build topology
val b = System.currentTimeMillis;
..... start protocol
println(b-System.currentTimeMillis)
```

Actor modeling: In this project you have to use exclusively the actor facility in F# (**projects that do not use multiple actors or use any other form of parallelism will receive no credit**).

README file In the README file you have to include the following material:

- Team members
- What is working
- What is the largest network you managed to deal with for each type of topology and algorithm

Report.pdf For each type of topology and algorithm, draw the dependency of convergence time as a function of the size of the network. You can overlap different topologies on the same graph, i.e. you can draw 4 curves, one for each topology and produce only 2 graphs for the two algorithms. Write about any interesting finding of your experiments in the report as well and mention the team members.

You can produce Report.pdf in any way you like, for example using spreadsheet software. You might have to use logarithmic scales to have a meaningful plot.

3 BONUS

In the above assignment, there is no failure at all. For a 30% bonus, implement node and failure models (a node dies, a connection dies temporarily or permanently). Write a Report-bonus.pdf to explain your findings (how you tested, what experiments you performed, what you observed) and submit `project2-bonus.tgz/zip` with your code. To get the bonus you must implement at least one failure model controlled by a parameter and draw plots that involve the parameter. At least one interesting observation has to be made based on these plots.

COP5615 – Fall 2020

Project 3 – Pastry

Alin Dobra

October 14, 2020

- Due Date: Check Canvas
- One submission per group
- Submit using eLearning
- What to include:
  - README file including group members, other requirements specified below
  - project3.tgz/project3.zip the code for the project
  - project3-bonus.tgz/project3-bonus.zip the code for the bonus part, if any

1 Problem definition

We talked extensively in class about the overlay networks and how they can be used to provide services. The goal of this project is to implement in F# using the actor model the Pastry protocol and a simple object access service to prove its usefulness. The specification of the Pastry protocol can be found in the paper Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems.by A. Rowstron and P. Druschel. You can find the paper at http://rowstron.azurewebsites.net/PAST/pastry.pdf. The paper above, in Section 2.3 contains a specification of the Pastry API and of the API to be implemented by the application.

2 Requirements

You must implement the network join and routing as described in the Pastry paper and encode the simple application that associates a key (same as the ids used in pastry) with a string. You can change the message type sent and the specific activity as long as you implement it using a similar API to the one described in the paper.

Input: The input provided (as command line to yourproject3) will be of the form:

project3 numNodes numRequests

Where numNodesis the number of peers to be created in the peer to peer system and numRequests the number of requests each peer has to make. When all peers performed that many requests, the program can exit. Each peer should send a request/second.

Output: Print the average number of hops (node connections) that have to be traversed to deliver a message.

Actor modeling: In this project you have to use exclusively the actor model in F# (projects that do not use multiple actors or use any other form of parallelism will receive no credit). You should have one actor foreach of the peers modeled.

README file In the README file you have to include the following material:

- Team members
- What is working
- What is the largest network you managed to deal with

3 BONUS

In the above assignment, there is no failure at all. For a 20% bonus, implement node and failure models (a node dies, a connection dies temporarily or permanently). Write a report describing how you tested that the systems is resilient and your findings.

COP5612 – Fall 2020

Alin Dobra

Project 4 – Part 1

In this project, you have to implement a Twitter Clone and a client tester/simulator.

As of now, Tweeter does not seem to support a WebSocket API. As part I of this project, you need to build an engine that (in part II) will be paired up with WebSockets to provide full functionality. Specific things you have to do are:

- Implement a Twitter like engine with the following functionality:
    - Register account
    - Send tweet. Tweets can have hashtags (e.g. #COP5615isgreat) and mentions (@bestuser)
    - Subscribe to user's tweets
    - Re-tweets (so that your subscribers get an interesting tweet you got by other means)
    - Allow querying tweets subscribed to, tweets with specific hashtags, tweets in which the user is mentioned (my mentions)
    - If the user is connected, deliver the above types of tweets live (without querying)
- Implement a tester/simulator to test the above
    - Simulate as many users as you can
    - Simulate periods of live connection and disconnection for users
    - Simulate a Zipf distribution on the number of subscribers. For accounts with a lot of subscribers, increase the number of tweets. Make some of these messages re-tweets
- Other considerations:
    - The client part (send/receive tweets) and the engine (distribute tweets) have to be in separate processes. Preferably, you use multiple independent client processes that simulate thousands of clients and a single engine process
    - You need to measure various aspects of your simulator and report performance
    - More detail in lecture as the project progresses.

You need to submit your code, instructions how to run it and a report with performance numbers.

COP5612 – Fall 2020

Alin Dobra

Project 4 – Part 2

Use WebSharper web framework to implement a WebSocket interface to your part I implementation. That means that, even though the F# implementation (Part I) you could use AKKA messaging to allow client-server implementation, you now need to design and use a proper WebSocket interface. Specifically:

1. You need to design a JSON based API that represents all messages and their replies (including errors)
2. You need to re-write parts of your engine using WebSharper to implement the WebSocket interface
3. You need to re-write parts of your client to use WebSockets.

BONUS (30 points)

Implement a public key based authentication method for your interface. Specifically,

1. A user, upon registration, provides a public key (can be RSA-2048 or a 256 bit ElipticCurve)
2. When the user re-connects via WebSockets, it first authenticates using a challenge based algorithm
    1. The engine sends a 256 bit challenge
    2. The client forms a message containing the challenge, the current time (UNIX time in seconds) and digitally signs it.
    3. The engine sends a confirmation or an error
    4. The engine is not allowed to reuse the challenge and it must only cache it for 1 second. If the protocol does not finish within 1s, it must be tried again
3. The user establishes a secret key with the engine (using Diffie-Helman protocol) and HMAC signs every message sent
    1. The HMAC is computed over the serialized JSON content.

Please upload a demo video (~5 minutes), a youtube link within the README or an mp4 file to explain the following:

1. How to run your code? Create an account, search, …
2. Show your implementation of (i) a JSON based API that represents all messages and their replies (including errors) (ii) implementation of the WebSocket interface (iii) client to use WebSockets. Show each part in your video. You can highlight/point to your code in the video, and explain in README.