Kahn

Nizar Ghandri, Megi Dervishi, Christophe Saad

May 15, 2020

1 Remarks

- addition of functions
- difference between network2window and the rest of khan implemnetation
- explain this with the Example.ml

2 Khan implementations

2.1 Network2window

explain scheme

2.2 Network

2.3 Sequential

The implentation of the sequencial part is manly taken from the paper A poor man's concurrency monad. We consider the process as a monad transformer which takes an action (which contains the actual computation) and link it a continuation. Its type is

```
type 'a process = ('a -> action) -> action
```

The type action is

```
type action = Atom of (unit -> action) | Fork of action * action | Stop
```

We use `Fork` to instanciate new processes and `Atom` to represent a computation.
In the `run` function, we recreate a pipeline containing the first `Fork` action to execute. Each time a `Fork` is executed, we push the two actions in the pipeline. When an `Atom` is read, we execute its computation and store his continuation back in the pipeline. This procedures ends when all the continuations are `Stop`.

2.4 Pipes

3 Applications

3.1 Mandelbrot

This application consits of the computation and visualization of the Mandelbrot set using the implemented kahn network. The Mandelbrot set $\mathcal{M}$ is the set of complex numbers c such that the point $z = 0$ of the polynomial $P_c(z) = z^2 + c$ has an orbit which remains bounded.
In other words, this set can be seen as the set of complex numbers c for which the application $z_{n+1} = z_n + c$ starting from 0 for $n \to \infty$ remains bounded.

3.1.1 Properties

A very useful property of the mandelbrot set which will make its computation easier is the following:

$$c \in \mathcal{M} \iff |\mathcal{P}_c^n(0)| \leq 2 \quad \forall n \geq 0$$

The sequence is certainly divergent if the modulus of z crosses 2.

3.1.2 Implementation

We give here a quick description of our implementation.

3.1.3 Objective

Our objective is to compute and plot the Mandelbrot set $\mathcal{M}$ on a given window and range of complex numbers.

3.1.4 Evaluation of points

We map each pixel of the graph to a complex value depending on a given origin and a zoom scale (radius around the origin).
A common method for evaluating a point and checking whether it belongs to $\mathcal{M}$ consists of iterating over the sequence described above starting from 0. The algorithm stops when the maximum number of steps (given as input) is reached or if we find that $|z| > 2$ (from previous property). An obvious optimization technique would be testing $|z|^2 > 4$ in order to avoid the squared root operation of the modulus.

3.1.5 Colors

A naive implementation would be to use two colors, one for pixels which belongs to $\mathcal{M}$ and another for those that don't. Another implementation technique to visualize similarity in points would be to use the number of iterations needed for the evaluation loop to exit. We can then visualize the deep points (those which needed the maximum number of steps) and the border points.
In our algorithm, we use a technique to visualize the equipotential lines of $\mathcal{M}$.
We replace the threshold 2 by another value r which can be seen as the escape radius. We use as inputs r and the number of iterations n to compute the potential

$$V(c) = \frac{\log(\max(1, |z_n|))}{2^n}$$

This technique stays coherent in the sense that $V(c) = 0$ if and only if $c \in \mathcal{M}$. More information about the potential and required adjustments here.

3.1.6 Kahn processes

We divide the output window into sections, each one will be handled by a process. We use one last process for plotting the resulting values of the processes computations.
Computing processes send points values (coordinates and color assigned) to the channel where they wait to be retrieved by the plotting process.

3.1.7 Running mandelbrot

The mandelbrot application can be ran with arguments (`arg default type`)

`-w 1300 int` Width of the window

`-h 1000 int` Height of the window

`-n 1000 int` Number of iterations

`-p 1 int` Number of processes for computation (must divide width)

`-xo -0.5 float` Real part of origin

`-yo 0. float` Imaginary part of origin

`-z 1. float` Zoom value (radius around origin)

`-r 4. float` Escape radius value

3.1.8 Some views

```
-xo -0.7463 -yo 0.1102 -z 0.005
```

```
-xo -0.7453 -yo 0.1127 -z 0.00065
```

```
-xo -0.16 -yo 1.0405 -z 0.026
```

```
-xo -0.925 -yo 0.266 -z 0.032
```

```
-xo -0.748 -yo 0.1 -z 0.0014
```

```
-xo -0.722 -yo 0.246 -z 0.019
```

```
-xo -0.235125 -yo 0.827215 -z 0.00004
```

```
-xo -0.81153120295763 -yo 0.20142958206181 -z 0.0003
```

sectionUnix processes The implentation of the Unix processes part is mainly based on the POSIX interface. Its type is

```
type 'a process = unit -> 'a
```

Computation is mostly based on the generation of heavy Unix processes through the `fork` method from the Unix library. Communication channel is an unnamed pipe access through it's channel in/out which are file descriptors.

3.2 K means

"k-means clustering is a method of vector quantization, originally from signal processing, that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster." -Wikipedia

K-means is an example of a bulk synchronous parallel algorithm (BSP). BSP algorithms are composed from a sequence of supersteps, each of which contains:

- parallel computation, in which processes independently perform local computations and produce some values
- communication, in which processes exchange data
- barrier synchronisation, during which processes wait until every process finishes

Data-parallel programming models are typically a good fit for BSP algorithms, as each bulk synchronous phase can correspond to some number of data-parallel operations.

3.2.1 Usage

Running the algorithm requires only one argument which is the number of processes used:

```
make k_means
```

```
./k_means.native 8
```

It will run automatically on the Iris data set.

Tictactoe

This application makes use of the above Khan implementation. It aims at creating a classic Tic Tac Toe game between two players. The file is `tictactoe.ml`.

3.2.2 Design of the communication

In a classic client/server architecture, the server would have to decide the execution and communication of the khan processes between the two clients. However this implementation is a bit of over kill on the server side. In our implementation the communication happens directly between the clients where one of them becomes the host as in the following scheme:

In this implementation we use one terminal and one graphics window. It may seem that since there are not two communicating terminals/graphic windows the connection between the sockets has not been established, but in fact the ”client host” and the ”client” are two khan processes that are executed in parallel and communicate with each other only through khan channels. An improvement of this can be to establish the communication in two terminals and two graphic windows. To achieve the second is best to use a different module beside `Graphics`.

3.2.3 Structure of messages

The messages that are sent through the channel fall into four categories (`MYM`, `FYI`, `STS`, `ERR`). When a message string is composed the first three letters of it are one of the above. Hence the receiving party can recognize the type of message being sent and act accordingly :

- `MYM` (Make your move) - messages the client-host asks the client to make their move
- `FYI` (For your information) - usually send the tic tac toe board
- `STS` (Status) - send the stage/status of the game i.e `Win of player`, `Draw`, `Continue`
- `ERR` (Error) - There is an invalid move happening.

As mentioned above the `FYI` send the tic tac toe board which is an `Array` of size 9.

3.2.4 Client (host)

The client host (server as referred in the code) does most of the computation of the game. The ”server” recursively does the following in the `server_main`

1. Waits for himself to make a valid move (a move inside the board that was not previously filled).
2. Updates the board with the current move
3. Checks if there is a winner or draw; If yes then it sends the status to the client and game ends i.e. stops the connection.
4. Otherwise sends the board to the client and waits for a response.
5. When the client sends a move the host checks if the move is valid. If not then repeat step 4.
6. Otherwise repeat step 2 and 3. If game has not ended then go back to step 1.

3.2.5 Client

The client has a simplier process. It only receives messages from the host and acts accordingly to the type of message received:

1. `MYM` makes a move by clicking on the board; the move is converted into an index of the array and then sent to the host.
2. `FYI` gets the board and displays it
3. `ERR` the client made a wrong move
4. `STS` receives the status of the game.

All the following can be found in the `client_main` function.

3.3 Usage

3.3.1 Run

To play the game Tic Tac Toe run:

```
make tictactoe
./tictactoe
```

Upon execution there is a graphic window appearing. The default measurements for the window are 1000 by 620 and the board is 600 by 600. The players could insert a move by simply clicking the position they like. The first player to play is `X` and the second `0`. If you want to customize the measurements of the window you could run the following code:

```
./tictactoe -length_window x -width_window x -length x -width x
```