Haskell Social Network Simulation

About the project

The Haskell social network simulation is a command-line program that simulates a social network. It generates 10 concurrent threads corresponding to 10 users. Each user will at random times send a random message to a random user. Once 100 messages are sent, the number of messages received by each user is printed.

Design decisions

For simplicity, all the data corresponding to a user is modelled by a single User record. This allows information to easily be accessed using Haskell record notation, *user field.* When a message is sent information for the user can easily be updated by creating a new User record with the messages_received incremented and the message appended to the previous messages content.

All data corresponding to a message is modelled by a single Message record. The from and to fields are linked to a user through the unique *user_id* which saves memory and ensures data consistency compared with inserting the user itself into from and to. Messages are stored in the user record; this model's real systems which store chats between users on the user's device instead of in a central server for increased security and privacy.

There are three MVars used.

1) Users. Instead of having separate MVars for each user, a list of users is used to fetch/update users and their corresponding messages.
2) Total. Needed to keep-track between the threads of how much messages have been sent and stop the simulation once a user-defined limit is reached.
3) Winner. Initially empty, only filled when the simulation ends. This is used make main wait until the simulation has finished before printing the output stats.

Issues faced

- Initially it was hard to implement concurrency with forkIO as the simulation would randomly stop before sending 100 messages due to the following code in main being prematurely executed. This was fixed by creating an empty MVar before forkIO, a putMVar in the threads, and a takeMVar after. The takeMVar will only run once the MVar is not empty which happens when the simulation finishes.
- Sometimes the simulation would end with a “thread blocked indefinitely in a MVar operation” error which stopped any MVars being accessed after the simulation. This was fixed by adding putMVar to the if and else block of process to ensure that no threads were blocking use of the MVar once the simulation ends
- (See extra feature for more details) An attempt was made to further improve performance using mutable arrays/vectors, however it proved difficult to modify the simulation to work with the ST monad.

Extra feature

The extension focused on improving the performance of the simulation to allow for large numbers of users/messages, like a real social network. First the program was changed to allow the user to enter an input for the number of users and messages and validate the input. The rest of the program was changed to handle arbitrary numbers of users/messages, e.g., by using forkIO inside a loop. After testing the program, it was found that it scaled well with messages (processing 10^5 in 0.5s) but not users which had a runtime of 33s for 10^5 messages and 10^3 users. It was predicted that this was due to users being stored in a list which meant that the whole list needed to be copied, in O(n) time, in order to update users when a message was sent.

After investigating different data structures, it was found that Data.Sequence offered the best performance. It allowed users to be updated by changing the recipient and sender user in O(log(n)) time. This resulted in over 10x faster performance, with the program taking 2s to run. Further performance enhancements were made through parallelisation, code using map was changed to use parMap and rdeepseq. This further decreased the run-time to 1.4s.