Parallel Cellular Automata Fluid Simulation

Rongcui Dong

December 16, 2019

1 Introduction

This project implements a simple fixed time step cellular automata fluid simulator using finite difference method. Two programs are included, `cellularfluid-sim` and `cellularfluid-view`. The former implements a command line program, reading initial state from an input file and outputting states of each time step to an output file. The latter reads the output file generated by the simulator, and plays back the simulation using an SDL2 window at nominal frame rate of 60 FPS. Detailed usage is included in Appendix A.

2 Algorithm

2.1 Cellular Automata

The project's algorithm is based on a grid cellular automata where the state of each grid depends only on the state of its eight adjacent grids. The grid is a `Data.Vector` of cells, stored in row major format. Each cell may be a fluid cell, a wall, or an edge. As will be described in the Simulation section, the wall and edge distinction was originally useful, but in final implementation they behave the same.

The grid's origin is at top-left, with x going right and y going down. This grid structure yields the following type for next state function, which is mapped over the grid:

```
stepCell :: AdjCells -> Cell -> Cell
```

The adjacent cells themselves are created by a vector `imap` operation. An adjacent cell located outside the grid is assumed to be an edge.

In essense, the next state function computes the next value of the center cell given a local 3×3 grid. I built parallism out of this map operation.

2.2 Simulation: Navier-Stokes Equation

Originally, I attempted to simulate the an incompressible Navier-Stokes equation without external force [1, 3], as shown in Equation 1. However, this resulted in

extremely numerically unstable simulation, oscillating within a few time steps and blowing up to NaN quickly. Therefore, the final implementation does not use this simulator. This simulation method is described for completeness and for show of effort.

$$\frac{\partial \boldsymbol{u}}{\partial t} = -(\boldsymbol{u} \cdot \nabla)\boldsymbol{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 u \tag{1}$$

I discretized the equation combining methods described in [3, 2]. I split the integrator into stages like [3], but in places where solving linear or Poisson systems were required, I used relaxation method described in [2]. This was done to avoid treating the simulation grid as a large matrix and defeating the purpose to use cellular automata in the first place.

I used staggered grid discretization, as showin in Figure 1. Therefore, for the 9 cells available for state updating, there are 9 defined points of velocity $\boldsymbol{u}$ (blue dots), 6 defined points for each component of gradient velocity $\boldsymbol{u}_x$ (red) and $\boldsymbol{u}_y$ (green). Pressure in staggered grid is also similar.

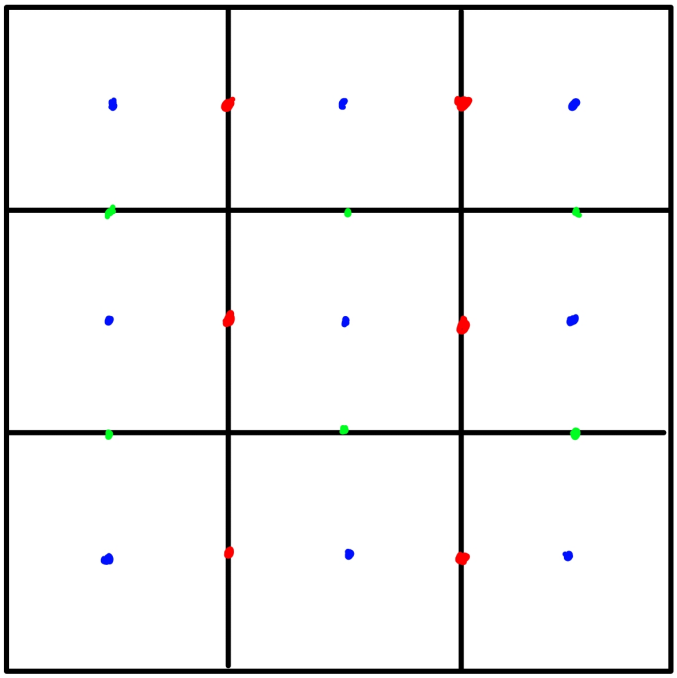

Figure 1: Staggared grid. Blue: a; red: a_x; green: a_y

Quantities at all points other than defined ones were linearly interpolated from the closest 4 valid points forming a square. Any quantity outside the rectangle bounded by all valid points were "clamped" to the boundary. Therefore, time step must be sufficiently small for discretization to be valid.

The following are discretization methods I used on defined points,

$$\boldsymbol{u}_x(x,y) = \frac{u(w/2+x,y) - u(-w/2+x,y)}{w} \tag{2}$$

$$\boldsymbol{u}_x(x,y) = \frac{u(x,w/2+y) - u(x,-w/2+y)}{w} \tag{3}$$

$$\nabla \cdot \boldsymbol{u}(x,y) = \frac{u_x(w/2+x,y) - u_x(-w/2+x,y) + u_y(x,w/2+y) - u_y(x,-w/2+y)}{w} \tag{4}$$

where w is grid width.

A wall is a boundary condition where velocity is always 0, and pressure gradient is 0. An edge is a boundary condition where both velocity gradient and pressure gradient are 0

The first step in integration was advection step, corresponding to the second step in [3],

$$\boldsymbol{u}_1 = \boldsymbol{u}(\boldsymbol{p}(\vec{x}, -\Delta t)) \tag{5}$$

where $\boldsymbol{p}(\vec{x}, t)$ is the velocity at $\vec{x} + \boldsymbol{u}(\vec{x})t$. In other words, it is the velocity obtained by tracing backwards in the velocity field for Δt. This step was straightforward, by sampling $\boldsymbol{u}(x - \Delta t\boldsymbol{u})$.

The second step was diffusion step, corresponding to the third step in [3].

$$\frac{\partial \boldsymbol{u}_2}{\partial t} = \nu \nabla^2 \boldsymbol{u}_1 \tag{6}$$

In that paper, the diffusion step is solved using an implicit method. For simplicity, I used explicit method for integration:

$$\boldsymbol{u}_2 = \boldsymbol{u}_1 + \Delta t \frac{\partial \boldsymbol{u}_2}{\partial t} \tag{7}$$

In the final step, divergence-free component of vector field $\boldsymbol{u}$ was solved to fullfill the incompressibility condition:

$$\nabla \cdot \boldsymbol{u} = 0 \tag{8}$$

Since I tried to avoid transforming the problem into matrix operations, I chose the velocity-pressure relaxation described in [2]. Note that due to the simulator's architecture (Section 2.1), the next state function could only update the center cell, the relaxation was done locally on the 3×3 grid and only the center cell's value is kept.

The relaxation started with updating pressure, a process modified from [2]:

$$\Delta p = -\beta \nabla \cdot \boldsymbol{u} \tag{9}$$

$$\beta = \frac{\beta_0 w^2}{4\Delta t} \tag{10}$$

where β_0 is a relaxation coefficient chosen for numerical stability, in range $[1, 2][2]$.

Then, local $3x3$ grid velocities were updated as following:

$$\Delta \boldsymbol{U} = \begin{pmatrix} 0 & (0, -\Delta t \Delta p / w) & 0 \\ (-\Delta t \Delta p / w, 0) & 0 & (\Delta t \Delta p / w, 0) \\ 0 & (0, \Delta t \Delta p / w) & 0 \end{pmatrix} \tag{11}$$

Then, center cell pressure was updated by Δp. The relaxation process was repeated until $\nabla \cdot \boldsymbol{u}$ was sufficiently small or until maximum iteration.

Finally, center cell velocity was updated by

$$\Delta \boldsymbol{u} = \alpha \Delta p \tag{12}$$

where α is a coefficient to guess the relaxation results for adjacent cells.

As the entire simulator used explicit integration, relaxation, and heuristics, the parameters were extremely difficult to tune for stability, and simulation diverged numerically even with small and smooth inputs. Therefore, the final implementation did not use the Navier-Stokes simulator.

2.3 Simulation: Divergence Flow

Due to the failure of the Navier-Stokes simlator, I designed a simple and stable fluid simulation algorithm that models only diffusion.

The central equation is:

$$\tilde{p} = p + (\nabla \cdot p) \Delta t / \alpha \tag{13}$$

$$\alpha = \alpha_0 \nu \tag{14}$$

where α_0 is a coefficient on the order of 10^9, used to dampen diffusion and provide numerical stability.

$\nabla \cdot p$ was discretized so that it used information on all adjacent cells:

$$\nabla \cdot p = \frac{4p - \sum_{p' \in edge} p'}{w} + \frac{4p - \sum_{p' \in corner} p'}{\sqrt{2} w} \tag{15}$$

Walls and edges are both treated with pressure gradient of 0.

This solver was stable, and allowed visually verification, such as in Figure 2.

3 Parallelization

Multiple parallization methods were attempted. All benchmarks were performed on a i9-9900K CPU with 8 physical core and 16 threads, with 64GB of memory. I identified, through profiling, two main area of performance bottleneck: grid stepping and grid outputting. The stepper was accelerated via parallization, while grid output acceleration was attempted using a dedicated output thread.

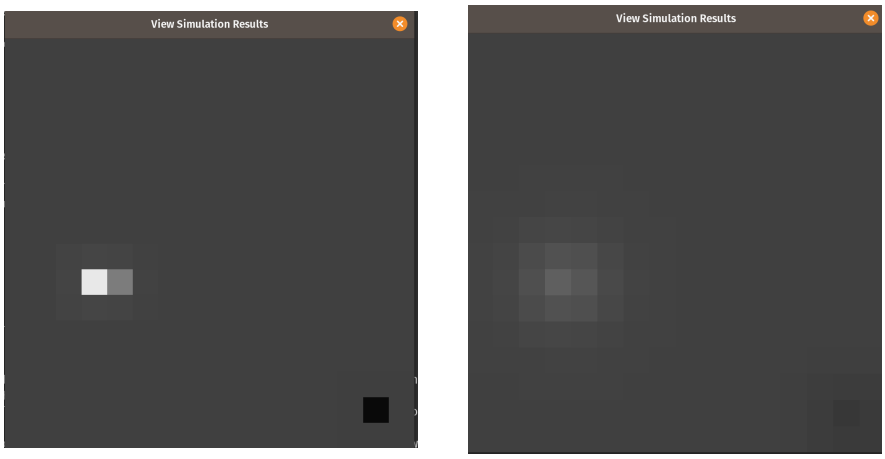

Figure 2: Small input simulation screenshots showing diffusion over time

For the stepper, multiple data structures were tested, and all were based on the Vector type. The first method I tried was encapsulating chunks of vectors in a datatype `ParVector a`, providing `map` and `imap` interface so semantically it functioned like `Vector a`. The second method I tried was to keep the `Vector a` structure, and to provide parallel versions of `map` and `imap`. In each method, I tested chunking as `Vector (Vector a)` and as `[Vector a]`. In addition, I tested sparking with `par`, `parMap` or `parTraversable`, and the Par monad. In each chunking method, I tested using Vector `Bundle` in hopes it is more efficient in concatenation.

For the output thread, I tested blocking IO in the computation thread and sending the grid through a `TBMQueue` of various sizes to an output thread. In each case, I tested cloning the vector and using the vector as is. In addition, I compared performance of storing as text format and serializing into binary format. In the same-thread case, I also tried various buffer sizes.

4 Results

Table 1 compares performance tests using final parallel implementation and sequential implementation (not included in source code, but easy to change by substituting all `parMapV` and `parMapV` with `map` and `imap`).

Grid Size	Time Steps	N1	N2	N4	N8	N16
16×16	600	$0.348s$	$0.357s$	$0.372s$	$0.410s$	$0.619s$
256×256	600	$40.5s$	$22.0s$	$19.5s$	$27.6s$	$44.1s$
512×512	600	$2m50s$	$2m16s$	$1m58s$	$2m14s$	$4m58s$

Table 1: Benchmark Results

I found that `ParVector` and parallelized `Vector` had nearly identical performance, given identical chunk size. Therefore, for readability I kept `Vector` in the final implementation. In either case, chunking did not provide any advantages until they were at least 1024 cells each, and until the simulation grid was at least 256×256 in size. In addition, chunking as `[Vector a]` provided significant speed up over `Vector (Vector a)`, possibly due to how they were manipulated when they were split and concatenated. Using `Bundle` provided no speedup. In all cases, the Par monad gave best performance, but `[Vector a]` chunking with `parMap` came close.

Single thread output writing with large buffer size worked better than sending to another output thread. Cloning did not help making output and computation concurrent, and had no positive effects. In all cases, binary output performed better than text output.

In final implementation, profiling showed that IO was not the performance bottleneck:

COST CENTRE	MODULE	SRC	%time	%alloc
divP'.f	CellularFluid.Grid.FD.FDSimpleStepper	src/CellularFluid/Grid/FD/FDSimpleStepper.hs:21:5-20	21.7	7.7
>>=	Data.Vector.Fusion.Util	Data/Vector/Fusion/Util.hs:36:3-18	11.2	10.6
adjCells.f.adjEdge	CellularFluid.Grid	src/CellularFluid/Grid.hs:60:11-51	8.3	4.4
primitive	Control.Monad.Primitive	Control/Monad/Primitive.hs:195:3-16	7.7	9.2
adjCells.f.adjCorner	CellularFluid.Grid	src/CellularFluid/Grid.hs:61:11-57	6.9	4.5
fmap	Data.Vector.Fusion.Stream.Monadic	Data/Vector/Fusion/Stream/Monadic.hs:(133,3)-(135,20)	6.2	7.6
sum	Data.Vector	Data/Vector.hs:425:3-11	4.9	3.8
basicUnsafeWrite	Data.Vector.Mutable	Data/Vector/Mutable.hs:118:3-65	3.6	5.1
basicUnsafeGrow	Data.Vector.Generic.Mutable.Base	Data/Vector/Generic/Mutable/Base.hs:(138,3)-(144,23)	3.4	1.4
basicUnsafeIndexM	Data.Vector	Data/Vector.hs:277:3-62	3.0	2.2
adjCells.f	CellularFluid.Grid	src/CellularFluid/Grid.hs:(50,5)-(63,19)	2.4	3.6
cellAt	CellularFluid.Grid	src/CellularFluid/Grid.hs:(34,1)-(42,20)	2.3	2.4
cellAt.idx	CellularFluid.Grid	src/CellularFluid/Grid.hs:39:5-24	2.2	0.0
basicUnsafeNew	Data.Vector.Mutable	Data/Vector/Mutable.hs:(99,3)-(102,32)	2.1	3.5
rnf.rnfAll	Data.Vector	Data/Vector.hs:(225,11)-(226,36)	1.9	0.5
basicUnsafeSlice	Data.Vector.Mutable	Data/Vector/Mutable.hs:89:3-62	1.2	4.1
parMapV.f'	Data.Vector.Parallel	src/Data/Vector/Parallel.hs:38:5-27	1.1	0.9
basicUnsafeFreeze	Data.Vector	Data/Vector.hs:(263,3)-(264,47)	1.0	3.2
basicUnsafeCopy	Data.Vector.Mutable	Data/Vector/Mutable.hs:(121,3)-(122,36)	0.5	1.8
>>=	Data.Serialize.Put	src/Data/Serialize/Put.hs:(173,5)-(176,37)	0.4	6.1
>>=.(...)	Data.Serialize.Put	src/Data/Serialize/Put.hs:175:13-36	0.3	2.4
*>	Data.Serialize.Put	src/Data/Serialize/Put.hs:(162,9)-(165,41)	0.1	1.7

Using threadscope, I identified sequential regions between two time steps, as shown in Figure 3. This region was present in all implementations, including the sequential one. I could not identify its origin from threadscope, and ghc-events-analyze seemed to suggest an internal synchronization of the `Data.Vector` implementation. The `ParVector` and cloning was an attempt to avoid this overhead, but they both had no effect. The first sequential region in the figure is reading grid input.

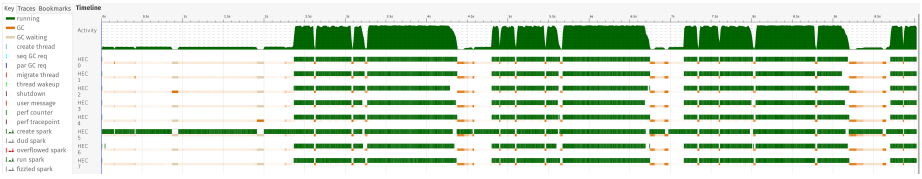

Figure 3: Threadscope output of a typical simulation.

5 Conclusion

In summary, the stepper of the simulator could be parallized. However, an unidentified sequential region limited the amount of speed up achievable.

References

[1] Navier-stokes equations. `https://en.wikipedia.org/wiki/Navier%E2%80%93Stokes_equations`. Accessed 2019-12-16.

[2] Nick Foster and Dimitri Metaxas. Realistic animation of liquids. *Graphical models and image processing*, 58(5):471–483, 1996.

[3] Jos Stam. Stable fluids. In *Siggraph*, volume 99, pages 121–128, 1999.

A Command Line Usage

```
Usage: cellularfluid-sim [--version] [--help] [-v|--verbose] (-i|--
input ARG)
                         (-o|--output ARG) --time ARG

Available options:
  --version                Show version
  --help                   Show this help text
  -v,--verbose             Verbose logging?
  -i,--input ARG           Input file
  -o,--output ARG          Output file
  --time ARG               Simultaion time
```

```
Views fluid simulation

Usage: cellularfluid-view [--version] [--help] [-v|--verbose] [--width ARG]
                          [--height ARG] [--hidpi] (-i|--input ARG)

Available options:
  --version                Show version
  --help                   Show this help text
  -v,--verbose             Verbose logging?
  --width ARG              Window width
  --height ARG             Window height
  --hidpi                  HiDPI support
  -i,--input ARG           Input file
```

B Grid Input Format

First line is a header storing grid metadata, separated by space:

```
Width(# Columns)::Int Height(# Rows)::Int Grid_Size::Double Density::Double Viscosity::Double
```

After the header, each entry has its own format, while entries are separated by spaces.

Walls are specified with `W`; fluid cells using divergence-flow simulation are specified with `FD p` where `p` is a `Double` value for pressure. Edges cannot be specified in grid input.

The number of cells must match *width* × *height* specified in header, or the program fails a sanity check and exits without starting simulation.

C Code Listing

```haskell
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE TemplateHaskell #-}
module Main (main) where

import Import
import Run
import RIO.Process
import Options.Applicative.Simple
import qualified Paths_cellularfluid

main :: IO ()
main = do
  (options, ()) <- simpleOptions
    $(simpleVersion Paths_cellularfluid.version)
    "Simulates fluid"
    "Description: TODO"
    (Options
       <$> switch ( long "verbose"
                  <> short 'v'
                  <> help "Verbose logging?"
                   )
       <*> strOption ( long "input"
                     <> short 'i'
                     <> help "Input file"
                     )
       <*> strOption ( long "output"
                     <> short 'o'
                     <> help "Output file"
                     )
       <*> option auto ( long "time"
                       <> help "Simultaion time"
                        )
    )
    empty
  lo <- logOptionsHandle stderr (optionsVerbose options)
  pc <- mkDefaultProcessContext
  withLogFunc lo $ \lf ->
    let app = App
          { appLogFunc = lf
          , appProcessContext = pc
          , appOptions = options
          }
     in runRIO app run
```

Listing 1: app/Main.hs

```haskell
module CellularFluid
    ( module CellularFluid.Grid
    , module CellularFluid.Grid.Types
    , module CellularFluid.Grid.Parse
```

```
  ) where

import CellularFluid.Grid
import CellularFluid.Grid.Types
import CellularFluid.Grid.Parse (parseGrid)
```

Listing 2: src/CellularFluid.hs

```
{-# LANGUAGE NoImplicitPrelude #-}
module Import
  ( module RIO
  , module Types
  ) where

import RIO
import Types
```

Listing 3: src/Import.hs

```
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
module Run (run) where

import Import

import Sim

import System.IO (openFile)

import CellularFluid.Grid.Types
import CellularFluid.Grid.Parse

-- | RIO top level entry point
run :: RIO App ()
run = do
  app <- ask
  let opt = appOptions app
  when (optionsVerbose opt) outputOptions
  let fpIn = optionsInput opt
      fpOut = optionsOutput opt
  egrid <- loadGrid fpIn
  outHandle <- openOutput fpOut
  sim <- case egrid of
           Right (grid, phys) -> setupSim grid phys outHandle
           Left s -> (logError . fromString $ "Grid parse error: " ++ s) >> exitFailure
  runRIO sim runSim



-- | Logs options when verbose is ON
outputOptions :: (HasOptions env, HasLogFunc env)
              => RIO env ()
outputOptions = do
    env <- ask
    let opt = env ^. optionsL
    logInfo "Verbose flag ON"
    logInfo . fromString $ "INPUT: " ++ optionsInput opt
    logInfo . fromString $ "OUTPUT: " ++ optionsOutput opt


-- | Loads simulation grid
loadGrid :: (HasLogFunc env)
         => FilePath
         -> RIO env (Either String (Grid, PhysCfg))
loadGrid fp = do
    logDebug . fromString $ "Loading grid from: " ++ fp
    txt <- readFileUtf8 fp
```

```
    let result = parseGrid txt
    logDebug . fromString $ "Done loading grid from: " ++ fp
    return result


-- | Opens output file
openOutput :: (HasLogFunc env)
           => FilePath
           -> RIO env Handle
openOutput fp = do
    logDebug . fromString $ "Opening output file: " ++ fp
    h <- liftIO $ openFile fp WriteMode
    hSetBuffering h $ BlockBuffering $ Just 268435456 -- 256 MB
    logDebug . fromString $ "Done opening output file: " ++ fp
    return h


-- | Sets up simulation environment
setupSim :: (HasLogFunc env, HasOptions env)
         => Grid
         -> PhysCfg
         -> Handle
         -> RIO env SimApp
setupSim grid phys h =
    let simCfg = SimulationCFG { cfgTimeStep = 1.0 / 60.0
                               , cfgPhysics = phys
                               }
     in do
        env <- ask
        let opt = env ^. optionsL
        return $ SimApp { simHandle = h
                        , simCfg = simCfg
                        , simGrid = grid
                        , simLogFunc = env ^. logFuncL
                        , simSteps = optionsTimeStep opt
                        }
```

Listing 4: src/Run.hs

```
module Sim
  ( runSim
  ) where

import           Import

import           RIO.ByteString
import           RIO.State

import qualified Data.Serialize                as S

import           CellularFluid

-- | The main function of simulation
runSim :: RIO SimApp ()
runSim = do
  env <- ask
  logDebug "Checking grid sanity..."
  when (not . gridIsSane $ simGrid env) $ do
    logError "Grid insane"
    exitFailure
  logDebug "Done checking grid sanity. Grid is sane."
  logDebug "Start simulation."
  simLoop $ simSteps env
  logDebug "End simulation."
  logDebug "Waiting for output to finish..."
  hClose $ env ^. outHandleL
  logDebug "Done waiting for output to finish."
```

```haskell
-- | Main simulation loop
simLoop :: Int -> RIO SimApp Grid
simLoop nmax = (simGrid <$> ask) >>= evalStateT (go 0)
  where
    go n'
      | n' >= nmax = get
      | otherwise = do
        when (n' `mod` 100 == 0) $
          lift $ logDebug . fromString $ "Iteration: " ++ show n'
        stepSim
        go (n' + 1)

-- | Steps grid once and outputs
stepSim :: HasSimInfo env => StateT Grid (RIO env) ()
stepSim = do
  env <- lift ask
  grid <- get
  let cfg = env ^. simCfgL
      h = env ^. outHandleL
  let grid' = stepGrid cfg grid
    -- Writes output
  hPut h (S.encode grid')
  put grid'
```

Listing 5: src/Sim.hs

```haskell
{-# LANGUAGE NoImplicitPrelude #-}

module Types where

import          RIO
import          RIO.Process

import          CellularFluid.Grid.Types

-- | Command line arguments
data Options =
  Options
    { optionsVerbose  :: !Bool
    , optionsInput    :: !FilePath
    , optionsOutput   :: !FilePath
    , optionsTimeStep :: !Int
    }

data App =
  App
    { appLogFunc        :: !LogFunc
    , appProcessContext :: !ProcessContext
    , appOptions        :: !Options
    }

data SimApp =
  SimApp
    { simHandle  :: !Handle
    , simCfg     :: !SimulationCFG
    , simGrid    :: !Grid
    , simLogFunc :: !LogFunc
    , simSteps   :: !Int
    }

instance HasLogFunc App where
  logFuncL = lens appLogFunc (\x y -> x {appLogFunc = y})

instance HasLogFunc SimApp where
  logFuncL = lens simLogFunc (\x y -> x {simLogFunc = y})

instance HasProcessContext App where
  processContextL = lens appProcessContext (\x y -> x {appProcessContext = y})
```

```
class HasOptions env where
  optionsL :: Lens' env Options

instance HasOptions App where
  optionsL = lens appOptions (\x y -> x {appOptions = y})

class HasSimInfo env where
  simCfgL :: Lens' env SimulationCFG
  outHandleL :: Lens' env Handle

instance HasSimInfo SimApp where
  simCfgL = lens simCfg (\x y -> x {simCfg = y})
  outHandleL = lens simHandle (\x y -> x {simHandle = y})
```

Listing 6: src/Types.hs

```
module CellularFluid.Grid where

import              RIO

import qualified RIO.Vector                                as V
import qualified RIO.Vector.Unsafe                         as V'

import              Data.Vector.Parallel

import qualified CellularFluid.Grid.FD.FDSimpleStepper as FD
import              CellularFluid.Grid.Types

-- | Simulates one cell for one timestep
stepCell :: SimulationCFG -> AdjCells -> Cell -> Cell
stepCell cfg adjs cell = step cell
  where
    phy = cfgPhysics cfg
    step (FluidD p) =
      let width = phyGridSize phy
          dt = cfgTimeStep cfg
          mu = phyFDMu phy
          rho = phyFDRho phy
          nu = mu / rho
          p' = FD.step width dt nu p adjs
       in FluidD p'
    step x = x

-- | Gets cell at (r, c)
--
-- If out of bounds, cell is Wall
cellAt :: Grid -> Int -> Int -> Cell
cellAt g r c
  | r < 0 || c < 0 || r >= h || c >= w = Wall
  | otherwise = cs `idx` (w * r + c) -- We guarentee not out of bounds
      -- idx = parIndex'
  where
    idx = V'.unsafeIndex
    cs = gridCells g
    w = gridWidth g
    h = gridHeight g

-- | Get vector of adjacent cells
adjCells :: Grid -> Vector (Cell, AdjCells)
adjCells g = parIMapV f cs -- (V.imap f cs)
  where
    cs = gridCells g
    w = gridWidth g
    cell = cellAt g
    f i x =
      let (r, c) = i `divMod` w
          acN = cell (r - 1) c
```

```
            acNW = cell (r - 1) (c - 1)
            acW = cell r (c - 1)
            acSW = cell (r + 1) (c - 1)
            acS = cell (r + 1) c
            acSE = cell (r + 1) (c + 1)
            acE = cell r (c + 1)
            acNE = cell (r - 1) (c + 1)
            adjEdge = V.fromList [acN, acW, acE, acS]
            adjCorner = V.fromList [acNW, acNE, acSW, acSE]
            adjs = Adj {..}
        in (x, adjs)

-- | Steps grid once
stepGrid :: SimulationCFG -> Grid -> Grid
stepGrid cfg grid = grid {gridCells = cells'}
  where
    cells' = step `parMapV` adjs
    adjs = adjCells grid
    step (cell, adjcells) = stepCell cfg adjcells cell

-- | Checks whether the grid is sane.
--
-- Currently verifies grid size is consistent with (width * height)
gridIsSane :: Grid -> Bool
gridIsSane grid =
  (V.length $ gridCells grid) == (gridHeight grid * gridWidth grid)
```

Listing 7: src/CellularFluid/Grid.hs

```
{-# LANGUAGE DeriveAnyClass #-}
{-# LANGUAGE DeriveGeneric  #-}

module CellularFluid.Grid.Types where

import           RIO
import qualified RIO.List                  as L
import qualified RIO.Vector                as V

import qualified Data.Serialize            as S
import           Data.Vector.Serialize()

import           Numeric

import           Linear

-- | Time
type T = Double

type DT = Double

-- | Length
type L = Double

-- | Position
type X = V2 Double

-- | Velocity
type U = V2 Double

-- | Acceleration
type DU = V2 Double

-- | ·(u/t)
type DIV_DU = Double

-- | Pressure
type P = Double
```

```
type DP = Double

-- | Density
type Rho = Double

-- | Kinematic Viscosity
type Mu = Double

type Nu = Double

-- | Next state logic
type NSL = Cell -> AdjCells -> Cell

{-|
   Data type for one simulation cell
-}
data Cell
  = FluidD P -- ^ Diffusive fluid
  | Wall -- ^ Perfect wall cell
  | Edge -- ^ Edge
  deriving (Generic, NFData)

instance S.Serialize Cell

-- | Adjacent cells
data Adj a =
  Adj
    { adjEdge   :: Vector a
    , adjCorner :: Vector a
    }
  deriving (Show, Functor, Generic, NFData)

type AdjCells = Adj Cell

data Grid =
  Grid
    { gridCells  :: !(Vector Cell) -- ^ Row major, top-left is (0, 0)
    , gridWidth  :: !Int
    , gridHeight :: !Int
    }
  deriving (Generic, NFData)

instance S.Serialize Grid

instance Show Grid where
  show = showGrid

data PhysCfg =
  PhysCfg
    { phyGridSize :: Double
    , phyFDRho    :: Double
    , phyFDMu     :: Double
    }
  deriving (Show)

showGrid :: Grid -> String
showGrid g = go cells
  where
    rowSize = gridWidth g
    cells = gridCells g
    go :: Vector Cell -> String
    go cs
      | null cs = ""
      | otherwise =
        let (h, t) = V.splitAt rowSize cs
            hstr = showRow h ++ "\n"
         in hstr `seq` (hstr ++ go t)
```

```
cloneGrid :: Grid -> Grid
cloneGrid g = g {gridCells = cells'}
  where
    cells = gridCells g
    cells' = (V.new . V.clone) cells

instance Show Cell where
  show = showCell

showCell :: Cell -> String
showCell (FluidD p) = L.intercalate "␣" ["FD", sg p]
  where
    sg a = showGFloat Nothing a ""
showCell Wall = "W"
showCell Edge = "E"

showRow :: Vector Cell -> String
showRow = L.intercalate "," . toList . fmap show

{-|
   Grid configuration
-}
data SimulationCFG =
  SimulationCFG
    { cfgTimeStep :: Double
    , cfgPhysics  :: PhysCfg
    }
```

Listing 8: src/CellularFluid/Grid/Types.hs

```
module CellularFluid.Grid.Parse where

import           RIO
import qualified RIO.Vector                as V

import           Data.Attoparsec.Text

import           CellularFluid.Grid.Types

data GridCfg =
  GridCfg
    { width     :: Int
    , height    :: Int
    , size      :: Double
    , density   :: Double
    , viscosity :: Double
    }

parseGrid :: Text -> Either String (Grid, PhysCfg)
parseGrid = parseOnly gridParser

gridParser :: Parser (Grid, PhysCfg)
gridParser = do
  cfg <- pGridCfg
  cells <- pGridCells cfg
  let grid =
        Grid {gridCells = cells, gridWidth = width cfg, gridHeight = height cfg}
      phys =
        PhysCfg
          { phyGridSize = size cfg
          , phyFDRho = density cfg
          , phyFDMu = viscosity cfg
          }
  return (grid, phys)

pGridCfg :: Parser GridCfg
pGridCfg = do
  width <- decimal
```

```
  many1 space
  height <- decimal
  many1 space
  size <- double
  many1 space
  density <- double
  many1 space
  viscosity <- double
  endOfLine <?> "Too many arguments on first line"
  return $ GridCfg {..}

pGridCells :: GridCfg -> Parser (Vector Cell)
pGridCells _ = do
  cs <- pCell `sepBy1` space
  return $ V.fromList cs

pCell :: Parser Cell
pCell = pFluidD <|> pWall

pFluidD, pWall :: Parser Cell
pFluidD = do
  string "FD" -- Diffusive Fluid
  many1 space
  p <- double
  return $ FluidD p

pWall = string "W" >> return Wall
```

Listing 9: src/CellularFluid/Grid/Parse.hs

```
module CellularFluid.Grid.FD.FDSimpleStepper where

{-
   Simple diffusive fluid
   -}
import            RIO

import            CellularFluid.Grid.Types

-- | Main stepping function
step :: L -> DT -> Nu -> P -> AdjCells -> P
step w dt  p adjs = p - divP * dt /
  where
     = 1e9 *  -- magic
    divP = divP' adjs p / w

-- | Scaled divergence of P: w(·P)
divP' :: AdjCells -> P -> Double
divP' (Adj es cs) p = f es + f cs / (sqrt 2.0)
  where
    f = sum . fmap g
    g (FluidD p') =
      let !dp = p - p'
       in dp
    g _ = 0.0
```

Listing 10: src/CellularFluid/Grid/FD/FDSimpleStepper.hs

```
module Data.Vector.Parallel where

import            RIO
import qualified RIO.List                  as L
import            RIO.List.Partial          (tail)
import qualified RIO.Vector                as V

import            Control.Monad.Par
```

```
-- | Splits vector into list of chunks.
-- Chunk order is reversed
--
-- concatV . chunksOf == id
chunksOf :: Int -> Vector a -> [Vector a]
chunksOf n v = vs
  where
    vs = go v []
    go v' xs
      | null v' = xs
      | otherwise =
        let (c, t) = V.splitAt n v'
         in go t (c : xs)

-- | Concatenates chunks
--
-- concatV . chunksOf == id
concatV :: [Vector a] -> Vector a
concatV = go V.empty
  where
    go v' [] = v'
    go v' (v:vs) =
      let v'' = v <> v'
       in go v'' vs

-- | Parallel version of V.map
parMapV :: NFData b => (a -> b) -> Vector a -> Vector b
parMapV f va = concatV . runPar $ f' `mapM` chunks >>= traverse get
  where
    chunks = chunksOf 4096 va
    f' v = spawnP $ f <$> v

-- | Parallel version of V.imap
parIMapV :: NFData b => (Int -> a -> b) -> Vector a -> Vector b
parIMapV f va = concatV . runPar $ zipWithM f' acc chunks >>= traverse get
    where
        chunks = chunksOf 4096 va
        lengths = V.length <$> chunks
        acc = tail $ L.scanr (+) 0 lengths
        f' i0 as = spawnP $ V.imap (\i a -> f (i+i0) a) as
```

Listing 11: src/Data/Vector/Parallel.hs