Parallelized 2-Dimensional Smoothed Particle Hydrodynamics Simulation

Zach Schuermann
zvs2002@columbia.edu

Fall 2019

Abstract

This paper presents a parallelized smoothed particle hydrodynamics (SPH) solver implemented in Haskell. Its implementation is manifested in a 2-dimensional fluid simulation. In an effort to demonstrate the robust set of parallelization techniques in Haskell, I have implemented Müller's algorithm for particle-based fluid simulation and tested a handful of parallelization techniques in the process. The solver includes a graphical user interface for tuning and validating the solver as well as a command-line interface for executing performance analysis. Empirical results are included showing diminishing returns as the number of cores increase, albeit better performance at each increment. The 2-core tests yielded a speedup of 1.99x, while the 6-core tests achieved 5.13x speedup on a modern Coffee Lake Intel CPU.

1 Introduction

As computing has entered an era lacking the luxuries afforded by Moore's Law, more systems rely on increasing the number of processing cores to provide greater processing capabilities as software workloads continue to grow. In order to make use of multiple cores, one must develop application software to explicitly spread the workload, share memory across threads/processes, and provide mutual exclusion to critical sections. In a pure, functional language like Haskell where there are no side effects, implementing parallel/concurrent software becomes much less of a chore of keeping track of shared state and mutual exclusion; instead, it often simply requires a few simple additions "talking to" Haskell's runtime to capitalize on multiple hardware cores.

In order to effectively test and demonstrate Haskell's capabilities in the realm of parallel processing, a CPU-bound task would need to be chosen as a problem to solve. Particle simulation is a *strictly* CPU-bound processing task that can scale to arbitrary difficulty with the number of particles in the simulation. Fluid simulation belongs to a rather popular subset of computer graphics as its use is found throughout gaming, simulation, and animation. The solver I have implemented is a particle-based fluid simulation and a pseudo-implementation of the solver described in Müller's "Particle-Based Fluid Simulation for Interactive Applications."[3]

While there are many SPH solvers/simulations in existence, this aims to be a rather elegant application in Haskell with minimal additions to provide maximum speedup on multiple cores. The solver is a based on Lucas Schuermann's "2D SPH solver in C++"[1][5].

2 Smoothed Particle Hydrodynamics

In the interest of time, my implementation omits surface tension; the remainder of the design mirrors that of Müller's original paper. This solver is a Lagrangian (particle-based) method (as opposed to Eulerian, which is grid-based). As such, the entire simulation rests on the premise that one can effectively simulate a fluid with a large number of discrete particles interacting with each other.

The simulation requires creating a number of particles which hold their current state: the position vector, velocity vector, force vector, pressure, and density. The particles move through time and space according to an "updating" function, which will transform the array of particles at time t

[1] GitHub for his implementation: https://github.com/cerrno/mueller-sph/blob/master/src/main.cpp

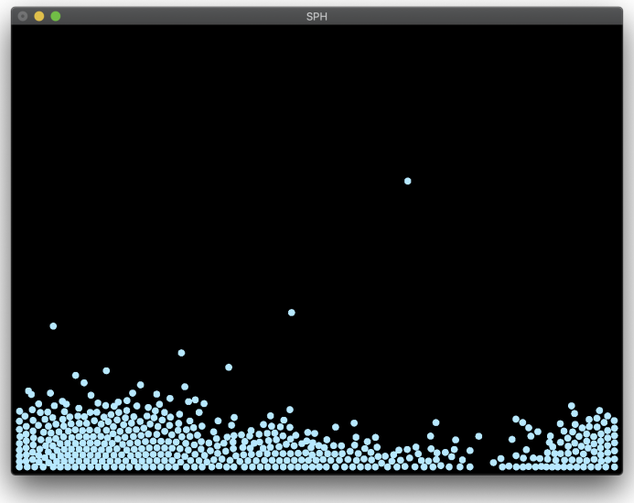

Figure 1: The OpenGL-rendered 2D particle simulation

to the new array at time $t+1$. The function that governs this updating step relies on something similar to a weighed average for each particle over all *other* particles [4]. Equation (1) below describes how to update the density of particle i from summing over all other particles. Likewise, Equation (2) and (3) govern the update step for a given particle for its pressure and viscosity, respectively.

$$\rho_i = \rho(\mathbf{r}_i) = \sum_j m_j \frac{\rho_j}{\rho_j} W(|\mathbf{r}_i - \mathbf{r}_j|, h) = \sum_j m_j W(|\mathbf{r}_i - \mathbf{r}_j|, h) \tag{1}$$

$$\mathbf{F}_i^{\text{pressure}} = -\nabla p(\mathbf{r}_i) = -\sum_j m_i m_j (\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}) \nabla W(|\mathbf{r}_i - \mathbf{r}_j|, h) \tag{2}$$

$$\mathbf{F}_i^{\text{viscosity}} = \eta \nabla^2 \mathbf{u}(\mathbf{r}_i) = \eta \sum_j m_j \frac{|\mathbf{u}_j - \mathbf{u}_i|}{\rho_j} \nabla^2 W(|\mathbf{r}_i - \mathbf{r}_j|, h) \tag{3}$$

In order to update the position and velocity according to the approximated force, pressure, and density given by SPH, one can simply apply Euler's method of forward numerical integration to calculate the new velocity and subsequently the new position. This is described in Equations (4) and (5)[5].

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \Delta t \cdot \frac{\mathbf{F}_{\text{total}}}{\rho} \tag{4}$$

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \Delta t \cdot \mathbf{v}_{t+1} \tag{5}$$

3 Haskell Implementation

The overall design will rely on the aforementioned unoptimized, feature-sparse method of SPH to "solve" the state of the simulation at every timestep before rendering [3, 5]. The simulation is the classic "dam break" in which a block of fluid is dropped into a rigid container. The application provides two means of interaction (1) through a graphical user interface (GUI), and (2) through a command-line interface (CLI). The GUI provides an effective means to visualize the simulation, allowing for tuning (with respect to the constants set for SPH) and validation that the simulation is behaving like a fluid. The CLI, on the contrary, provides a means to forgo the overhead associated with rendering and simply run the solver on an list of particles in order to perform more accurate analysis on runtime metrics.

3.1 Solver

In the conventional imperative approach, the state of the simulation is stored as a vector of particles which are traversed at every iteration of the simulation to produce the animation. In Haskell, the implementation relies on recursing over a list of particles to transform the current state into future states of simulation. The state of each particle is stored in the `Particle` datatype. This holds three two-dimensional vectors (the `V2` type, from `Linear.V2`) and two scalar `Doubles`.

```haskell
data Particle = Particle {point    :: V2 Double,
                          velocity :: V2 Double,
                          force    :: V2 Double,
                          density  :: Double,
                          pressure :: Double} deriving (Eq, Show)
```

The chief functionality of the solver is implemented in the `update` function. This function takes a list of `Particles` and returns a new list after computing the new density, pressure, and forces, as well as performing forward integration to compute the new velocity and position.

```haskell
update :: [Particle] -> [Particle]
update ps = map integrate (forces (densityPressure ps))

integrate :: Particle -> Particle              -- forward integration
forces :: [Particle] -> [Particle]             -- calculate new forces
densityPressure :: [Particle] -> [Particle]    -- new density, pressure
```

3.2 Renderer

Rendering side-effects is handled via binding the `update` function to callbacks within the OpenGL runtime. This project relies heavily on GLUT, The OpenGL Utility Toolkit. Haskell has bindings in the GLUT package, in `Graphics.UI.GLUT`. In order to pass around particles for updating, `IORef` was used to provide mutable references in the `IO` monad. In the application, the particles (ps below) are initialized by creating an `IORef` for the particles and passing this reference around to various callbacks including the `displayCallback` (to render the particles) and the `idleCallback` (to update the particles). A simplified version of the application code is given below to illustrate the usage of the particle references and binding callbacks for rendering/updating:

```haskell
ps <- newIORef particles  -- particles 'mutated' in IORef

-- bind OpenGL callbacks to our functions
displayCallback $= display ps
idleCallback $= Just (idle pps iters ps iterRef)

-- idle function updates state (simplified)
idle :: IORef [Particle] -> IORef Int -> IdleCallback
idle ps iter = do
  ps' <- get ps
  i <- get iter

  if i > num_iters
    then exitWith ExitSuccess
    else writeIORef iter (i+1)

  writeIORef ps (update ppc ps')
```

3.3 Dam Break

In the snippet above, I glossed over the origin of the `particles` variable. They must be created in the "dam break" configuration. The "dam break" is achieved by initializing a large array of particles arranged in a rectangle with zero velocity (and preset density etc.). In order to provide a clean simulation, a small amount of *jitter* is added to the x-dimension of the particles on initialization [5]. This is done through a simple random number generator applying jitter to each initialized particle.

```
-- number of points, iterations, chunks
simInit :: Int -> Int -> Int -> IO [Particle]
simInit nps iters chks = do
  gen <- getStdGen
  let jitters = randoms gen :: [Double]
      points = take nps $ initPoints jitters
  return $ makeParticles points
```

Then, the particles can be passed into the aforementioned step of creating an `IORef` and used in the simulation.

4 Parallelization

4.1 Implementation

Although many parallelization techniques were attempted, only two of the attempts reliant on Strategies are discussed, for brevity. In order to parallelize the solver, I utilize Evaluation Strategies; these are nothing more than a function in the `Eval` monad that take a value of some type and return the same type[1]. The general idea for parallelism is spawning a number of threads, or rather spawn a computation that can be handled with a sort of thread pool, (via *sparks*) for some portion of the `update` operation. That is, I have utilized Strategies for generating sparks for portions of the list of particles to update - effectively dividing the list of particles among possible worker threads to gain parallelism.

4.2 Strategies

In order to employ the aforementioned Strategies (and evaluate the computation to normal form in the CLI mode), the `Particle` type is added as an instance of the `NFData` typeclass (representing normal form data).

```
instance NFData Particle where
  rnf (Particle p v f d pr) = rnf p `deepseq`
                              rnf v `deepseq`
                              rnf f `deepseq`
                              rnf d `deepseq`
                              rnf pr
```

Next, I investigate two means of parallelism: a naïve approach using `parList` and a chunking method using `parListChunks`.

4.3 Naïve `parList` or `parBuffer`

A surprisingly minimal amount of code need be written to implement many of the strategies available. In order to implement `parList` combined with `rseq` to evaluate list elements in parallel, we can simply augment the `update` function.

```
parList :: Strategy a -> Strategy [a]
rseq :: Strategy a
```

This sparks evaluation of the list in parallel using the `parList` Strategy.

```
update :: [Particle] -> [Particle]
update ps = map integrate (forces (densityPressure ps))
            `using` parList rseq
```

Unfortunately, the sparks are not well distributed, and one can see that it is sparking far too many computations:

```
SPARKS: 1000000(8832 converted, 991072 overflowed, 0 dud, 0 GC'd, 96 fizzled)
```

Furthermore, the **user** runtime increased by a factor of 2, suggesting that we are incurring a prohibitively large amount of overhead.

```
74.67s user 0.20s system 199% cpu 37.505 total
```

In the reference simulation of $N = 1000$ particles and $I = 1000$ iterations, using the `parList` or `parBuffer` strategies both yielded suboptimal performance due to generating a large number of sparks at the beginning of computation. This is shown below in Figure 3 and 4. One can alleviate these problems with chunking[1].

4.4 Chunking with `parListChunks`

In order to spread sparks throughout execution and keep the load well-balanced across threads, chunking is employed via the `parListChunks` strategy. It is similar to above, with the addition of the `Int` parameter which takes the number of parts per chunk.

```
parList :: Strategy a -> Strategy [a]
rseq :: Strategy a

-- parts per chunk -> Strategy a -> Strategy [a]
parListChunks :: Int -> Strategy a -> Strategy [a]
```

Again, implementation is as simple as augmenting the `update` function. Although this time, we add an additional parameter such that the user can control the relative granularity of chunking.

```
update :: Int -> [Particle] -> [Particle]
update partPerChunk ps = map integrate (forces (densityPressure ps))
                         `using` parListChunk partPerChunk rseq
```

Through chunking, the sparks are effectively spread throughout execution and greater performance is achieved. This parallelization is analyzed in detail in the following results.

4.5 Testing

In order to test the different approaches, a handful of Python scripts ran the tests multiple times (usually five) and averaged the outcome. The scripts performed various searches and comparisons between combinations of active cores, number of chunks, and number of particles.

5 Results

5.1 Performance Metrics

In order to provide pertinent performance comparisons, the command-line interface of the program was utilized in all testing to remove the rendering overhead. Each test's runtime statistics were averaged over five iterations. The reference test was one with 1000 particles and 1000 iterations. The

number of active cores and chunks varied per set of tests. In order to analyze runtime performance spread across multiple cores, Threadscope[2] was used.

5.2 Serial Performance

To start, the serial performance for the reference test was calculated. The test averaged `29.85` seconds. When analyzed in Threadscope, one can see the single core working hard.

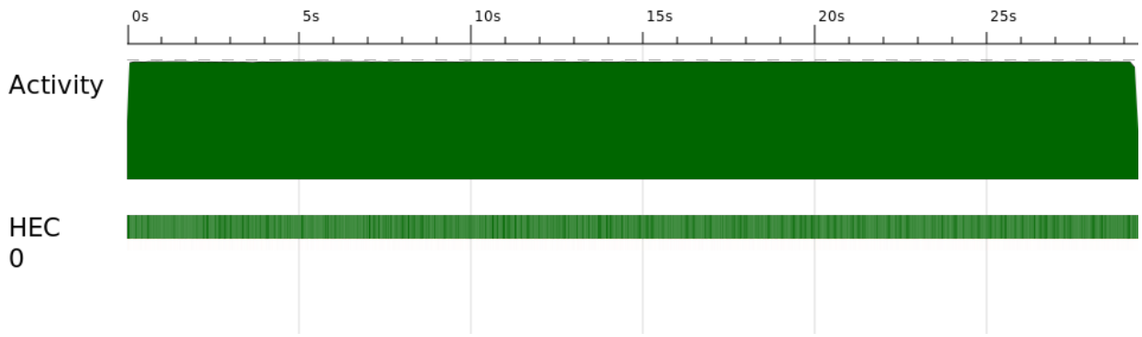

Figure 2: Serial performance for $N = 1000$ particles, $I = 1000$ iterations.

5.3 Naive `parList` or `parBuffer`

In the aforementioned `parList` implementation, the sparks are generated at the inception of this computation, thereby crippling performance and drowning the application in overhead. The `parBuffer` implementation (surprisingly) behaved the same. The tests below used two cores and easily show the difficulty in spawning so many sparks at once. The vast majority were garbage collected, and despite successfully saturating both threads for the entirety of the application, the performance was poor since the **user** time increased nearly twofold due to overhead.

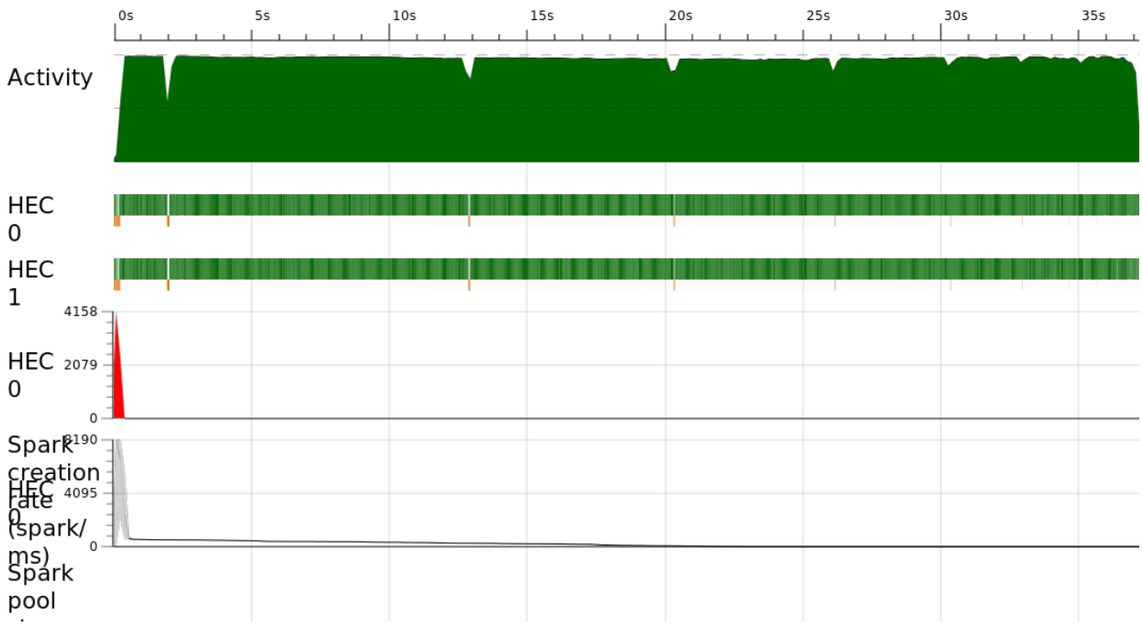

Figure 3: `parList`/`parBuffer` spark overflows.

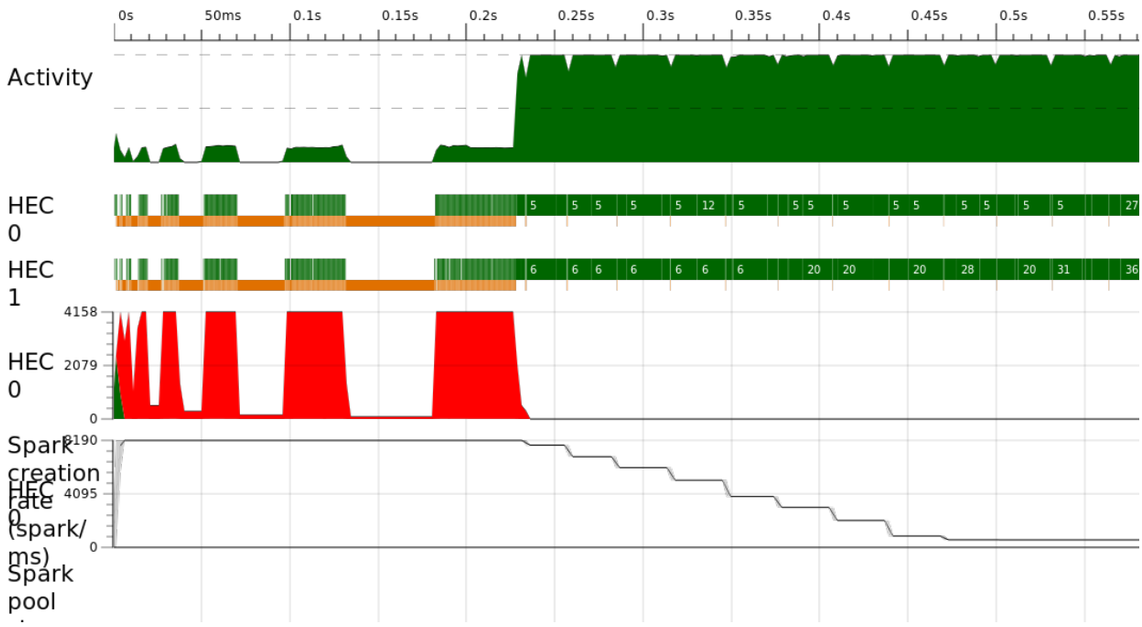

Figure 4: `parList/parBuffer` spark overflows.

5.4 Chunking Tests

Chunking drastically improved performance. In the $N = 2$ cores test below, nearly all sparks converted successfully:

```
SPARKS: 46000(45983 converted, 0 overflowed, 0 dud, 10 GC'd, 7 fizzled)
```

The sparks are spread throughout computation (despite shifting from one core to another), and the vast majority are converted. Furthermore, the spark pool is constantly around 30 sparks which is preventing starvation and allowing for the steady conversion rates on both threads.

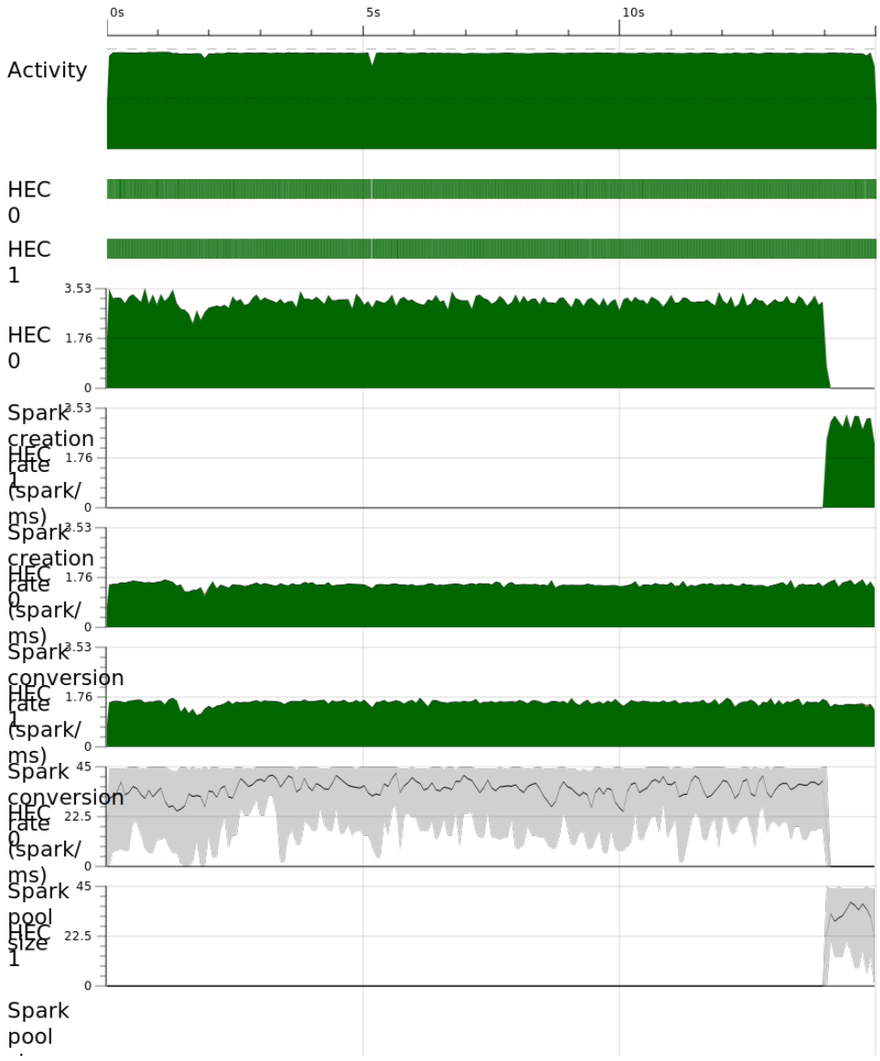

Figure 5: Excellent parallelism exhibited across two cores using chunking.

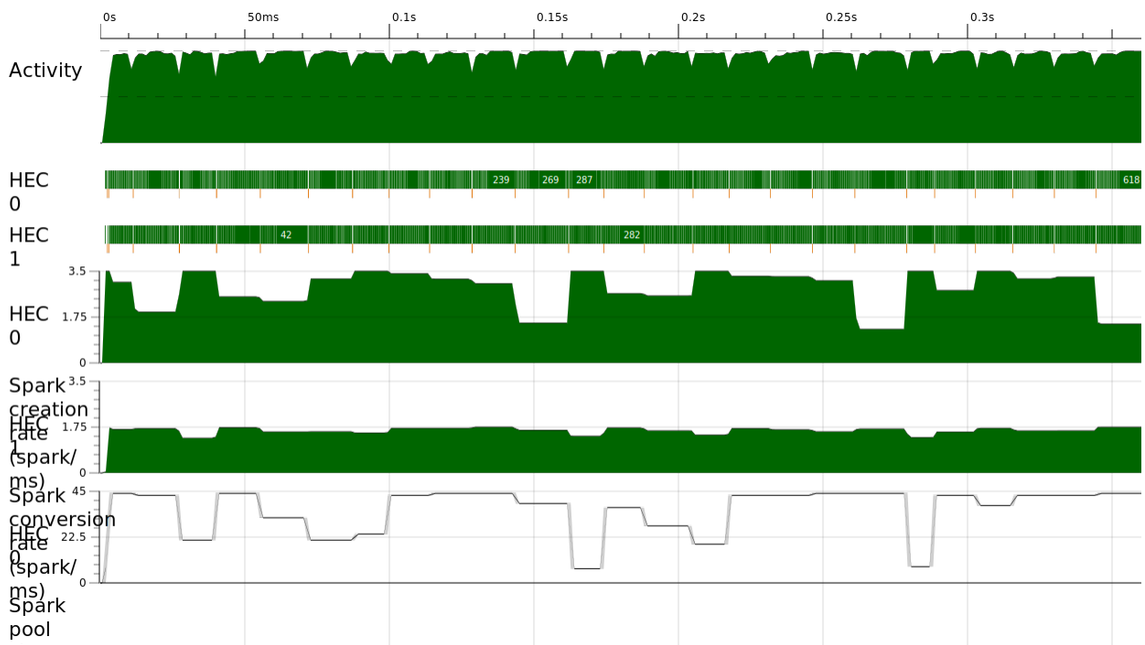

Figure 6: Zoom image showing nearly constant spark creation, conversion, and pooling.

In order to compare the effects of chunking on runtime, one of the Python test scripts ran each 5-test batch (for averaging) on all combinations of cores and chunks pairs. Below, one can see the decreasing runtime as the number of cores increases as well as the preference for more chunks.

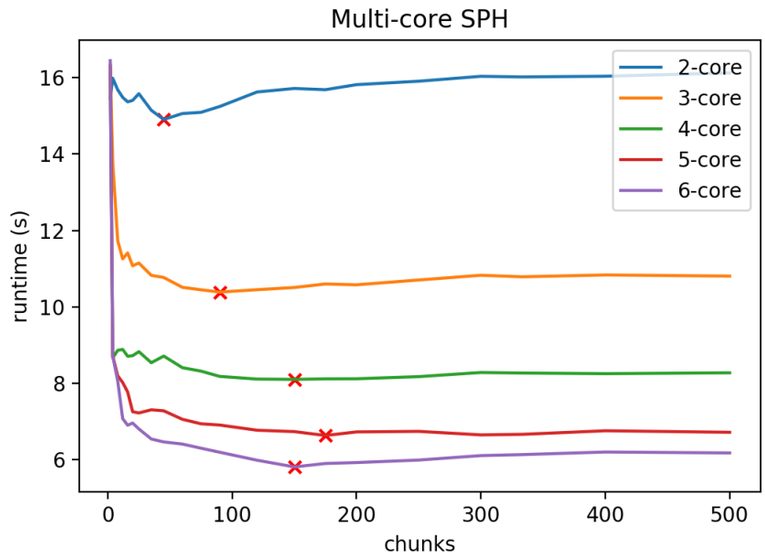

Figure 7: Chunking in multi-core SPH, red `X`'s marking the minumum runtimes for each set of tests.

Despite the ever-increasing speedup as seen above, an ever-increasing total runtime can be observed due to the additional overhead required in increasing parallelism. Below, the chunking in the same multi-core simulation is measured, except the runtime seen is the total (**user** + **sys**) runtime. This increasing runtime contributes to the diminishing returns (in speedup) as parallelism increases. Nonetheless, in the following section we will see that this is not to the detriment of the overall simulation and it does, in fact, perform exceedingly well up to 6 cores without hyperthreading.

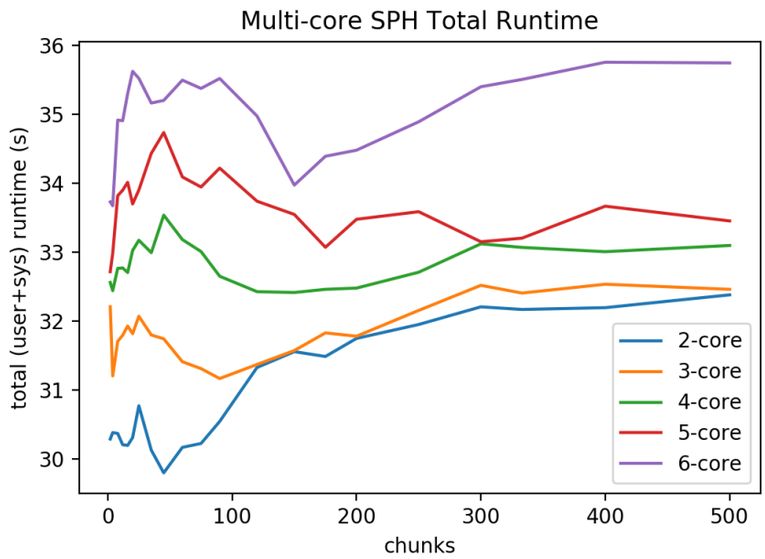

Figure 8: Chunking in multi-core SPH, showing the total runtime (user + system) increasing with parallelism and the number of chunks.

5.5 Multi-core Tests

The test machine is a desktop running Linux 5.3.0 atop an Intel i5-8600K Coffee Lake CPU with 6 cores (no hyperthreading) overclocked to 5.0 GHz. This proved to be a much more reliable source than local testing on a MacBook Pro. [2] After the above chunking analysis, the following simulations were the best combination of chunks and cores which stays within 15% of the theoretically possible speedup. Table 1 and Figures 9 and 10 display the best results achieved.

[2]Hyperthreading + the macOS scheduler and additional workload proved to be an unpredictable/unreliable beast. See 'Future Work' section.

Table 1: Best achieved performance for $N = 1000$ particles, $I = 1000$ iterations reference test.

Cores	Chunks	Time (s)	Speedup
1	1	29.8500	1.0000
2	45	14.9040	2.0028
3	90	10.3900	2.8730
4	150	8.1060	3.6825
5	175	6.6380	4.4968
6	150	5.8160	5.1324

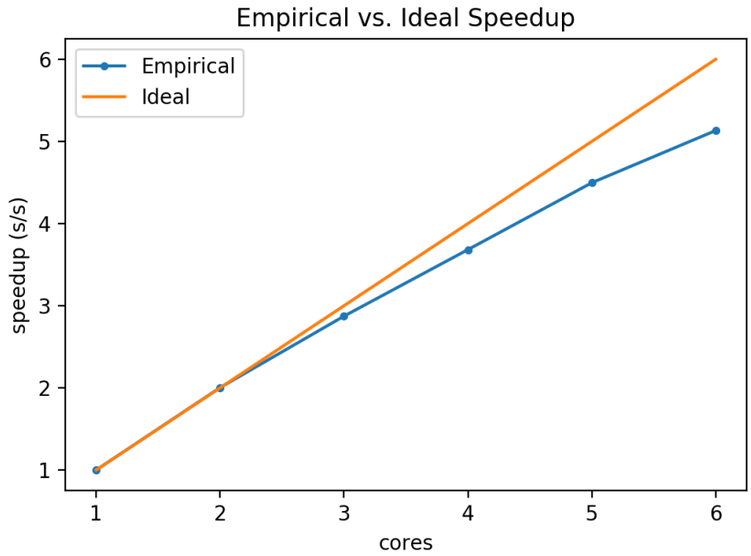

Figure 9: Best achieved performance for $N = 1000$ particles, $I = 1000$ iterations.

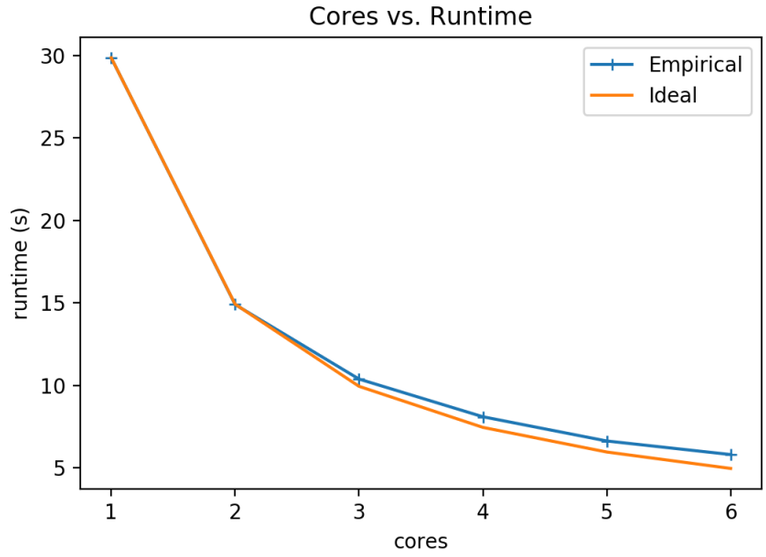

Figure 10: Best achieved performance for $N = 1000$ particles, $I = 1000$ iterations.

6 Conclusion

The ability for Haskell's robust runtime to provide a means of parallelizing the solver in only a handful of lines of code underscores the results of this study. The safety and clarity in a pure, functional language like Haskell have afforded an elegant solution to creating an SPH simulation which was easily parallelized to capture the capabilities of modern multi-core processors. Even with such simple parallelization, the simulation was able to stay within 15% of the maximum theoretically possible speedup on a six-core CPU.

7 Future Work

In the future, expanding the solver to include surface tension could provide more realistic simulation. Additionally, moving to a grid-based solver, and ignore particles outside of kernel's radius of support (yielding runtime complexity of $O(N)$) would provide a much more efficient algorithm and likely allow for scaling the simulation to many magnitudes larger than it is currently capable. Finally, further investigation on parallelization on architectures with hyperthreading is necessary. In local experiments on a MacBook Pro with hyperthreading, results were much less favorable, although there are a number of variables (like schedulers, workload) in play which would need to be measured/controlled.

The project source is both listed below and hosted on GitHub.[3]

[3] SPH on GitHub: https://github.com/schuermannator/sph

References

[1] Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly Media, Inc., Sebastopol, California, Aug 2013.

[2] Simon Marlow et. al. Threadscope. `https://hackage.haskell.org/package/threadscope`, 2009.

[3] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[4] Lucas Schuermann. Particle-based fluid simulation. `https://bigtheta.io/2016/05/23/particle-based-fluid-simulation.html`, May 2016.

[5] Lucas Schuermann. Implementing sph in 2d. `https://bigtheta.io/2017/07/08/implementing-sph-in-2d.html`, July 2017.

Program Listing

Main.hs

```haskell
module Main where
import Display (display, idle)
import Lib
import Solver (update, supdate)
import Graphics.UI.GLUT
import Linear.V2
import Control.DeepSeq
import System.Environment (getArgs, getProgName)
import System.Exit
import System.Random
import Data.IORef

-- default number of points/iterations
num_points :: Int
iter :: Int
chunks :: Int
num_points = 500
iter = 1000
chunks = 10

main :: IO ()
main = do
  args <- getArgs
  case args of
    -- just doing simple parsing for now
    [] -> guiMain num_points iter chunks
    ["-t"] -> cliMainPar num_points chunks iter
    ["-t", "-n", n, "-c", c] -> cliMainPar (read n) (read c) iter
    ["-t", "-n", n, "-i", iters] -> cliMainPar (read n) chunks (read iters)
    ["-t", "-i", iters] -> cliMainPar num_points chunks (read iters)
    ["-ts"] -> cliMainSeq num_points chunks iter
    ["-ts", "-n", n] -> cliMainSeq (read n) chunks iter
    ["-ts", "-n", n, "-i", iters] -> cliMainSeq (read n) chunks (read iters)
    ["-ts", "-i", iters] -> cliMainSeq num_points chunks (read iters)
    _ -> usage
  where cliMainPar nps ncs = (cliMain $ update ppc) nps ncs
          where ppc = nps `div` ncs
        cliMainSeq = cliMain supdate

usage :: IO ()
usage = do pn <- getProgName
           die $ "Usage: " ++ pn ++
             " [-ts] [-n <number particles>] [-i <number iterations>]" ++
             " [-c <number of chunks>]"

cliMain :: ([Particle] -> [Particle]) -> Int -> Int -> Int -> IO ()
cliMain f nps chks iters = do
  particles <- simInit nps iters chks
  let sol = iterate f particles !! iters
  sol `deepseq` putStrLn "Done."

guiMain :: Int -> Int -> Int -> IO ()
guiMain nps iters chks = do
  (_progName, _args) <- getArgsAndInitialize
  initialDisplayMode $= [DoubleBuffered] -- unused: RGBMode, WithDepthBuffer
```

```
  initialWindowSize $=
    Size (fromIntegral window_width) (fromIntegral window_height)
  _window <- createWindow "SPH"
  reshapeCallback $= Just reshape
  init_

  particles <- simInit nps iters chks
  ps <- newIORef $ particles
  iterRef <- newIORef $ (0 :: Int)

  displayCallback $= display ps
  idleCallback $= Just (idle (num_points `div` chunks) iters ps iterRef)
  keyboardMouseCallback $= Just (keyboard)
  mainLoop

simInit :: Int -> Int -> Int -> IO [Particle]
simInit nps iters chks = do
  gen <- getStdGen

  let jitters = randoms gen :: [Double]
      points = take nps $ initPoints jitters

  putStrLn $ "Starting " ++ show nps ++ " point simulation, "
    ++ show iters ++ " iterations (" ++ show chks ++ " chunks)..."
  return $ makeParticles points

reshape :: ReshapeCallback
reshape size = do
  viewport $= (Position 0 0, size)

init_ :: IO ()
init_ = do
  -- add "exit" menu entry
  attachMenu RightButton (Menu [MenuEntry "Exit" (exitWith ExitSuccess)])

keyboard :: KeyboardMouseCallback
keyboard key keyState _ _ = do
   postRedisplay Nothing
   case (key, keyState) of
      (Char 'q', Down) -> exitWith ExitSuccess
      (Char '\27', Down) -> exitWith ExitSuccess
      (_, _) -> return ()

initPoints :: [Double] -> [(Double, Double)]
initPoints jitters = zipWith (\(x, y) j -> (x+j, y)) makePoints jitters
  where makePoints :: [(Double, Double)]
        makePoints = [(x, y) | y <- [view_width/4, view_width/4
                                     + h..view_height-eps*2],
                               x <- [eps, eps+h..view_width/2]]

makeParticles :: [(Double, Double)] -> [Particle]
makeParticles pts = map maker pts
  where maker :: (Double, Double) -> Particle
        maker (x, y) = Particle (V2 x y) (V2 0 0) (V2 0 0) 0 0
```

Display.hs

```haskell
module Display (display, idle) where
import Solver (update)
import Lib
import Linear.V2
import Graphics.UI.GLUT
import Data.IORef
import System.Exit

drawPoints :: [(Float, Float)] -> IO ()
drawPoints ps = do
  let vertex3f x' y' z = vertex $ Vertex3 x' y' (z :: GLfloat)
      color3f r gr b = color $ Color3 r gr (b :: GLfloat)
  clear [ColorBuffer]
  matrixMode $= Projection
  loadIdentity
  ortho 0 view_width 0 view_height 0 1
  pointSize $= realToFrac h
  pointSmooth $= Enabled
  color3f 0.7188 0.9098 1
  renderPrimitive Points $
    mapM_ (\(x, y) -> vertex3f x y 0) ps

display :: IORef [Particle] -> DisplayCallback
display ps = do
  ps' <- get ps
  drawPoints (getPoints ps')
  swapBuffers

getPoints :: [Particle] -> [(Float, Float)]
getPoints ps = map getPoint ps

getPoint :: Particle -> (Float, Float)
getPoint (Particle p _ _ _ _) = parse p
  where parse (V2 x y) = (realToFrac x, realToFrac y)

idle :: Int -> Int -> IORef [Particle] -> IORef Int -> IdleCallback
idle ppc num_iters ps iter = do
  ps' <- get ps
  i <- get iter

  if i > num_iters
    then exitWith ExitSuccess
    else writeIORef iter (i+1)

  writeIORef ps (update ppc ps')
  postRedisplay Nothing
```

Lib.hs

```haskell
module Lib where
import Linear.V2
import Control.DeepSeq

data Particle = Particle {point :: V2 Double,
                          velocity :: V2 Double,
                          force :: V2 Double,
                          density :: Double,
                          pressure :: Double} deriving (Eq, Show)

instance NFData Particle where
  rnf (Particle p v f d pr) = rnf p `deepseq`
                              rnf v `deepseq`
                              rnf f `deepseq`
                              rnf d `deepseq`
                              rnf pr

g :: V2 Double
rest_dens :: Double
gas_const :: Double
h :: Double
hsq :: Double
mass :: Double
visc :: Double
dt :: Double
poly6 :: Double
spiky_grad :: Double
visc_lap :: Double
eps :: Double
bound_damping :: Double
view_width :: Double
view_height :: Double
window_width :: Int
window_height :: Int

g         = V2 0 (12000 * (-9.8)) -- 12000
rest_dens = 1000.0
gas_const = 2000.0
h         = 16.0     -- kernel radius
hsq       = h*h
mass      = 65.0     -- assume all particles have the same mass
visc      = 250.0    -- viscosity constant
dt        = 0.0009   -- 0.0008

poly6 = 315.0 / (65.0*pi*(h^(9 :: Int)))
spiky_grad = (-45.0) / (pi*h^(6 :: Int))
visc_lap = 45.0 / (pi*h^(6 :: Int))

-- simulation parameters
eps = h -- boundary epsilon
bound_damping = (-1) -- (-0.5)

window_width = 800;
window_height = 600;
view_width = 1.5 * fromIntegral window_width;
view_height = 1.5 * fromIntegral window_height;
```

Solver.hs

```haskell
module Solver (update, supdate) where
import Lib
import Linear.V2
import Linear.Metric (norm, quadrance)
import Control.Parallel.Strategies (using, parListChunk, rseq)

update :: Int -> [Particle] -> [Particle]
update partPerChunk ps = map integrate (forces (densityPressure ps))
                         `using` parListChunk partPerChunk rseq

-- sequential
supdate :: [Particle] -> [Particle]
supdate ps = map integrate (forces (densityPressure ps))

integrate :: Particle -> Particle
integrate (Particle p v f d pr) = enforceBC $ Particle updateP updateV f d pr
           where updateP = p + ((realToFrac dt) * updateV)
                 updateV = v + ((realToFrac dt * (f / (realToFrac d))))

enforceBC :: Particle -> Particle
enforceBC = bot . top . left . right
  where bot pa@(Particle p v f d pr)
          | check p = newp p v f d pr
          | otherwise = pa
          where check (V2 px _) = px - eps < 0.0
                newp (V2 _ py) (V2 vx vy) f' d' pr' =
                  Particle (V2 eps py) (V2 (vx * bound_damping) vy) f' d' pr'
        top pa@(Particle p v f d pr)
          | check p = newp p v f d pr
          | otherwise = pa
          where check (V2 px _) = px + eps > view_width
                newp (V2 _ py) (V2 vx vy) f' d' pr' =
                  Particle (V2 (view_width - eps) py)
                           (V2 (vx * bound_damping) vy) f' d' pr'
        left pa@(Particle p v f d pr)
          | check p = newp p v f d pr
          | otherwise = pa
          where check (V2 _ py) = py - eps < 0.0
                newp (V2 px _) (V2 vx vy) f' d' pr' =
                  Particle (V2 px eps) (V2 vx (vy * bound_damping)) f' d' pr'
        right pa@(Particle p v f d pr)
          | check p = newp p v f d pr
          | otherwise = pa
          where check (V2 _ py) = py + eps > view_height
                newp (V2 px _) (V2 vx vy) f' d' pr' =
                  Particle (V2 px (view_height - eps))
                           (V2 vx (vy * bound_damping)) f' d' pr'

densityPressure :: [Particle] -> [Particle]
densityPressure ps = map (calcDP ps) ps

calcDP :: [Particle] -> Particle -> Particle
calcDP ps (Particle pix piv pif _ _) = Particle pix piv pif newpid newpipr
  where newpid = sum $ map go ps
          where go :: Particle -> Double
                go (Particle pjx _ _ _ _)
                  | r2 < hsq  = mass * poly6 * (hsq-r2)^(3 :: Int)
                  | otherwise = 0
```

```
                  where r2 = quadrance (pjx - pix)
        newpipr = gas_const * (newpid - rest_dens)

fGravity :: Double -> V2 Double
fGravity dens = g * (realToFrac dens)

forces :: [Particle] -> [Particle]
forces ps = map (calcPV ps) ps

calcPV :: [Particle] -> Particle -> Particle
calcPV ps pi'@(Particle pix piv _ pid pipr) =
  Particle pix piv (fpress + fvisc + fgrav) pid pipr
  where
    fgrav = fGravity pid
    (fpress, fvisc) = folder $ map go ps
          where go :: Particle -> (V2 Double, V2 Double)
                go pj@(Particle pjx pjv _ pjd pjpr)
                  | pi' == pj = (V2 0 0, V2 0 0)
                  | r < h     = (fpress', fvisc')
                  | otherwise = (V2 0 0, V2 0 0)
                  where fpress' = ((-1)*(pjx - pix)/(realToFrac r)*
                                   (realToFrac mass)*(realToFrac (pipr+pjpr))/
                                   (realToFrac (2.0*pjd))*realToFrac spiky_grad
                                   *(realToFrac (h-r))^(2 :: Int))
                        fvisc' = (realToFrac visc)*(realToFrac mass)*(pjv-piv)/
                                 (realToFrac pjd)*realToFrac visc_lap*
                                 realToFrac (h-r)
                        r = norm (pjx - pix)
                folder = foldl (\(ax, ay) (x, y) -> (ax + x, ay + y))
                               ((V2 0 0), (V2 0 0))
```