2D Fluid Sim With Pony

Jaime Barillas
Ontario Tech University

***Abstract*—This report presents the results of implementing a simple 2D fluid simulator using Pony, an object oriented, actor model programming language for concurrent programs. It also provides an introduction to Pony in the context of building the fluid simulator. Focus is placed on the discussion of difficulties arising from Pony's implementation of the actor model and its use of *reference capabilities* to ensure *data-race* and *lock-free* programs.**

I. Preliminaries - Particle Based Fluid Simulation

There are two main approaches to simulating fluids: A grid based method, which focuses on the motion of fluid through grid cells, and a particle based method, which focuses on the motion of individual particles. The implementation accompanying this report takes the particle based approach, in particular, Smoothed Particle Hydrodynamics (SPH) techniques are used to approximate the properties of particles needed to calculate acceleration, velocity, and ultimately, position. This report does not cover SPH in any detail, only the broad. See [1] for good overview of the subject.

The main take-away from SPH is that many properties of fluids at specific point r can be calculated as a weighted sum of that same property of the neighbouring particles multiplied with a smoothing function W:

$$A(r) = \sum_j m_j \frac{A_j}{\rho_j} W(r - r_j, h) \tag{1}$$

Here ρ_j refers to the density of neighbouring particle j, r_j is the position of particle j, m_j is the mass of the particle, and h is a "support radius" (the maximum distance neighbouring particles can be to contribute to $A(r)$). This formula is used to calculate density (2), pressure forces (3), and viscosity forces (4), all of which are needed to create a working and convincing simulation.

$$A_{\rho(r)} = \sum_j m_j W(r - r_j, h) \tag{2}$$

$$A_{\text{pressure}}(r) = \sum_j m_j \frac{A_{\text{pressure},j}}{\rho_j} \nabla W(r - r_j, h) \tag{3}$$

$$A_{\text{viscosity}}(r) = \sum_j m_j \frac{A_{\text{viscosity},j}}{\rho_j} \nabla^2 W(r - r_j, h) \tag{4}$$

For (3) we require the gradient of the smoothing function W and for (4) we need to calculate the laplacian. It should be noted that each W in the above equations can be chosen seperately. The purpose of the smoothing functions to determine how much the neighbour particle contributes to the property at position r, further away neighbours contribute less. The exact smoothing functions chosen are those used in [2]. Note also how replacing A_ρ with density (ρ) causes it to cancel out in the right-hand side of equation (2).

The fluid simulator comes in two implementations: A C++ version using OpenMP to parallelize computations, and a Pony version using the fork_join library to try and match the way OpenMP distributes work across available threads.

II. The Pony Programming Language

We start with a quick introduction to the basics of Pony before moving on to discuss the implementation. Pony is an actor model based language that claims to be *data-race free* and *deadlock free*[3]. It uses its type system to statically ensure that a Pony program cannot have data races. Variables and their aliases can be annotated with different *reference capabilities* that determine how the data can be shared among actors. For example, the `val` capability allows data annotated with it to be shared amongst actors, whereas, the `iso` capability does not. So, how does Pony use actors and reference capabilities to model a concurrent program?

A. Actors & Classes

On top of being based on the actor model, Pony is also object oriented. Actors and classes represent two "modes" of concurrency in Pony. Classes provide soley synchronous execution of code (i.e. no concurrency) while actors allow for asynchronous execution. Given the bellow Pony program, all execution of the code in both classes will run sequentially.

```
class A
  fun say_hello(env: Env) => env.out.print("Hello A!")

class B
  fun say_hello(env: Env) => env.out.print("Hello B!")

actor Main
  new create(env: Env) =>
    let a = A
    let b = B
    a.say_hello(env)
    b.say_hello(env)
```

Listing 1: This pony program will *always* print `Hello A!` followed by `Hello B!`.

Take the following program instead. Classes `A` and `B` are replaced with actors. This program has the potential to run the `say_hello()` behaviours in any order.

```
actor A
  be say_hello(env: Env) => env.out.print("Hello A!")

actor B
  be say_hello(env: Env) => env.out.print("Hello B!")

actor Main
  new create(env: Env) =>
    let a = A
    let b = B
    a.say_hello(env)
    b.say_hello(env)
```

Listing 2: This pony program will could print `Hello A!` and `Hello B!` in any order.

This shows the main difference between actors and classes: Actors have *behaviours* that can execute asynchronously while classes do not. It is worth noting that code defined within an actor runs sequentially. It is never the case that two different behaviours of the same actor (or the same behaviour called twice) will run concurrently.

B. *Reference Capabilities*

Pony's approach to concurrency is to represent concurrent parts of a program with actors and their associated behaviours. State can be passed around actors but requires knowledge of reference capabilities. The "Why Pony?" section of their website[4] provides simple reasons, summarised here, as to why reference capabilities were added to Pony.

- *Mutable state is hard.* - In traditional concurrent programs, two or more threads of execution could potentially update a shared variable at the same time. Sharing mutable state between threads of execution can be hard to get right.
- *Immutable data is easy.* - If the state consists of data that cannot be changed, it is safe to share between threads of execution.
- *Isolated data is safe.* - If data is isolated (there is only one variable reference to it in existence), then it is safe to update. No other thread could potentially read or write to it.

There are a total of six reference capabilities in Pony. The table from the "Reference Capabilities > Reference Capabilities Matrix" section of the Pony tutorial[5] is recreated below:

TABLE I: Reference Capabilities

	Deny global read/write aliases	Deny global write aliases	Don't deny any global aliases
Deny local read/write aliases	**iso**		
Deny local write aliases	trn	**val**	
Don't deny any local aliases	ref	box	**tag**
	Mutable	Immutable	Opaque

The reference capabilities in bold are the *only* ones that allow sharing data across actors:

```
actor A
  // Keep local `tag` alias to msg. `tag` can alias all of iso, val,
  // and tag.
  var msg: String tag = String
  be set_msg_iso(s: String iso) => msg = s
  be set_msg_val(s: String val) => msg = s
  be set_msg_tag(s: String tag) => msg = s

actor Main
  new create(env: Env) => None
```

Listing 3: Sendable reference capabilities.

The above code shows the three *sendable* reference capabilities in action. Each of the behaviours for `A` can be sent data that is either annotated with the corresponding referrence capability. There is also a sort of heirarchy between reference capabilities in that data tagged with `iso` can be sent as `tag`, `ref` and `val` can be sent as `box`, and all capabilities can be sent as `tag`. Take the bellow code. It will not compile due to the use of `ref` as the reference capability attached to the `s` parameter:

```
actor A
  var msg: String tag = String
  be set_msg_ref(s: String ref) => msg = s

actor Main
  new create(env: Env) => None
```

Listing 4: Sendable reference capabilities.

Also, if we want to share `iso` data, but keep it as `iso` or *convert it to a non-*`tag` *reference capability*, we can do that by "consuming" the alias as we pass it along. This is necessary to maintain the guarantees of `iso` as the *only existing reference* to that piece of data:

```
actor A
  // Note that we now keep a local `String iso` field by consuming
  // the `s` parameter.
  var msg: String iso = String
  be set_msg_iso(s: String iso) => msg = consume s

actor Main
  new create(env: Env) =>
    let a = A
    // Strings are `ref` by default.
    // `recover iso ... end` tells Pony we want an `iso` string.
    let s: String iso = recover iso String end
    a.set_msg_iso(consume s)
```

Listing 5: Passing around `iso` data. Only one alias to `iso` data is allowed to exist so we have to `consume` aliases we no longer need. The `recover <refcap>` … `end` syntax tells pony to give us data with the `<refcap` capability instead of whatever the default is.

III. Implementations and Results

As mentioned before, there are two implementations. The C++ version uses OpenMP and its `parallel for` constructs to parallelize the fluid simulator. The Pony version makes use of actors for parallelization. We begin by taking a look the basic design.

A. *The Common Plan*

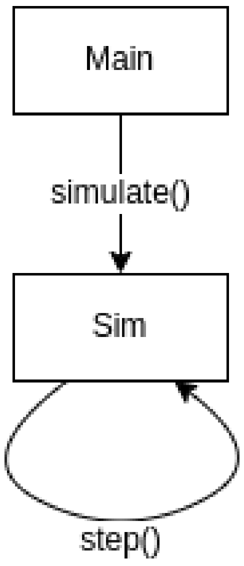
The overall approach to simulating particle based fluids is as follows:

1) Initialize the particles once at the beginning.
2) Iterate through each particle and:
   a) For every other particle:
      i) Test if its distance is within the support radius h.
      ii) If it is, use equation (2) to calculate its density contribution.
   b) Calculate pressure using the accumulated density value.
3) Iterate through each particle a second time:
   a) For every other particle:
      i) Test if its distance is within the support radius h.
      ii) If it is, use equations (3) and (4) to calculate both its pressure force and viscosity force contribution.
4) Perform one last iteration of each particle:
   a) Use the accumulated forces to calculate the particle's acceleration, velocity, and ultimately, position.
5) Draw each particle on screen.

The code for each implementation is available at: https://github.com/Jaime-Barillas/csci4060u-final

B. *The C++ Implementation*

The main function drives the app by calling the `simulate()` function of the `sim` class every frame. The `step()` function of the `sim` class is responsible for a single simulation step which goes through each of the iterations mentioned above. The `sim` class maintains a single array of particle objects throughout.



The code is a bit gnarly it will not be included in full for this report. Please visit the github repo to see it. The outer loops that iterate each particle are parallelized with the use of OpenMP's `parallel for` construct. The inner loops that iterate neighbours are not parallelized. Doing this allows OpenMP to split the work into batches that are processed simultaneously, which greatly speeds up the computation.

```
#ifdef ENABLE_PARALLELISM
  #pragma omp parallel for private(length, contribution, distance)
#endif
  for (int32_t i = 0; i < ps.size(); i++) {
    ps[i].density = 0.0f;
    ps[i].near_density = 0.0f;
    ps[i].pressure = 0.0f;
    ps[i].near_pressure = 0.0f;

    for (int32_t j = 0; j < ps.size(); j++) {
```

Listing 6: Using OpenMP's parallel for. cpp/simulator.cpp line ~65.

It is worth noting that the iteration of the neighbours is a nested loop over all particles. This results in $O(n^2)$ runtime per simulation step where n is the number of particles. This is by-far the most significant source of slowdown with higher particle counts (The sequential C++ implementation struggles to hit 60fps at 400 particles, the parallel version can reach 600 particles with 60fps or more).

C. *The Pony Implementation*

This one is particularly rough. In an attempt to batch the work similar to how the C++ program does it, Pony's "fork_join" library is used to batch the work across worker actors. The idea is to keep an array of particle objects within the `Sim` actor to update every frame. But, in order to use the fork_join library and parallelize the computations, we need to split this array up and distribute it across worker actors, then merge the batches back into one for drawing. Additionally, reviewing the approach outlined in "The Common Plan" above, each iteration needs access to the neighbouring particles to perform its computation.

This is where we run into difficulty with Pony's reference capabilities:

- The particle array and its batches need to be mutable and sharable. Use the `iso` capability.
- The particle array will be split up into batches for parallel computation spread across worker actors.
- Each worker actor needs a copy of the entire particle array for the calculations that require knowledge of the neighbouring particles.
  - So we need a reference to the original array, *but the array needs to be defined as `iso`* which doesn't allow multiple references.
  - Ergo, we need to make a copy of the particle array.

I will again defer to the github repo (link, click me) to see the code for the `Sim` actor and associated code. Unfortunately, the copy of the particle array needs to be done twice to account for the three iterations over the particles (the two that calculate forces and particle positions can be merged) due to how the latter two loops require density information calculated in the first loop. This *murders* performance and consumes memory *fast*, so don't run the Pony version for long. A different approach to try would be to not imitate how OpenMP batches work and instead represent every particle as its own actor.

D. Results

The numbers bellow were taken from on a machine with an i7 7700HQ processor (4 cores, 8 Threads):

	C++ (Sequential)	C++ (Parallel)	Pony (1 Thread)	Pony (4 Threads)
Particle Count	400	400	400	400
FPS (Simulation Step)	72	290	110	270
Speed Up	1x	4.02x	1.52x	3.75x (2.45x from Pony (1 Thread))

The performance of the Pony programs degrades over time due to the afore mentioned problems, so those numbers are rough estimates at best. They are taken as the number that the program settles around in the first second or so of execution. It is interesting to note that the parallel C++ version scales linearly in performance yet the Pony version does not. Another interesting data point is that the single threaded Pony version is faster than the C++ program. It is unfortunate that the current version of the Pony program is unstable so these numbers don't mean much.

References

[1] M. Müller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications," in *Proceedings of the 2003 ACM SIGGRAPH/ Eurographics symposium on Computer animation*, Goslar Germany, Germany: Eurographics Association, 2003, pp. 154–159.
[2] S. Barbara, P. Bucher, N. Chentanez, M. Müller, and M. Gross, "Virtual Reality Interactions and Physical Simulations (VRIPhys)," 2011.
[3] "What makes Pony different?." [Online]. Available: https://www.ponylang.io/discover/what-makes-pony-different/
[4] "Why Pony?." [Online]. Available: https://www.ponylang.io/discover/why-pony/#immutable-data-can-be-safely-shared
[5] "Pony Tutorial." [Online]. Available: https://tutorial.ponylang.io/