The Parallelization of complex computational problems

Capstone

Jan Eshun

June 10, 2022

Contents

The Parallelization of complex computational problems

0. Introduction

Modern computers are amazing at solving problems. With the right program, they can solve problems in fractions of a second that would take a human years to solve; and they are only getting better. However, there are some problems that traditional computers are incapable of solving efficiently. This paper will explore some of the problems which are practically impossible for traditional computers to solve, the optimizations and compromises we make, and how by changing the way computational problems are solved it is possible re-imagine what computing means.

1. Path-finding

Path-finding is a problem we encounter on a daily basis, from navigating cities to routing internet traffic, path-finding is an integral part of modern life. For the rest of this paper path-finding will refer to finding the shortest route that connects a cloud of dots (Figure 1.1). This is not the only application, but the principles apply to path-finding problems, albeit with minor tweaks.

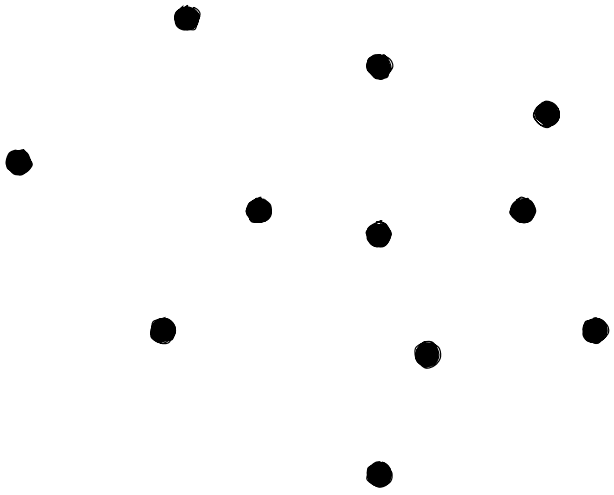

Figure 1.1

Path-finding is a deceptively expensive problem. The straightforward solution to problem solving is to iterate through every possible route and then pick the shortest. This method works when you only have a few options, but as you increase the number of dots, the number of possible solutions increases significantly. Specifically, the of solutions equals the number of dots minus 1 factorial divided by 2 ($\frac{ponts-1!}{2}$); this introduces a problem as the number of possible solutions quickly becomes unmanageable. For reference, at 10 points we hit 181440 possible solutions, at 20 the number of solutions explodes to over 60 quadrillion,

or in other words, a lot. The problem here lies with the need to find *THE* solution, not *a* solution. As soon as one removes the need to be able to mathematically prove that one has found the best option, one can turn their attention to finding a 'good enough' solution.

A good enough solution

There are many ways to find a good enough solution to path-finding problems, this paper will focus primarily on Ant Colony Optimization (ACO). Ant Colony Optimization, as the name implies, draws inspiration from the foraging behavior of ants to solve complex routes. Ants use many systems to path-find, often relying on pheromone trails to assist navigation. In our program we will use a system of weighted probabilities to help our ants navigate.

The first step is for the ant to travel to all of the points, picking each point based off a desirability score. The desirability score will be calculated by taking $\frac{1}{distance}^{n}$. The higher the *n* value, the more strongly the ant will prefer nearby points. The final decision will be made at random but probabilities are weighted by the desirability score. After each of the ants navigate all of the points we score each of their routes based on the total distance. Using this information we can create a pheromone trail, with shorter paths having stronger trails and longer paths having weaker trails. With the ants choosing their next destination based on the desirability score of each point multiplied by the strength of the pheromone trail, they will begin to naturally optimize more efficient pathways. This method of path-finding will often result in exponentially faster solutions, but it cannot definitively find *THE* best solution.

Figure 1.2

Iterative path finding compared to ACO

Number of Points	Possible Solutions	Iterative Time	ACO Time
12	19958400	1.67 seconds	0.05 seconds
13	239600800	23.5 seconds	0.05 seconds
20	60822550204416000	+600 years	0.06 seconds

(Lague, 2021)

2: Computational architecture

Problem solving methods like Ant Colony Optimization can drastically reduce the computational expense to solving these kind of problems, but we are still limited by the linear nature of the code. Each ant's decisions are calculated one at a time, and even with the best hardware and well-written code we can only

hope to simulate a few dozen ants at a time on consumer hardware, and only a few hundred on enterprise equipment. This is where true parallel compute comes into play. Modern computers have a Central Processing Unit (CPU) that is designed to execute a few complex instructions at once, but they also contain a variety of co-processors that specialize in other types of computation. The Graphics Processing Unit (GPU) is a co-processor that primarily focuses on rendering graphics to a display. However, the GPU is not limited to working with displays; GPU architecture is designed to execute simple commands at a massive parallel scale(Savage, 2009). Even entry level devices are capable of executing thousands of parallel instructions—perfect for rendering complex scenes on to millions of pixels. By repurposing the GPU's immense parallel compute ability we are able to simulate exponentially more ants than we could with a CPU.

3: Example of parallel compute

Thanks to the power of parallel compute, simulation of natural phenomenon has become much easier and more accurate. To demonstrate this paper will examine one example of this behavior in nutrient distribution in acellular organisms.

***Physarum Polycephalum*: It's alive!**

Physarum Polycephalum is an acellular slime mold or, myxomycete. Physarum Polycephalum looks like a mass of bright yellow interlaced tubes that slowly pulse and shift around its environment. While interesting to biologists for their unique biology, this is the slime that has hundreds of unique sexes (Zaugg, 2017), what really makes *P.polycephalum* stand out is its incredible path-finding and distribution abilities. In several experiments, *P.polycephalum* was able to map the highway networks of the United Kingdom as well as the railroad networks of the cites surrounding Tokyo with shocking accuracy (Tero et al., 2010); (ADAMATZKY & JONES, 2010). Mimicking the work of thousands of experienced engineers and urban planers, all while having only one cell, makes *P.Polycephalum* an excellent on which example to base path finding algorithms.

Simulating *P.Polycephalum*

The first step in simulating *P.Polycephalum* efficiently is to break its behavior down into simple rules that can be spread across thousands, or millions of individual agents. For *P.Polycephalum* each agent will contain 3 values: the x and y positions as well as the direction it is facing(ϕ).

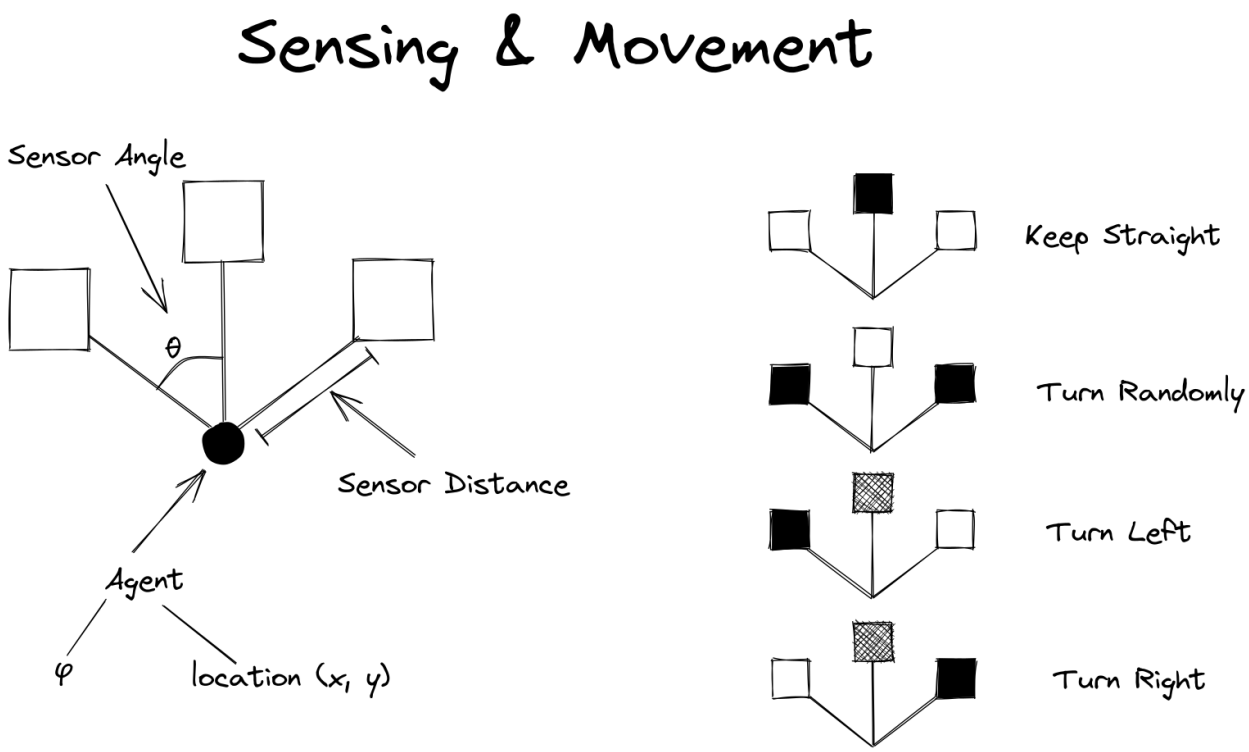

Figure 3.1

Every time step each agent will move in its current direction and then turn randomly, with the direction it turns weighted by the density of pheromones in 3 regions around it (Figure 3.1). Then it will record its current position to a 2D array, representing the pheromones (Figure 3.2).

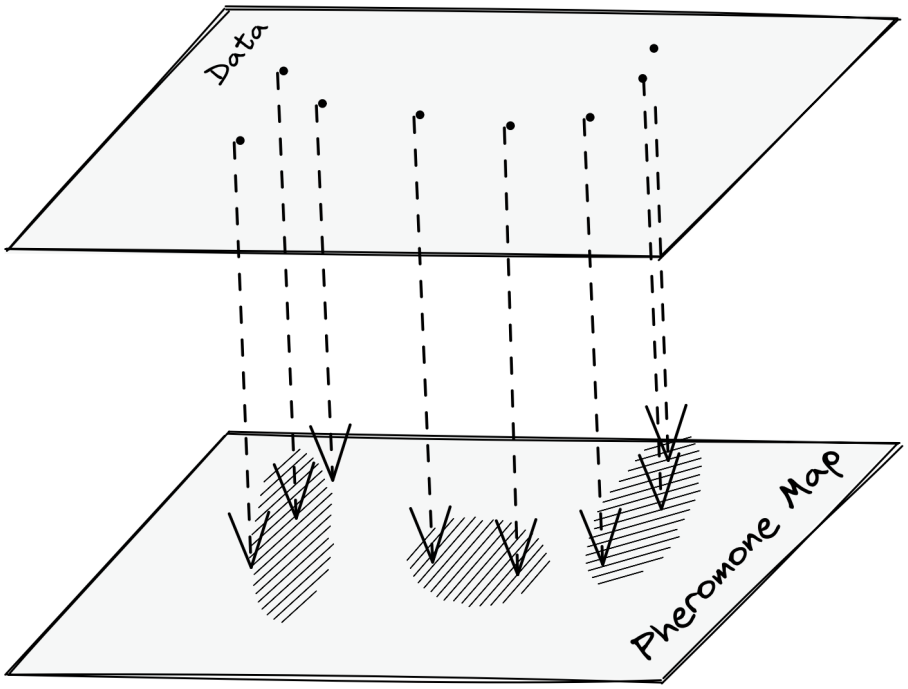

Figure 3.2

This 2d array will represent the pheromones. Once all of the agents have moved and recorded their position, the trail map will diffuse the pheromones by averaging the values across neighboring cells and reducing the value of each cell by a set amount (Figure 3.3). With these simple rules it is possible to create a reasonably accurate, high performance, 2D simulation of *P.Polycephalum*.

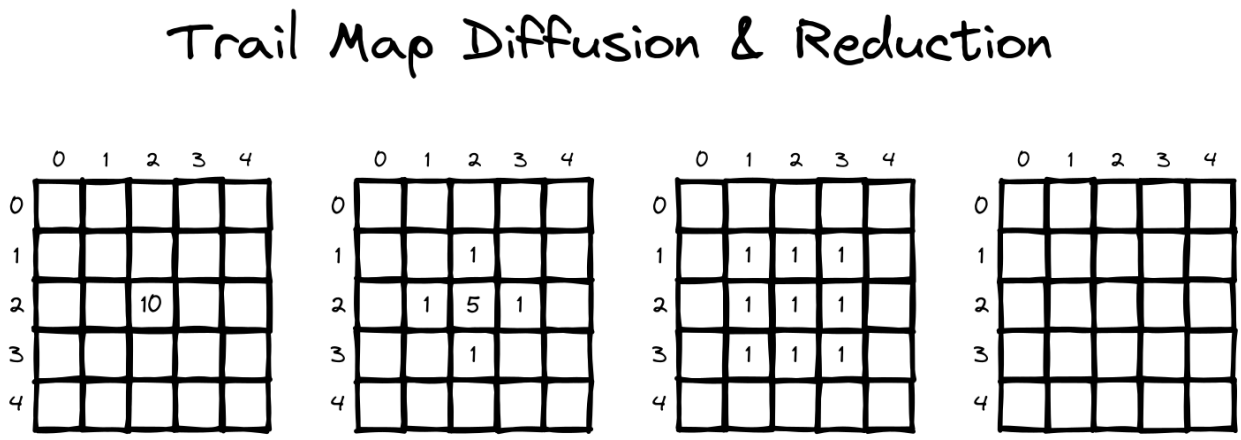

Figure 3.3

***P.Polycephalum* code breakdown:**

This is the code used to simulate *P.Polycephalum* using the Rust programming language and the Arrayfire

GPGPU compute library (Klabnik & Nichols, 2019); (Yalamanchili et al., 2015). Please note when operations are performed on arrays using this library they are done as parallel operations as opposed to traditional matrix manipulation.

After initializing Arrayfire the first step is to set the system variables.

```
const NUM_AGENTS: u64 = 1_000_000;
const WIDTH: u64 = 1000;
const HEIGHT: u64 = 1000;
const MARGIN: f32 = 5.0;
const SPEED: f32 = 0.5;
const THETA: f32 = 1.9548;
const SENSOR_DIST: f32 = 20.0;
const TURN_SPEED: f32 = 0.0524;
const DECAY_RATE: f32 = 0.75;
const DIFFUSE_RATE: f32 = 0.30;
const MINUS_PI: f32 = -3.1415;
const PI: f32 = 3.1415;
```

[1] Then the main loop sets the compute device, prints relevant device info, and calls our simulation.

```
fn main() {
    set_device(0);
    info();
    slime_sim();
}
```

The `slime_sim()` function starts by setting the dimensions of the simulation and creating the lists of agents, each initialized at the center of the screen facing in a random direction. The gaussian kernal is initialized here as well[2].

[1] These values will need to be tweaked as they are currently set as testing values.

[2] The Gaussin kernal is used in conjunction with a convolve function to apply a Gaussin Blur to the image, also known as a 2 dimentonal Weierstrass transform (Zayed, 1996, Chapter 18).

```
let sim_dims = dim4!(WIDTH, HEIGHT);
let height = constant::<f32>(HEIGHT as f32, dim4!(NUM_AGENTS));
let width = constant::<f32>(WIDTH as f32, dim4!(NUM_AGENTS));
let margin = constant::<f32>(MARGIN, dim4!(1));
let val_table = constant::<f32>(55.0, dim4!(NUM_AGENTS, 1));
let mut agent_map = constant::<f32>(HEIGHT as f32 / 2.0,
 ↪  Dim4::new(&[NUM_AGENTS, 3, 1, 1]));
let mut trail_map = constant::<f32>(0.0, big_dims);
let mut angle_table = randu::<f32>(dim4!(NUM_AGENTS));
angle_table = angle_table * 6.28 as f32;
set_col(&mut agent_map, &(&width/ 2 as f32), 1);
set_col(&mut agent_map, &angle_table, 2);
let gaus = gaussian_kernel(3, 3, DIFFUSE_RATE.into(), DIFFUSE_RATE.into());
```

Along with the agent arrays this first section creates arrays that are used to mutate positions later in the code.

Now that the environment has been setup the window and window loop can be created. This can be substituted with a for loop and saving each iteration as an image for recording.

```
let win = Window::new(1440, 1440, "P. Polycephalum".to_string());
while !win.is_closed() {...}
```

At the beginning of the loop values that need to be reset once per timestep are set. This includes the random number generator as well as the sensor positons for each agent[3].

[3]Normalise function::

```
fn normalise(a: &Array<f32>) -> Array<f32> {
a / (max_all(&abs(a)).0 as f32)
}
```

```rust
let mut rand_buffer = randu::<f32>(dim4!(NUM_AGENTS));
rand_buffer = normalise(&rand_buffer);
let seed_setter = counter as u64;
set_seed(seed_setter);
```

Within the same loop the sin and cos of each agent's (Φ) is used to calculate a vector allong each agent will move. This vector is then multiplied by the `SPEED` variable to denote how far each agent moves per time step. All of this is added to the agents current position and then clamped to the arena area minus the margin before being returned to the `agent_map`.

```rust
let ty = sin(&col(&agent_map, 2));
let tx = cos(&col(&agent_map, 2));
let mut cy = &ty * SPEED;
let mut cx = &tx * SPEED;

cy = &cy + &col(&agent_map, 1);
cx = &cx + &col(&agent_map, 0);

let cx = clamp::<f32, f32>(&cx, &MARGIN, &(&(WIDTH as f32) - &MARGIN), true);
let cy = clamp::<f32, f32>(&cy, &MARGIN, &(&(HEIGHT as f32) - &MARGIN), true);

set_col(&mut agent_map, &cx, 0);
set_col(&mut agent_map, &cy, 1);
```

The final step in moving the agents is to write their position to the 'trail_map'{.rust}. Because of the architecture of ArrayFire the positions must first be made into a sparse array before converting them dense array and adding to the trail map. To ensure data integreity the sparse map is first converted from COO format to CSR format then added to the 'trail_map'{.rust}.

```rust
let add_map = sparse(
    HEIGHT,
    WIDTH,
    &val_table,
```

```
    &cx.cast::<i32>(),
    &cy.cast::<i32>(),
    SparseFormat::COO,
);

let csr = sparse_convert_to(&add_map, SparseFormat::CSR);
let add_map_dense = sparse_to_dense(&csr);
trail_map = &trail_map + add_map_dense;

trail_map = &trail_map - DECAY_RATE as f32;
trail_map = convolve2(&trail_map, &gaus, ConvMode::DEFAULT,
 ↪  ConvDomain::SPATIAL);
trail_map = clamp::<f32, f32>(&trail_map, &0.0, &1.0, true);

win.draw_image(&trail_map, None);
```

The last few steps reduce the trail map and diffuse it as seen in Figure 3.3.

To allow for efficent indexing of the `trail_map` the 2D array is flattened into a 1D array(`flat_trails`) that is accessed with the `ix()` function[4]. Bewlow that the sensor positions are calcualted using the `agent_map`, the sensor offset (`THETA`), and the `SENSOR_DIST`.

```
let flat_trails = flat(&trail_map);



let left_sense = col(&agent_map, 2) + THETA;
let c_sense = col(&agent_map, 2);
let right_sense = col(&agent_map, 2) - THETA;
```

[4] ix indexing function:

```
fn ix(x: &Array<f32>, y: &Array<f32>) -> Array<f32>{
(x * WIDTH as f32) + y
}
```

```
let left_point_y = (sin(&left_sense) * SENSOR_DIST) + col(&agent_map, 1);
let left_point_x = (cos(&left_sense) * SENSOR_DIST) + col(&agent_map, 0);

let center_point_y = (sin(&c_sense) * SENSOR_DIST) + col(&agent_map, 1);
let center_point_x = (cos(&c_sense) * SENSOR_DIST) + col(&agent_map, 0);

let right_point_y = (sin(&right_sense) * SENSOR_DIST) + col(&agent_map, 1);
let right_point_x = (cos(&right_sense) * SENSOR_DIST) + col(&agent_map, 0);
```

These positons are then indexed to create three arrays containing the sensor vales for all of the agents.

```
let left_flat = ix(&left_point_x, &left_point_y);
let left_xy = view!(flat_trails[left_flat]);



let right_flat = ix(&right_point_x, &right_point_y);
let right_xy = view!(flat_trails[right_flat]);



let center_flat = ix(&center_point_x, &center_point_y);
let center_xy = view!(flat_trails[center_flat]);
```

Using combinations of greater than operators the agents decision is weighted.

```
let left_gt_right = gt(&left_xy, &right_xy, true);
let left_gt_center = gt(&left_xy, &center_xy, true);
let right_gt_center = gt(&right_xy, &center_xy, true);
let right_gt_left = gt(&right_xy, &left_xy, true);
let center_gt_left = gt(&center_xy, &left_xy, true);
let center_gt_right = gt(&center_xy, &right_xy, true);

let mut weigh_forward = and(&center_gt_left, &center_gt_right, true);
```

```
let mut weigh_left = and(&left_gt_center, &left_gt_right, true);
let mut weigh_right = and(&right_gt_center, &right_gt_left, true);
weigh_forward = bitnot(&or(&weigh_left, &weigh_right, true)) + &weigh_forward;
let mut turn_weight = (&weigh_left.cast::<f32>() * -1 * TURN_SPEED *
 ↪  &rand_buffer) + (&weigh_right.cast::<f32>() * 1 * TURN_SPEED * &rand_buffer)
 ↪  + (&weigh_forward.cast::<f32>() * TURN_SPEED * 2.0 * (&rand_buffer - 0.5));
turn_weight = &turn_weight.cast::<f32>() * PI;
```

After finally aquireing the `turn_weight` the agents are now able to turn. However, if any of the agents have strayed into the margin they will be turned arround 180°.

```
let cyo = ge(&cy, &(&height - &margin), true);
let cxo = ge(&cx, &(&width - &margin), true);
let ayo = le(&cy, &margin, true);
let axo = le(&cx, &margin, true);
let mut bounds_angle = (&cxo.cast::<f32>() + &cyo.cast::<f32>() +
 ↪  &axo.cast::<f32>() + &ayo.cast::<f32>());
bounds_angle = &bounds_angle * MINUS_PI;
bounds_angle = &col(&agent_map, 2) + turn_weight + bounds_angle ;
set_col(&mut agent_map, &bounds_angle, 2);
```

With the `agent_map` updated with the new angles the main loop can be repeated.

4: Limitations of Parallelization

For all of the advantages parallel compute brings to problem solving, there are situations where it offers minimal benefit, or even a hinderence. Even in the most heavily parallelized programs there is a least some component of the program that is limited to sequential compute[5]. To account for this limitation Amdahl's law is used to calculate the total compute time for complex parallelized programs (Amdahl, 1967). Knowing what percentage of the workload is limited to serial compute (p) it is trivial to estimate the uper limits of a potential speed boost provided by parallelization using this simplified version of Amdahl's equation: $\frac{1}{1-p}$ (Lieserson, 2008). Other limitations of parallel compute stem from the immense amount of data that large

[5] Any job that requires the result of the previous step to proced. Like building a skyscraper, the foundation must be built first, then the first floor etc.

scale parallel systems process and the sheer distance that data has to travel within these systems. However, with clever algorithmic and processor scheduling it is possible to avoid many of the pitfalls Amdahl warned of.

5: Return code 0

Computers affect every aspect of modern life and for over 40 years we were able to comfortably rely on Moore's Law to carry sequential computing to new heights (Moore, 1965). However, as we run into limitations imposed by the laws of physics we are forced to seek new areas to optimize for improved performance. Modern computers rely more and more on specialized co-processors beyond the GPU and while these processors are not designed purely for mass scale parallel compute the same principles used in writing safe, efficient parallel programs apply to systems built on these new heterogeneous computers.

Works Cited

ADAMATZKY, A., & JONES, J. (2010). ROAD PLANNING WITH SLIME MOULD: IF PHYSARUM BUILT MOTORWAYS IT WOULD ROUTE M6/M74 THROUGH NEWCASTLE. *International Journal of Bifurcation and Chaos*, *20*(10), 3065–3084. https://doi.org/10.1142/s0218127410027568

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference on - AFIPS '67 (Spring)*, 483–485. https://doi.org/10.1145/1465482.1465560

Arlauskas, N. (2021). Rust and OpenGL lessons. In *GitHub repository*. https://github.com/Nercury/rust-and-opengl-lessons; GitHub. https://nercury.github.io/rust/opengl/tutorial/2018/02/08/opengl-in-rust-from-scratch-00-setup.html

Gardner, M. (1970). Mathematical games. *Scientific American*, *223*(4), 120–123. https://doi.org/10.1038/scientificamerican1070-120

Jenson, S. (2019). *Physarum - Sage Jenson*. Self. https://cargocollective.com/sagejenson/physarum

Johnson, A., Kent, L., & Krieger, P. (2022). Vulkano. In *GitHub repository*. https://github.com/vulkano-rs/vulkano; GitHub. http://vulkano.rs/

Jones, J. (2010). Characteristics of Pattern Formation and Evolution in Approximations of Physarum Transport Networks. *Artificial Life*, *16*(2), 127–153. https://doi.org/10.1162/artl.2010.16.2.16202

Jorge Rodriguez, et all. (2022). Docs.gl. In *GitHub repository*. https://github.com/BSVino/docs.gl; GitHub. https://docs.gl/#

Klabnik, S., & Nichols, C. (2019). *Rust programming language* (p. 560). No Starch Press, Incorporated.

Lague, S. (2021). *Coding adventure: Ant and slime simulations*. Self. https://www.youtube.com/watch?v=X-iSQQgOd1A

Lieserson, C. E. (2008). https://www.cprogramming.com/parallelism.html

Mane, S. U., Lokare, P. S., & Gaikwad, H. R. (2022). *Overview and applications of GPGPU based parallel ant colony optimization*. arXiv. https://doi.org/10.48550/ARXIV.2203.11487

Mayes, K. (2020). *Vulkan tutorial (rust)*. https://kylemayes.github.io/vulkanalia/overview.html#api-concepts

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, *38*(8). https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf

Mordvintsev, A., Randazzo, E., Niklasson, E., & Levin, M. (2020). Growing neural cellular automata. *Distill*, *5*(2). https://doi.org/10.23915/distill.00023

Quirós, G. (2016a). *Are You Smarter Than A Slime Mold?* https://www.youtube.com/watch?v=K8HEDqoTPgk

Quirós, G. (2016b). This Pulsating Slime Mold Comes in Peace. In *KQED*. https://www.kqed.org/science/635319/this-pulsating-slime-mold-comes-in-peace

Reeves, W. T. (1983). Particle systems a technique for modeling a class of fuzzy objects. *ACM SIGGRAPH Computer Graphics*, *17*(3), 359–375. https://doi.org/10.1145/964967.801167

Reynolds, C. W. (1987). Flocks, herds and schools: A distributed behavioral model. *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, 25–34. https://doi.org/10.1145/37401.37406

Robinson, M. (2021). *(28) What are neural cellular automata? - YouTube*. Self. https://www.youtube.com/

Savage, J., Adam / Hyneman. (2009). *Mythbusters demo GPU versus CPU*. https://www.youtube.com/watch?v=-P28LKWTzrI

Shiffman, D. (2012). *The nature of code: Simulating natural systems with processing* (1st ed.).

sotrh. (2022). Learn WGPU. In *GitHub repository*. https://github.com/sotrh/learn-wgpu; GitHub. https://sotrh.github.io/learn-wgpu/

Tero, A., Takagi, S., Saigusa, T., Ito, K., Bebber, D. P., Fricker, M. D., Yumiki, K., Kobayashi, R., & Nakagaki, T. (2010). Rules for biologically inspired adaptive network design. *Science*, *327*(5964), 439–442. https://doi.org/10.1126/science.1177894

Yalamanchili, P., Arshad, U., Mohammed, Z., Garigipati, P., Entschev, P., Kloppenborg, B., Malcolm, J., & Melonakos, J. (2015). *ArrayFire - A high performance software library for parallel computing with an easy-to-use API*. AccelerEyes. https://github.com/arrayfire/arrayfire

Zaugg, J. (2017). *The 'blob': Paris zoo unveils unusual organism which can heal itself and has 720 sexes*. https://www.cnn.com/2019/10/17/europe/france-new-organism-zoo-intl-scli-scn-hnk/index.html

Zayed, A. I. (1996). *Handbook of function and generalized function transformations*. CRC Press.