# An Exploration and Comparison of the Effects of Data Structures on the Computational Time of Fluid Simulation

**To what extent does the time per frame of calculating fluid simulation differ due to the underlying data structure used?**

A Computer Science IB Extended Essay

3924 words

**Table of Contents**

## 1. Introduction

In 2018, Square Enix released the videogame *Shadow of the Tomb Raider*, which was immediately praised by reviewers for having stunning graphics (Metacritic, 2018). Below the surface, highly optimized real-time fluid simulation algorithms created the game's smoke and water. Nowadays, many games include interactive flowing fluids, a refreshing change from usual rigid physics. As a game developer, simulating fluids not only helps me create more realistic game environments, but also inspires me to create unique and interactive game mechanics.

Algorithms are measured by their computational efficiency, or the amount of computational resources (space and time) required depending on the input (Thomas, 2020). A more efficient and desirable algorithm would maintain visual quality while having higher frame rates and lower memory usage. After all, my games require aesthetically detailed and interactable real-time fluids. This essay focuses on one aspect of computational efficiency: computational time, measured by the time per update or time per frame.

In the field of computational fluid dynamics (CFD), computers simulate fluids by solving the Navier-Stokes equations on a set of small masses of fluids, called fluid parcels. The various fluid simulation algorithms can be categorized into grid-based Eulerian algorithms, where fluid flows through specific locations in space (or grid cells) as time passes, or particle-based Lagrangian algorithms, where individual fluid parcels (or particles) are traced through space and time (Schuermann, 2016). Grid-based techniques are more numerically accurate, but only operate in limited grid spaces. Particle-based techniques better model fluid splashes and are unrestricted in space, making them more suited for video games. Optimized techniques rely on the Eulerian or Lagrangian framework but use other data structures like trees. Compared to brute force searching for a list of particles, an underlying grid significantly decreases computational

time as knowing the indexes of relevant neighboring fluid parcels allows for faster data access and manipulation.

## 2. Background Information

### 2.1 Fluid Physics

A fluid is any liquid or gas that continuously changes in shape and is subject to stress, allowing it to flow. There are several types, including the inviscid ideal fluid, Newtonian fluids which have constant viscosity, and non-Newtonian fluids with varied viscosity (The Editors of Encyclopaedia Britannica, 2021). Fluids can be compressible, with variable density and volume, or incompressible.

Incompressible fluid motion is precisely described by the Navier-Stokes equations, with the first for the conservation of momentum and the second for incompressibility:

$$\rho\left(\frac{\partial u}{\partial t} + u \cdot \Delta u\right) = -\nabla p + \eta \nabla^2 u + \rho g$$

$$\nabla \cdot u = 0$$

$\rho$ is the fluid density, $u$ is the velocity vector field, $t$ is the time, $p$ is the pressure, $\eta$ is the viscosity, and $g$ is the external force (or gravity) vector. The equation roughly states that net force on a particle is the sum of the pressure force, viscosity force, and external force (Schuermann, 2016).

### 2.2 Previous Fluid Simulation Work

Fluid flow governed by the Navier-Stokes equations were first modelled on a computer by the T-3 group at the Los Alamos National Lab (Harlow & Welch, 1965). In the field of computer graphics, the Navier-Stokes equations were first solved by Foster & Mexatas (1996). A

few decades later, Stam (1999) introduced a stable and highly efficient grid-based Eulerian fluid simulation, consisting of a semi-Lagrangian advection technique. Fluid simulation was also computed with the Smoothed Particle Hydrodynamics Lagrangian method, first introduced for solving astronomical problems by Gingold and Monaghan (1977), and then remodeled by Müller, Charypar, and Gross (2003) to solve the Navier-Stokes equations. Lamorlette et al. have also used the Navier-Stokes equations to model fire, clouds, particle explosions, variable viscosity, bubbles and surface tension, splash and foam, etc. (as cited in Losasso, Gibou, & Fedkiw, 2004).

## 2.3 Grid-Based Algorithms

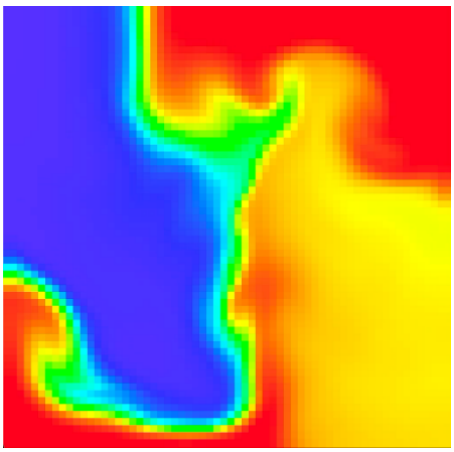


Figure 1: MAC Eulerian simulation of two colliding fluid bodies (blue and yellow). Color depicts density.

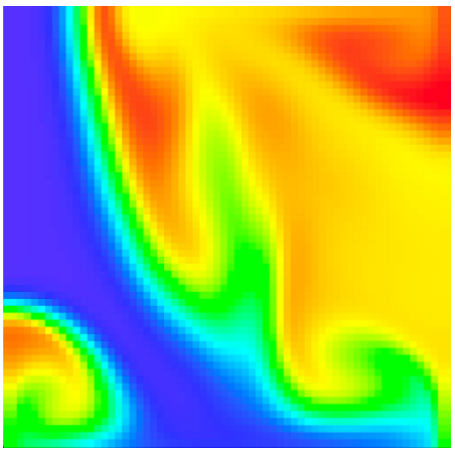


Figure 2: MAC grid after diffusion and advection for several frames

The marker-and-cell (MAC) method, developed by Harlow and Welch (1965), models fluid by discretizing fluid into "marker" particles on a velocity field, representing them with a grid of cells. It is widely used in grid-based fluid simulation. This essay analyzes the unconditionally stable MAC Eulerian incompressible fluid simulation algorithm by Stam (2003), using Ash's (2006) simplified code implementation. Fluid is simulated from an Eulerian viewpoint on a MAC grid, representing the fluid density and velocity at discrete points in space

(Figures 1 and 2). Boundary grid cells reflect the x- and y-components of adjacent fluid velocities and duplicate the adjacent fluid densities. For each update, the order is (1) diffuse velocity vectors, (2) enforce incompressibility, (3) advect velocity vectors over the velocity vector field, (4) enforce incompressibility again, (5) diffuse density, and (6) advect density over the velocity vector field.

For the diffusion of viscous fluid, each grid cell's field quantity (density or velocity) decreases from outflow and increases from inflow from the four adjacent neighbors. Those new quantities are solved using an iterative method called Gauss-Seidel relaxation, such that diffusing backwards in time yields the starting quantities. There is a safe constant iteration count of ten.

By eliminating inward or outward flows in the velocity field, conservation of mass and thus fluid incompressibility is enforced. This also allows for swirly fluid vortices. Applying Hodge decomposition, the mass-conserving field is the difference of the current velocity field and a gradient field. This gradient field is calculated by using Gauss-Seidel relaxation for every grid cell to solve a partial differential equation called the Poisson equation.

Advection, the transfer of matter by the fluid's flow, is calculated through Stam's groundbreaking Semi-Lagrangian model. Each grid cell's fluid density is modelled as a Lagrangian particle in the center and traced through the velocity field one timestep backwards to find its previous position. At this previous position, the four grid cell neighbors' densities are accessed in constant time, and linearly interpolated to calculate the original grid cell's new density value. This requires accessing two grid data structures, one previous density field and one for the new density values. Velocity advection, where the "velocity field is moved along itself" (Stam, 2003, p. 8), is likewise calculated by replacing fluid density with velocity.

During an update, each grid cell mathematically calculates the indexes of a constant number of other grid cells, allowing $O(1)$ constant access time. As this runs for each grid cell, the algorithm theoretically updates in linear time, based on the amount of grid cells.

## 2.4 Optimized Grid-Based Algorithm

The reliance of grid-based fluid simulators on an underlying grid covering the entire simulation space is a disadvantage. Detailed fluids require denser grids, and cells with zero fluid density must still be updated, costing both memory and time. An optimization is to use a quadtree or octree data structure to replace the grid (Losasso, Gibou, & Fedkiw, 2004). A quadtree is a tree with four nodes, used in 2D simulation, while octrees for 3D have eight nodes. For quadtrees, each node's children are subtrees and cover a fourth of the area that the parent node covers. Only leaf nodes (nodes without children) contain field quantities like density and velocity.

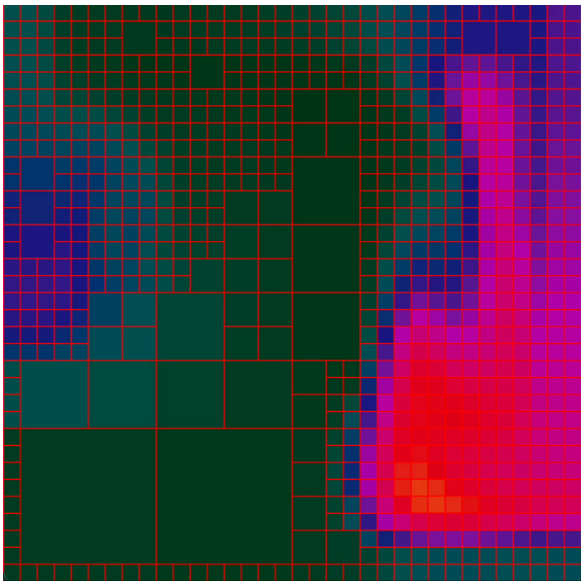


Figure 3: Quadtree-optimized Eulerian simulation with adaptive refinement

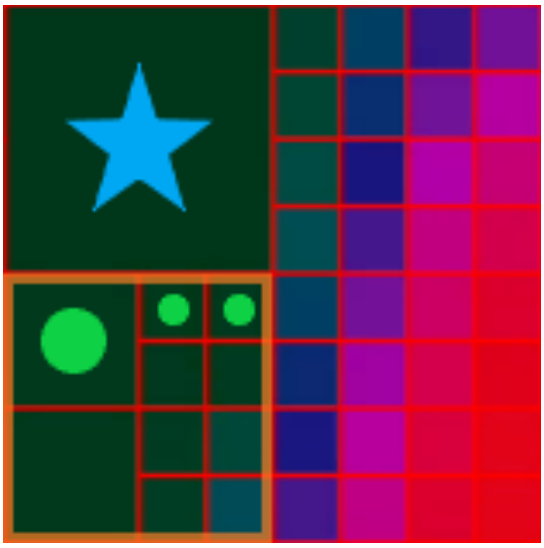


Figure 4: South neighbor-finding from the source node starred in blue. Orange outlines the neighbor node of equal height, and green dots the final leaf neighbors.

My algorithm roughly follows Shi and Yu's (2004) approach, and basically replaces the MAC grid of Stam's (2003) algorithm with a quadtree that adaptively refines or coarsens depending on regional fluid visual complexity (Figure 3).

First, for each node whose children are all leaf nodes, a crude check for fluid visual uniformity is performed: if the maximum difference in fluid density of the node's children is below 0.4 units, then the node is coarsened, meaning its own field quantities become averages of its children's quantities, and the children are deleted.

Then, node neighbors were found through Samet's (1989) algorithm and stored (as demonstrated in Figure 4). For each source leaf node, and for each of the four directions, the algorithm recursively traverses up the tree until the current node contains a neighbor node. The algorithm then traverses down the tree until the current node is a leaf node greater or equal in height to the source node. If the current node is a non-leaf node of equal height, then all its leaf children adjacent to the source node are recursively searched. If a neighbor is beyond the spatial bounds of the root node, then a neighbor "border node" is searched from a separate border nodes

array (as shown on the edges in Figure 3). I created border nodes to efficiently solve edge cases, avoiding unnecessary quadtree refinement along its borders.

Next, if the maximum difference of density of a leaf node's neighbors is more than 40 units, then that node is refined (by copying parent quantities to new children).

```
private interface CallableLeaf {
    void f(Quad q);
}

private static void forAll(Quad q, CallableLeaf leafInterface) {
    if (q.isLeaf) {
        leafInterface.f(q);
    } else {
        for(Quad child : q.children) {
            forAll(child, leafInterface);
        }
    }
}
```

Figure 5: Java higher-order function *forAll* that recursively loops through all quadtree leaf nodes and calls some function $f$ on those nodes

```
public Quad search(int i, int j) {
    if (isLeaf) return this;

    int midI = (iStart + iEnd) / 2;
    int midJ = (jStart + jEnd) / 2;
    if (j < midJ) {
        if (i < midI) {
            return children[0].search(i, j);
        } else {
            return children[1].search(i, j);
        }
    } else {
        if (i < midI) {
            return children[2].search(i, j);
        } else {
            return children[3].search(i, j);
        }
    }
}
```

Figure 6: Function *search* that recursively searches in the child node the point $(i, j)$ belongs in, until it reaches a leaf node

The functions of boundary setting, diffusion, projection, and advection were completely modelled off Stam's (2003) Eulerian algorithm. In these functions, instead of a grid, all leaf nodes are recursively looped through (using the higher-order function in Figure 5), and their values were subsequently set based on the values of precomputed neighbor nodes. Semi-Lagrangian advection was performed by executing a recursive search function in $O(\log n)$ time (Figure 6) to back-trace fluid parcels.

Let $n$ be the maximum number of highest resolution nodes. For each node, neighbor-finding and node searching both take $O(\log n)$ time. In the worst-case, both are performed for $n$ leaf nodes, leading to $O(n \log n)$ total time complexity. Looping through all leaf nodes only

takes $O(n)$ time and thus is ignored. $O(n \log n)$ is worse than the linear time complexity of the original Eulerian algorithm, challenging the "optimization" of this quadtree algorithm. However, depending on the simulation situation, this algorithm's average computational time may still drastically improve.

## 2.5 Particle-Based Algorithm

One of the most prominent particle-based algorithms is Smoothed Particle Hydrodynamics (SPH). A particle system represents discretized fluid parcels. Each particle field quantity is calculated based on its neighbors within a certain support radius $H$, using the SPH rule that "a scalar quantity A is interpolated at location r by a weighted sum of contributions from all particles" (Müller, Charypar, & Gross, 2003, p. 155). The SPH rule equation is

$$A_S(r) = \sum_j m_j \frac{A_j}{\rho_j} W(r - r_j, H)$$

Where each particle of index $j$ has mass $m_j$, position $r_j$, density $\rho_j$, and field quantity $A_j$. $W(r, h)$ is the smoothing kernel that gives weights to neighboring particles within support radius $h$, such that the resulting weighted average quantity from this equation matches physical fluid particle interaction.

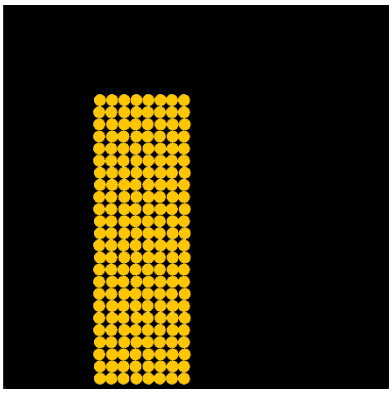


Figure 7: Lagrangian simulation with gravity on an initial block of fluid particles

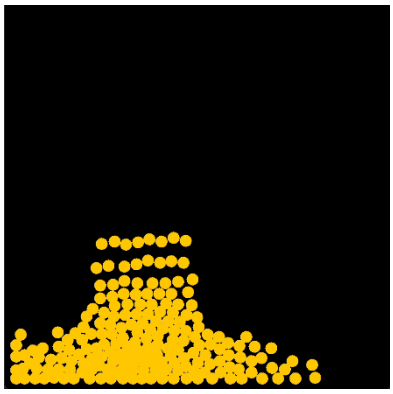


Figure 8: Fluid particles break formation as forces are applied to every particle

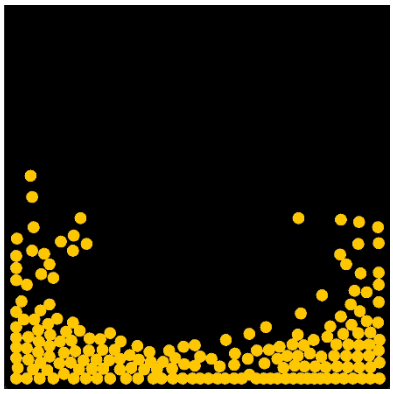


Figure 9: Fluid particles collide with the edges of simulation space and splash

Using the Navier-Stokes equations, the SPH rule, and other physics equations, equations for the density, acceleration, pressure force, and viscous force are derived. To maintain code simplicity, surface tension is ignored. First, the algorithm brute forces neighbor finding by looping through each particle $i$, and then checking if particle $j$ is a neighbor within radius $H$ in a nested loop. Neighbor density contributions are summed to calculate pressure. After, density and pressure of neighboring particles are used to calculate the forces acting on each particle. Finally, the algorithm loops through each particle and accelerates and moves them based on those forces (using forward Euler integration). Having nested loops means the theoretical time complexity is $O(n^2)$. SPH simulation over multiple updates is displayed in Figures 7-9.

## 2.6 Optimized Particle-Based Algorithms

That naïve nested loop neighbor finding algorithm can be optimized to reduce its $O(n^2)$ time complexity. Lagrangian particles can be represented by a grid with cell size $H$, where each cell is a list of all the particles within that square location (ideally, there is only one particle per grid cell). Each particle's position directly maps to its grid index (Figure 10). For a particle in a grid cell, its neighbors must be within an $H$ radius, meaning they only exist in adjacent cells.

Assuming one particle exists per cell, neighbor-finding would thus take constant time, causing $O(n)$ total time complexity. This optimization was originally proposed by Desbrun and Gascuel (1996).

```
private static ArrayList<Integer>[][] grid;
private static ArrayList<Pair<Particle, Pair<Integer, Integer>>> particles;

private static void addParticle(float posX, float posY, float vX, float vY)
{
    Particle particle = new Particle(posX, posY, vX, vY);
    int i = (int)(posY / H) + 1; //+ 1 to account for empty boundary grid cells
    int j = (int)(posX / H) + 1;
    Pair<Integer, Integer> idx = new Pair<Integer, Integer>(i, j);
    particles.add(new Pair<Particle, Pair<Integer, Integer>>(particle, idx));
    grid[i][j].add(particles.size() - 1);
}
```

Figure 10: 2D *grid* array and *particles* ArrayList. They are used in function *addParticle*, which maps a particle's (posX, posY) coordinate to a corresponding grid index and adds that particle to the end of that grid cell's ArrayList.

I personally implemented this algorithm (Figure 10). To accomodate for particles clustering in the same grid cell under high pressures, the grid is represented as a 2D array of ArrayLists. The grid stores the indexes of particles which are stored in another ArrayList. Looping over a separate list of particles allows skipping over grid cells without particles stored within. Each particle class is paired with $x$ and $y$ grid indexes. The pair class is a generic Java class supporting two separate data types and values.

Other optimizations exist for Lagrangian fluid simulation. A spatial hash table allows particles of any position (beyond the range of a set grid) to be stored, and simultaneously saves memory as data is only stored for occupied cells. Spatial hashing can sometimes be inefficient if multiple positions are hashed to the same index. Another optimization is to adaptively refine particles based on the regional fluid flow complexity, similar to the quadtree optimization (Adams, Pauly, Keiser, & Guibas, 2007). Fluid particles are split into smaller particles if the flow

is dynamic, or merged into one big particle if the flow is slow and steady. This effectively reduces the amount of particles, which quickens computation while maintaining visual quality.

### 2.7 Other Algorithms

Many other grid- and particle-based algorithms exist to simulate fluids. Hybrid algorithms compute Lagrangian particles on an Eulerian grid, gaining the advantages of both particle- and grid-based methods. These include particle-in-cell, fluid-implicit-particle, and material point method. Deep learning convolutional neural networks can also learn to simulate fluids.

## 3. Experiment Methodology

### 3.1 The Algorithms Used

Four fluid simulation algorithms are measured and compared:

1. Ash's (2006) implementation of Stam's (2003) grid-based MAC Eulerian algorithm
2. Quadtree-optimized Eulerian algorithm, which I based off algorithm (1)
3. Schuermann's (2017) implementation of Müller, Charypar, and Gross's (2003) particle-based SPH Lagrangian algorithm
4. Grid-optimized Lagrangian algorithm, which I based off algorithm (3)

I implemented all algorithms in Java with minor changes to the original code. In algorithm (3), Schuermann uses an external C++ library for vector math. I instead utilized my own Java Vector class, which contained the functions of vector addition, subtraction, normalization, dot product, and magnitude.

## 3.2 Independent and Dependent Variables

The dependent variable is the time per frame, or the time elapsed after one simulation update. Specifically, the update function computes the forces, velocities, densities, and positions of each fluid parcel. To measure this dependent variable, Java's built-in `System.nanoTime()` is called right before and after calling the main update function and subtracted to obtain a long-type elapsed time variable. For each independent variable manipulation, there was three trials, and each trial consisted of ten frames (updates). The time per frame was averaged over these ten frames and subsequently printed to the console.

The independent variable $N$ is the amount of fluid parcels per simulation. In the Eulerian algorithm, $N$ is the number of cells in a square grid, meaning the length of the grid is $\sqrt{N}$ cells. Using a square grid allowed for a more balanced fluid space and simpler code. The density and velocity components of each grid cell was initialized randomly within a certain range. The quadtree-optimized Eulerian algorithm was initialized with the highest resolution of $N$ total leaf nodes, set randomly to ensure more varied adaptive refinement as to obtain more holistic data.

In particle-based algorithms, $N$ is the number of particles. Particles move within a constant square space of length $L$. During initialization, $N$ particles are spawned at random locations within this space. In the grid-optimized Lagrangian algorithm, this meant particles would more uniformly distribute among grid cells, reducing the effect of initial clustering on computational time. Additionally, the support radius $H$ is set to $\left(\frac{1}{2}\right)\frac{L}{\sqrt{N}}$, meaning for a 1D space, $\sqrt{N}$ particles cover a length of $\frac{L}{2}$, so $N$ particles only cover a fourth of the 2D space. Doing so ensures particles can fit and move within the space. Each particle's velocity was also randomized within the same range as those in Eulerian simulations to produce comparable data.

### 3.3 Controlled Variables

Each simulation had ten updates or ten frames. To collect data focusing on comparing time per frame, fluid parcels were not rendered, which eliminated many unrelated rendering efficiency variables such as visual shape of fluid parcels, computation of pixel colors, and anti-aliasing.

The code was compiled using Javac and run on a 2020 13-inch MacBook Pro computer with an Apple M1 Chip and 8 GB of memory. During the experiment, the only applications running were Excel and IntelliJ IDEA Community Edition 2021. Checking Mac's Activity Monitor (which displayed all running processes on the computer) ensured that no other major background processes were running. Thus, the available CPU power and memory remained as constant as possible, ensuring maximum precision of the experimental data.

## 4. Experiment Results

### 4.1 Data Table

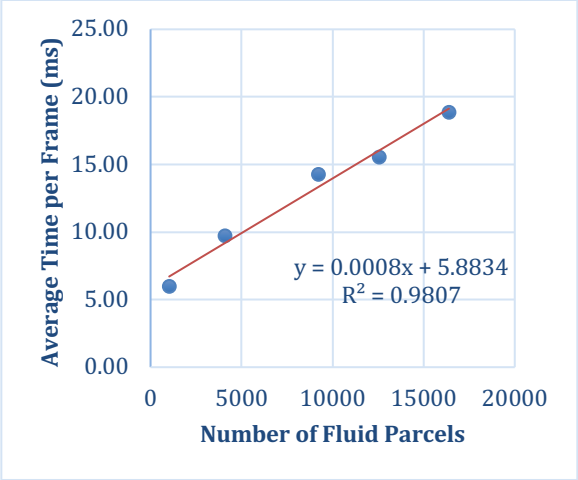
The table below lists the data for all four algorithms. The average time per frame column uses milliseconds (converted from raw nanosecond data) for easier readability, and displays values to the hundredth place to maintain high accuracy. An extra column of frame rate (inverse of time per frame, converted to seconds) is provided to intuitively measure the time per frame.

| Algorithm | Number of Fluid Parcels | Average Time per Frame (ms) | Frame Rate (Hz) |
|---|---|---|---|
| Eulerian | 1024 | 5.96 | 168 |
| | 4096 | 9.74 | 103 |
| | 9216 | 14.26 | 70 |
| | 12544 | 15.56 | 64 |
| | 16384 | 18.88 | 53 |
| Quadtree-Optimized Eulerian | 1024 | 27.82 | 36 |
| | 4096 | 49.07 | 20 |
| | 9216 | 79.41 | 13 |
| | 12544 | 102.12 | 10 |
| | 16384 | 124.62 | 8 |
| Lagrangian | 1024 | 22.71 | 44 |
| | 4096 | 189.70 | 5 |
| | 9216 | 722.42 | 1 |
| | 12544 | 1309.44 | 1 |
| | 16384 | 2180.73 | 0 |
| Grid-Optimized Lagrangian | 1024 | 7.83 | 128 |
| | 4096 | 18.70 | 53 |
| | 9216 | 31.01 | 32 |
| | 12544 | 38.25 | 26 |
| | 16384 | 41.27 | 24 |

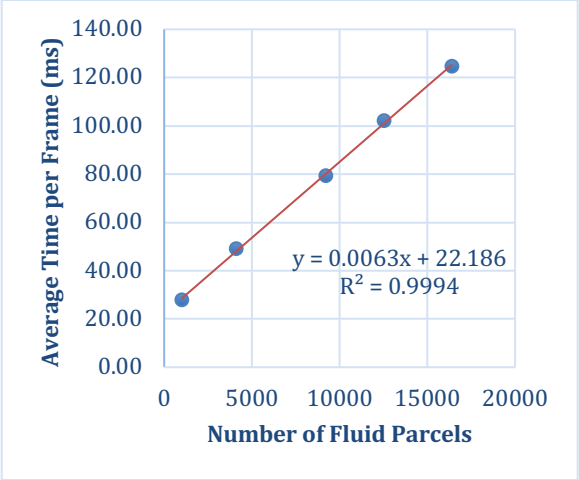
Table 1: Each algorithm's average time per frame (for ten frames)

## 4.2 Algorithm Performance Graphs

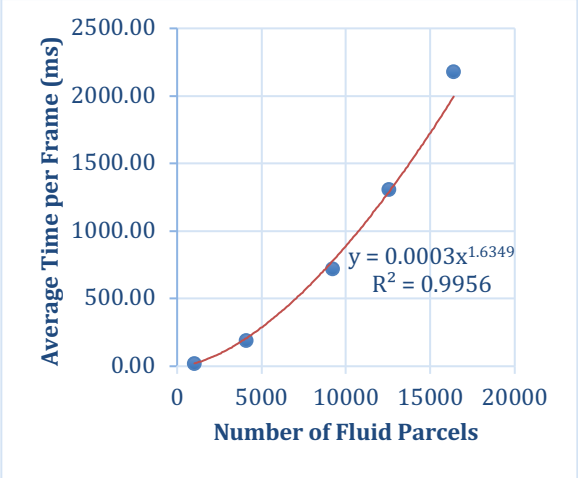
The graphs below illustrate the data of the four algorithms. Each graph also displays a trend line, its equation, and its correlation $R^2$ value.



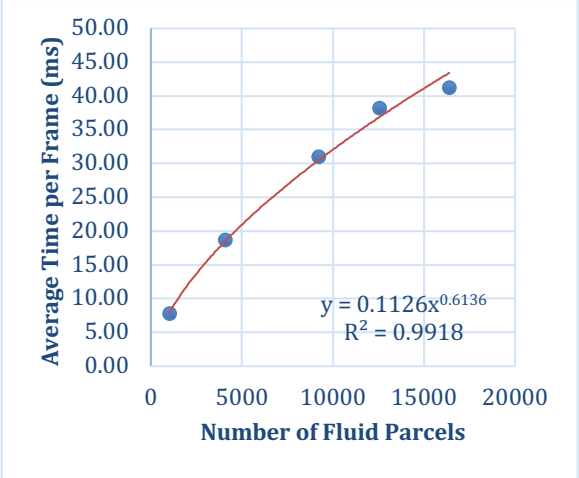

Graph 1: Eulerian algorithm data




Graph 2: Quadtree-Optimized Eulerian algorithm data




Graph 3: Lagrangian algorithm data




Graph 4: Grid-Optimized Lagrangian algorithm data

## 4.3 General Analysis

Evidently, grid-based algorithms and optimizations run faster. The grid-based Eulerian algorithm has a linear relationship graph, affirming its theoretical linear time complexity (Graph 1). For all numbers of parcels, it had the lowest time per frame compared to all the other algorithms. This makes sense as the for-loop iterations to update the MAC grid is proportional to the number of parcels, and accessing any grid fluid parcel is done in constant time.

The overall computational time of the grid-optimized Lagrangian algorithm was second best. Whereas Eulerian MAC cells all have 4 neighbors, Lagrangian particles may have more than four neighbors in adjacent cells, especially in the initial frames when same-cell particle clustering occurs from random particle spawning. As a result, increasing neighbor-finding iteration count caused more time per frame. However, the time per frame conformed to an approximately square root relationship instead of the theoretical $O(n)$ for grid-based algorithms (Graph 4). $N$ particles only cover a fourth of the space, meaning for higher amounts of particles, there were more grid cells to spread out to. This made particle clustering less likely, meaning less ArrayList looping, and thus less time per frame. However, the algorithm should demonstrate a more linear relationship if there is one particle per cell due to constant time neighbor finding, suggesting a limited range of data. A future experiment should thus have more manipulations of fluid parcel count.

In contrast, the unoptimized Lagrangian algorithm's data supports its $O(n^2)$ theoretical time complexity (Graph 3). Its underlying list data structure meant neighbor finding had to be brute forced. This caused time per frame to be whole seconds more than other algorithms. Thus, using lists to store fluid parcels should be avoided at all costs.

Perhaps most surprisingly was how the quadtree-optimized Eulerian algorithm data conformed to an almost linear relationship (Graph 2). The algorithm's theoretical $O(n \log n)$ time complexity was only calculated off worst-case quadtree recursive searches. It does not reflect adaptive refinement during simulation. Without external influences, regions that were initially uniform were coarsened, resulting in more regional uniformity, creating an overall trend of coarsening. Coarsening means less nodes, which saves node searching time. This balances out with the $O(\log n)$ node search time complexity, causing linear time complexity, but time per

frame is still five times that of the Eulerian algorithm due to extra recursion. Gradual coarsening also suggests a concave down relationship matching $O(\log n)$ complexity for higher values of $N$, demonstrating a methodological fault in having the maximum of independent variable $N$ be $16384$, which is logarithmically unnoticeable.

## 4.4 Sample Raw Data and Analysis

Examining the experimental raw data provides further insight into the performances of the algorithms, as well as the experimental methodology's effectiveness. The raw data below displays the times per frame over 10 frames of the MAC-unoptimized and quadtree-optimized Eulerian algorithm simulating 12544 fluid parcels:

| | Time per Frame (ms) | | |
|---|---|---|---|
| Frame Number | Trial 1 | Trial 2 | Trial 3 |
| 1 | 53.83 | 51.69 | 54.60 |
| 2 | 25.43 | 23.81 | 23.42 |
| 3 | 12.36 | 14.51 | 11.67 |
| 4 | 9.84 | 9.65 | 9.01 |
| 5 | 11.40 | 10.50 | 12.25 |
| 6 | 9.62 | 9.58 | 9.69 |
| 7 | 9.37 | 9.92 | 9.75 |
| 8 | 9.02 | 9.53 | 9.71 |
| 9 | 7.69 | 7.10 | 7.19 |
| 10 | 10.23 | 7.21 | 7.18 |

Table 2: MAC Eulerian algorithm raw data

| | Time per Frame (ms) | | |
|---|---|---|---|
| Frame Number | Trial 1 | Trial 2 | Trial 3 |
| 1 | 334.10 | 333.82 | 321.78 |
| 2 | 91.98 | 82.38 | 86.47 |
| 3 | 95.86 | 93.04 | 66.81 |
| 4 | 86.41 | 82.93 | 69.33 |
| 5 | 66.61 | 62.51 | 68.61 |
| 6 | 93.38 | 91.60 | 54.32 |
| 7 | 101.77 | 90.33 | 91.77 |
| 8 | 65.68 | 64.02 | 82.04 |
| 9 | 78.86 | 73.05 | 53.99 |
| 10 | 59.08 | 57.69 | 63.31 |

Table 3: Quadtree-optimized Eulerian algorithm raw data

There is extra time taken for the first frame (Table 2). In fact, this applies to all other algorithms. This was surprising as the Eulerian algorithm maintained a constant number of iterations per update. The data is explained by Java's Just-In-Time compiler (JIT), which uses and saves computational resources by gradually analyzing and optimizing running Java code. Initially, the JIT optimizes method calls, data flow, control flow, and memory, increasing the

update time (IBM Corporation, n.d.). Then, as the optimized code is run, the time per frame decreases and plateaus.

The quadtree-optimized Eulerian algorithm (Table 3) displays more exaggerated decreases in time per frame, especially in frames 1-3. This suggests that the algorithm's self-adapting resolution saved significant amounts of computational resources. The time per frame fluctuations after the first 4 frames demonstrate the effects of arbitrary refinement and coarsening.

The unoptimized and grid-optimized Lagrangian algorithms' raw data for simulating 12544 fluid parcels also demonstrate nuanced computational behavior:

| | Time per Frame (ms) | | |
|---|---|---|---|
| Frame Number | Trial 1 | Trial 2 | Trial 3 |
| 1 | 1899.15 | 1895.69 | 1984.00 |
| 2 | 1552.41 | 1572.28 | 1634.90 |
| 3 | 834.88 | 842.52 | 842.27 |
| 4 | 1304.29 | 1319.29 | 1314.82 |
| 5 | 1309.56 | 1316.14 | 1310.27 |
| 6 | 1314.39 | 1323.05 | 1332.34 |
| 7 | 1320.49 | 1312.29 | 1322.67 |
| 8 | 1316.93 | 1326.68 | 1328.92 |
| 9 | 1315.15 | 1329.59 | 1327.14 |
| 10 | 818.69 | 830.80 | 831.50 |

Table 4: Unoptimized Lagrangian algorithm raw data

| | Time per Frame (ms) | | |
|---|---|---|---|
| Frame Number | Trial 1 | Trial 2 | Trial 3 |
| 1 | 130.77 | 119.74 | 128.40 |
| 2 | 63.91 | 48.07 | 65.23 |
| 3 | 38.12 | 41.23 | 34.33 |
| 4 | 32.57 | 30.71 | 31.11 |
| 5 | 24.48 | 31.99 | 30.00 |
| 6 | 20.81 | 19.09 | 22.43 |
| 7 | 18.40 | 15.88 | 28.52 |
| 8 | 21.12 | 16.00 | 20.43 |
| 9 | 22.42 | 18.09 | 16.87 |
| 10 | 17.90 | 20.21 | 18.60 |

Table 5: Grid-optimized Lagrangian algorithm raw data

These algorithms both demonstrate the first frame taking the longest, and time per frame plateauing after 4 frames (Tables 4 and 5). The Table 4 frame 10 data seems unusual, but a second experiment involving 20 frames revealed that for frames 10-20, the time was between 700 ms to 900 ms, meaning the time per frame just happened to reach its final plateau at frame 10. However, more inexplicable outliers appear at frame 3 in Table 4, similar to frame 4 in Table 2, frame 5 in Table 3, and frames 6-7 in Table 5. Thus, although the data mostly conforms to the

decreasing and eventual plateauing trend caused by JIT optimizations, systematic outliers demonstrate uncontrolled compiler effects, and are topics of further investigation.

## 5. Further Research

The data in this experiment reveals the behaviors of algorithms that are list-based, grid-based, and tree-based, but other CFD algorithms (especially hybrids) can also be tested and compared for their computational times. Fluid starting positions can be predetermined to analyze certain aspects of algorithms. The frame count and parcel count can be increased to better analyze more general effects of underlying data structures on computational time.

Memory usage can also be considered to holistically analyze computational complexity. It can shed new light on the extent to which the quadtree-optimized Eulerian algorithm actually optimizes the MAC Eulerian algorithm. Quadtree simulation is best suited for large spaces, where subtrees are only refined around fluid concentrated in specific locations. Testing that might yield much more favorable computational efficiency data.

There can be further exploration on CFD algorithms by testing the addition or removal of fluid matter over time. Physical constants such as viscosity and diffusion can be changed to simulate different kinds of fluids. Fluids can be subject to external forces from gravity and collisions with solid obstacles, or even other types of fluids. Fluids can even be computed in higher dimensions, like 3D space.

Moreover, both grid-based and particle-based algorithms can leverage the powerful parallel computation of GPUs to run even faster than they do on the CPU. This would better suit modern standards and be more directly applicable to modern computers.

## 6. Conclusion

The experimental results demonstrate that different underlying data structures cause highly different computational times. From fastest to slowest are grid-based algorithms, tree-based algorithms, then particle-based algorithms. Computational time was largely based off the time complexity of neighbor data finding and accessing. The decreasing time per frame throughout each trial also demonstrate the heavy influence of JIT optimizations.

There were many experimental limitations, and most were due to the copious uncontrolled variables that made comparing computer algorithms difficult. If I were to repeat the experiment, I would set viscosity, diffusion, gravity, parcel size, and bounding space to be the same for all algorithms. I would disable real-time compiler optimizations (e.g., the JIT), to eliminate the influence of a hidden compiler layer on computational time. I would also additionally measure memory usage to better analyze computational efficiency.

The applications of this experiment are numerous. Knowledge of the different computational times of various algorithms allows one to make better judgements on using the appropriate algorithm in certain situations. If I were to make a game that runs in real-time on modern laptops, I would use the Eulerian algorithm for highly detailed smoke emissions limited to a small volume, or the grid-optimized Lagrangian algorithm for flowing water throughout an environment. Analysis of individual algorithms and why some perform better is crucial to computational physicists and computer graphics researchers to designing new CFD algorithms. If neighbor-finding is further optimized, then fluid can more efficiently be simulated to suit specific constraints, or to apply to more general situations with dynamic fluid control. As a whole, the essay demonstrates the overall trends in computer science, especially the efficiencies and applications of various data structures to model discrete information, such as fluids.

## 7. References

Adams, B., Pauly, M., Keiser, R., & Guibas, L. J. (2007, July 29). Adaptively Sampled Particle Fluids. *ACM Transactions on Graphics, 26*(3), 48-54. doi:10.1145/1276377.1276437

Ash, M. (2006, March 13). *Fluid Simulation for Dummies.* From Mike Ash: https://mikeash.com/pyblog/fluid-simulation-for-dummies.html

Desbrun, M., & Gascuel, M.-P. (1996). Smoothed Particles: A new paradigm for animating highly deformable bodies. *Computer Animation and Simulation '96 (Proceedings of EG Workshop on Animation and Simulation)* (pp. 61-76). Springer-Verlag.

Foster, N., & Metaxas, D. (1996, September). Realistic Animation of Liquids. *Graphical Models and Image Processing, 58*(5), 471-483. doi:10.1006/gmip.1996.0039

Gingold, R. A., & Monaghan, J. J. (1977, December 1). Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society, 181*(3), 375-389. doi:10.1093/mnras/181.3.375

Harlow, F. H., & Welch, J. E. (1965). Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface. *Physics of Fluids, 8*, 2182-2189. doi:10.1063/1.1761178

IBM Corporation. (n.d.). *How the JIT compiler optimizes code*. From IBM Documentation: https://www.ibm.com/docs/en/ztpf/1.1.0.15?topic=compiler-how-jit-optimizes-code

Losasso, F., Gibou, F., & Fedkiw, R. (2004, August 1). Simulating Water and Smoke with an Octree Data Structure. *ACM Transactions on Graphics, 23*(3), 457-462. doi:10.1145/1015706.1015745

Metacritic. (2018). *Shadow of the Tomb Raider*. From Metacritic: https://www.metacritic.com/game/playstation-4/shadow-of-the-tomb-raider/critic-reviews

Müller, M., Charypar, D., & Gross, M. (2003). Particle-Based Fluid Simulation for Interactive Applications. In D. Breen, & M. Lin (Ed.), *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, (pp. 154-159). From https://matthias-research.github.io/pages/publications/sca03.pdf

Samet, H. (1989). Neighbor Finding in Images Represented by Octrees. *Computer Vision, Graphics, and Image Processing, 46*, 367-386. From http://www.cs.umd.edu/~hjs/pubs/SameCVGIP89.pdf

Schuermann, L. V. (2016, May 23). *Particle-Based Fluid Simulation with SPH.* From Lucas V. Schuermann: https://lucasschuermann.com/writing/particle-based-fluid-simulation

Schuermann, L. V. (2017, July 8). *Implementing SPH in 2D.* From Lucas V. Schuermann: https://lucasschuermann.com/writing/implementing-sph-in-2d

Shi, L., & Yu, Y. (2004). Visual smoke simulation with adaptive octree refinement. *The 7th IASTED International Conference on Computer Graphics and Imaging (CGIM 2004).* Kauai.

Stam, J. (1999, Jul). Stable fluids. *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, (pp. 121-128). doi:10.1145/311535.311548

Stam, J. (2003). Real-Time Fluid Dynamics for Games. *Game Developers Conference.* San Jose. From https://www.dgp.toronto.edu/public_user/stam/reality/Research/pdf/GDC03.pdf

The Editors of Encyclopaedia Britannica. (2021, May 11). *Fluid.* From Encyclopaedia Britannica: https://www.britannica.com/science/fluid-physics

Thomas, A. C. (2020, Jan 22). *What is computational complexity?* From Medium: https://medium.com/the-ultimate-engineer/what-is-computational-complexity-66722cd5f8dd