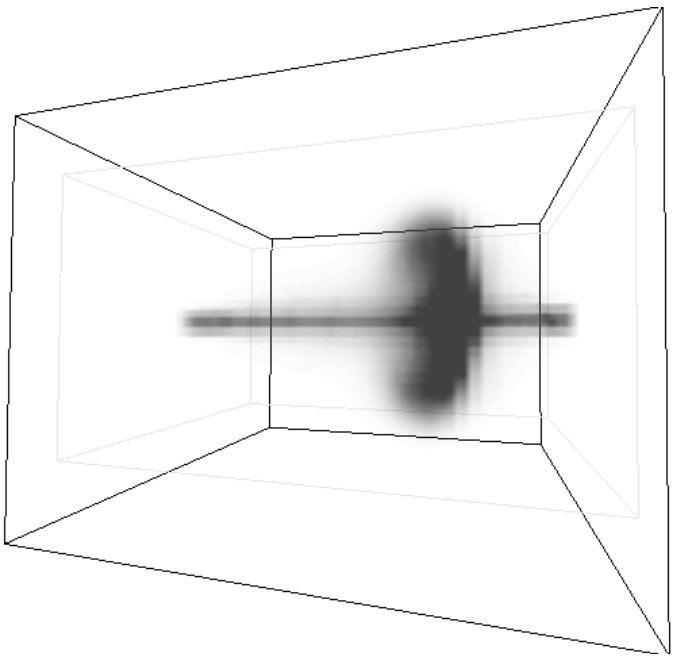
Simulation and Visualization of a 3D Fluid

Michael Ash

September 2005



Acknowledgements

I would like to thank my thesis advisor, Sophie Robert, and the entire Virtual Reality team at the LIFO. I would also like to thank Jonathan Albers at North Dakota State University for giving me access to their cluster for performance tests, Ed Wynne for his advice on optimizing the raytracing algorithm, and Eugene Ciurana for his help with the network performance analysis.

Contents

1 Introduction

Scientific visualization is becoming more and more important in physics, engineering, aviation, and many other fields. Investigating the origins of the universe, building a nuclear power plant, or designing an airplane wing can all benefit from computational scientific visualization. Visualizing an interactive 3D simulation allows a scientist to more clearly understand the subject and the results of the simulation. Humans are easily able to understand a shape, an interaction, or a movement when it's in the form of an image. Computational visualization allows us to exploit this ability to better communicate information from the computer to the human.

The state of the art of scientific simulation and scientific instruments is always progressing, which constantly increases the amount of data produced. Visualizing this data requires new data-processing techniques.

Clusters of PCs are able to process a large amount of data while still using standard computer hardware. Avoiding specialized hardware gives a better price/performance ratio for suitable applications.

1.1 Subject of the Research

This work deals with a 3D fluid simulation. Fluid simulation is a typical and useful application in scientific simulation which can produce data which is difficult to visualize with traditional 3D visualization techniques. The goal of this work was to research the available techniques for this type of visualization, increase the size of the data the computer is able to process, and visualize the data using a cluster of low-cost PCs.

Fluid simulations in 3D produce volumetric data, which is data organized in a three-dimensional grid. This kind of organization is very different from traditional 3D rendering, which only deals with two-dimensional surfaces in three-dimensional space. The method used to display this data is called volumetric rendering.

The work was carried out during a five-month period in the summer of 2005 at the Laboratoire d'Informatique Fondamentale d'Orléans at the Université d'Orléans. The first part consisted of researching the state of the art of volumetric rendering and fluid simulation. The second part consisted of creating an interactive fluid simulation and adapting them to large-scale simulations. This required optimizing the algorithms in question and adapting the application for parallel computation on clusters in order to increase the size of the data the application could process.

The first part of this document describes and analyzes the state of the art of fluid simulation and volumetric rendering. Next, we detail the various approaches for volumetric rendering and the advantages and disadvantages of each, keeping in mind the ultimate goal of parallelizing them to run on a cluster. Next we discuss optimizing and parallelizing the fluid simulation and rendering implementations. The final section analyzes the work with respect to the theoretical

and actual performance of the application using data obtained during tests on two clusters.

2 State of the Art of Volumetric Rendering

2.1 Applications

Volumetric rendering is a technique that is very costly in terms of processor time, and which also requires a great deal of memory to store the data used.

One of the first important applications for volumetric rendering was visualization of medical data. Medical scanners such as CT scanners or MRI scanners produce a large quantity of three-dimensional data. A CT scan generates many images of slices inside the subject's body. It is of course possible to look at these images directly, but these slices can be very difficult to understand on their own. Assembling the slices into a single three-dimensional image gives a better overview of the data. At the beginning, computing technology was limited to providing only two-dimensional images of the resulting data, but since the 1990s it has been possible to visualize the data from these scans in three dimensions using volumetric rendering techniques, which gives the user a better understanding of the scan [4].

2.2 Current Techniques

Currently there are two principal techniques that are used for volumetric rendering, with some variants. One technique is a raytracing algorithm which simulates the physics of light within the volumetric data, and the other technique consists of searching for surfaces within the data, which gives two-dimensional objects that the computer can display using traditional 3D rendering techniques.

2.2.1 Raytracing

Raytracing consists of creating a virtual screen, then tracing the path of light rays while they pass through the volumetric data in order to determine the color of each pixel. This technique is used to draw fog in games, and for medical data [3] [6].

Voxels One of the simplest techniques is to draw each point of data in the 3D cube directly on the screen by using one or several polygons or points in 3D space. This technique is very easy to implement but tends to be slow. This variant is actually a mathematical simplification of the raytracing technique, although its implementation is completely different and much less complicated than a true raytracing algorithm.

A team at Mitsubishi developed an architecture for a graphics card which can render relatively large quantities of volumetric data with low-cost hardware. This architecture uses a data storage layout which allows a large number of graphics processors to simultaneously render different pieces of the same data.

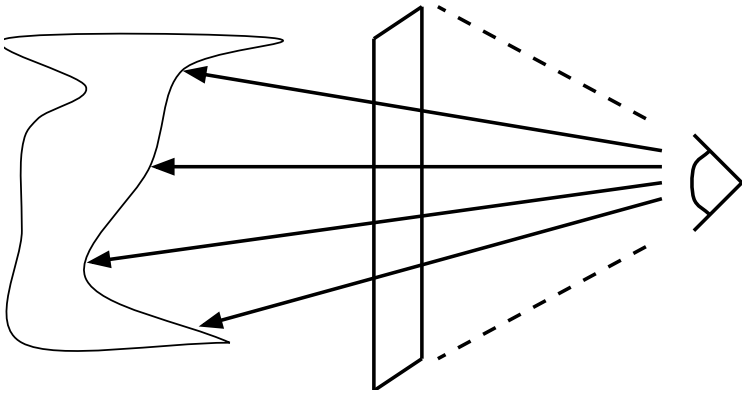


Figure 1: Volumetric rendering using raytracing. On the right, the camera; in the center, the virtual screen; on the left, the object to display.

This allows for a high degree of parallelism and therefore high-speed rendering, up to 30 frames per second for a 256^3 cube. This architecture is based on the voxel rendering technique [15].

Slices in 3D Textures It is possible to use a standard graphics card which has support for 3D textures to accelerate volumetric rendering. The architecture created by the Mitsubishi team allows for very high-speed rendering, but the hardware is specialized for this single application and is therefore costly. Standard graphics cards are produced on a large scale which makes them less expensive and faster. Such a card is also general-purpose, meaning the same hardware can be used in other applications.

The volumetric data is loaded onto the video card as a 3D texture. The cube to display is cut into slices which are positioned to face the camera. Each slice shows a piece of the 3D cube. When the slices are drawn in a back-to-front order, the user sees the cube displayed on his screen. Like the voxels technique, this technique is a mathematical simplification of raytracing, but with an extremely different implementation. This technique uses a dedicated graphics card which is present in most modern computers, which allows the main processor to be used for other tasks while the graphics card performs the rendering [14].

2.2.2 Surface reconstruction techniques

The other principal technique is surface reconstruction. We define regions inside the volumetric cube, and the computer searches for surfaces between the regions. For example, if the cube contains a sphere with cell values of 1 on the interior and 0 on the exterior, the two regions could be defined as $value < 1$ and $value \geq 1$. The surface reconstruction algorithm would then construct the surface of the sphere using the values present in the cube.

A fluid simulation could define a region where the density is greater than

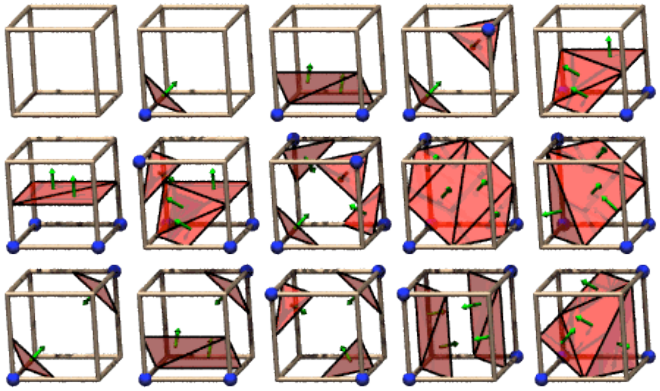


Figure 2: The fifteen combinations possible during one step of Marching Cubes [13]

a certain value, and this technique would then search for isosurfaces where the density is equal to this value. Medical data would define regions such as the brain, the skull, arteries, etc., and search for the surfaces between them. After defining the regions, the surface reconstruction algorithm analyzes the data and creates the surfaces.

The standard surface reconstruction algorithm is Maching Cubes. This algorithm examines the data in groups of eight, with each group forming a small cube. For each corner of the cube, the algorithm decides whether it is on the inside or the outside by using the provided region definitions. Taking into account all possible rotations and reflections, there are only fifteen possible combination (see Figure 2), each one with a unique set of polygons which the algorithm adds to the surface under construction.

Surface reconstruction techniques are fairly slow, and in general too slow to be used in real time. For static data, the algorithm only has to be applied one time and the algorithm's speed is not a large problem. Once the algorithm has been applied, the computer works directly with the resulting 2D surfaces [12].

3 Fluid Simulation

Fluids are seen everywhere in physics. Airplanes, boats, power plants, and many other things depend on fluid mechanics. The math behind fluid mechanics do not have exact solutions except in very simple cases, which makes numerical fluid simulations which generate approximate solutions very useful.

Jos Stam [7] proposes a two-dimensional fluid simulation based on real fluid physics, but whose speed is adequate for simulation on a standard PC. We developed a fluid simulation based on Stam's, generalized in three dimensions.

3.1 Fluid Physics

A fluid is modeled as a vector field which represents the velocity of the fluid, and a scalar field which represents the density. The movement of the fluid is determined by the Navier-Stokes equations.

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + v\nabla^2\mathbf{u} + \mathbf{f}$$

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa\nabla^2\rho + S$$

The first equation determines the movement of the velocity field, and the second determines the movement of the density field.

3.2 Simulation

For a computer simulation, it is necessary to create a discrete representation of the fluid. Our simulation places the fluid in a grid where each cell contains a velocity and a density. The movement of the fluid is determined by several simulation steps carried out on the velocity and density.

3.2.1 Velocity Diffusion

The second term in the Navier-Stokes equations determines diffusion in the fluid. The first step of the simulation is the diffusion of velocity values. The diffusion step calculates how the simulated fluid moves in order to satisfy this term. An iterative Gauss-Seidel solver in the `lin_solve` function finds the solution. This function is applied to each velocity component.

The solver generates solutions which do not respect conservation of mass within the fluid. A second function, `project`, which also uses a Gauss-Seidel solver, is applied to transform the velocity field into one which respects conservation of mass.

3.2.2 Velocity Advection

The velocity field determines the movement of the fluid. The advection step applies the velocity field to the fluid in order to calculate this movement. It's necessary to apply the velocity field to the density field and also to the velocity field itself. The function `advect` examines the velocity in each cell and determines updates the cell's contents according to the velocity it finds and the values of the surrounding cells.

3.2.3 Density Diffusion and Advection

After determining the movement of the velocity field, the simulation then determines the movement of the density field. The same functions, `diffuse` and `advect`, are used on the density field to determine its new state.

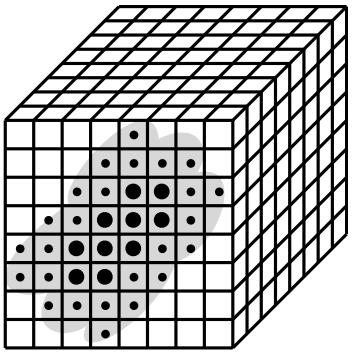


Figure 3: Organization of a fluid simulation's volumetric data

3.3 Extension in 3D

Stam's simulation is presented as a two-dimensional simulation, but all of his techniques can be applied to a three-dimensional simulation. It is necessary to add a third term in everything which manipulates the grids, and change the loops and data storage to correspond to take the third dimension into account within the simulation.

This change makes the results of the simulation difficult to display. A two-dimensional simulation produces data which can be easily transformed into pixels that are then displayed directly on the screen. However, a three-dimensional simulation creates volumetric data which are difficult to display. Displaying these results requires volumetric rendering techniques.

4 Volumetric Rendering

In order to represent a fluid in general, space (2D or 3D) is divided into as many cells as necessary to ensure sufficient precision in the simulation. Each cell contains values which represent the fluid (density and velocity). The more cells used in the simulation, the more accurate the fluid's representation becomes.

A three-dimensional fluid simulation produces volumetric data. This data is organized in the same manner as a two-dimensional image, but generalized in three dimensions, essentially a three-dimensional grid. When this grid is discussed in a rendering context, the individual cells are often referred to as voxels, a generalization of the word *pixel*. For a fluid simulation, the grid to display on the screen is a grid where each cell contains the density of the fluid at that location in space (see Figure 3). This representation can also be applied to medical data, for example, where each cell would contain a density, the type of tissue, the amount of blood, etc.

Even though volumetric data is very similar to standard two-dimensional data, it is much more difficult to display due to the fact that they are generally

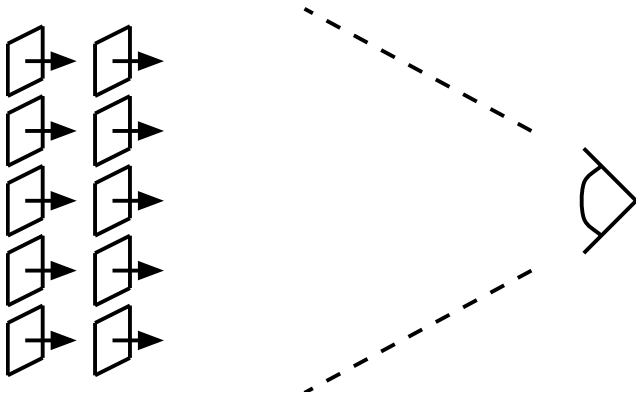


Figure 4: Volumetric rendering with voxels drawn directly on the screen

much larger, and much less adapted to computer display hardware. Most 3D rendering techniques concentrate on rendering 2D polygons in 3D space. Volumetric data is not generally transformable into 2D polygons unless every voxel is translated into polygons, which generates an enormous number of polygons. We are faced with a choice of using the graphics card to display data to which it is not very well adapted, or of not taking advantage of the graphics card and instead using the CPU to execute a more appropriate algorithm.

4.1 Techniques

We explored current volumetric rendering techniques, looking for their ability to display dynamic data (such as a constantly-changing fluid simulation) and the possibility for parallelization.

4.1.1 Voxels

The simplest technique is to draw the voxels directly on the screen with a simple 2D square. For each voxel in the cube, the computer draws a square in the same position, facing towards the comera, and with the appropriate color.

Often, the data to be displayed is very sparse, where most voxels are empty. For this type of data, the algorithm can be optimized not to draw the squares which correspond to a voxel whose value is less than a certain threshold.

This technique is very easy to implement and reasonably fast for small quantities of data, or very sparse data sets, but it can become extremely slow. On a typical graphics card, drawing a polygon is relatively slow compared to drawing a single textured pixel. A modern graphics card at the time of this writing can draw perhaps 40 million triangles per second. For a volumetric cube whose dimensions are 256^3, there are almost 34 million triangles to draw, which makes it impossible to obtain a fluid real-time display for a cube of this size given the state of the art of current hardware.

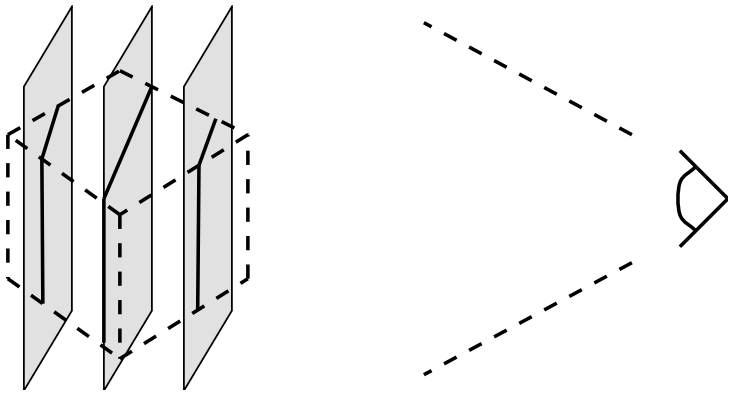


Figure 5: Volumetric rendering using 3D texture slices

Parallelizing this technique, in order to use a multiprocessor machine or a cluster, is difficult. Every node would need a powerful graphics card, even the nodes which aren't involved in the final display of data. The nodes would need to read the final rendering out of the graphics card in order to transmit the results to the nodes which are tasked with displaying them to the user, and this operation is usually fairly slow. Given these limitations, this technique is not appropriate for large-scale use.

4.1.2 Slices

OpenGL and a large number of modern graphics cards support 3D textures. A 3D texture is used to texture 2D polygons exactly like a standard 2D texture, and it is not possible to display the entire 3D texture in a single operation.

However, it is possible to use 3D textures to display volumetric data with some extra work. A series of slices are calculated within the cube, facing the camera, which are textured with the 3D texture. When the computer draws these slices, the result is a representation of the volumetric cube on the screen. By varying the number of slices, it is possible to easily change the level of detail in order to have more detailed images, or a more fluid display.

This technique takes advantage of the graphics hardware and therefore can be much faster than the direct voxel technique. However, it is also limited by the abilities of the graphics hardware. This technique requires the volumetric data to be loaded into the graphics card's memory, which is generally significantly smaller than the computer's main memory.

For dynamic data, this fact means that the computer is forced to reload the data onto the graphics card after each change in the data. Since the bandwidth to the graphics card is relatively small compared to the bandwidth to main memory, this can create a bottleneck for data updates. Also, simulations often produce data in a format which is not directly supported by the graphics card, requiring an expensive conversion for every update.

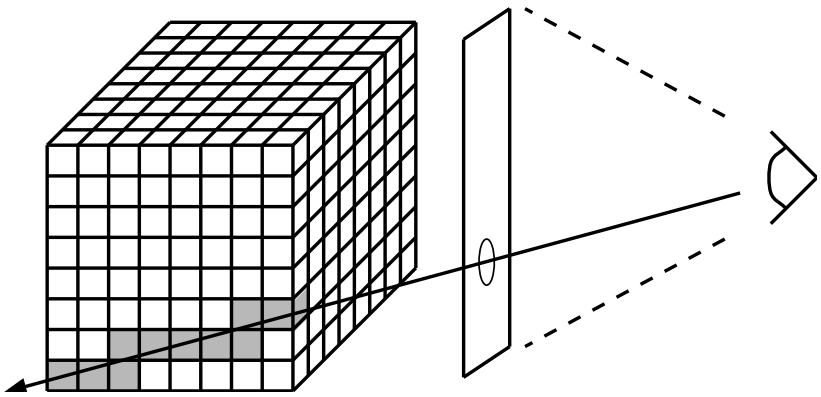


Figure 6: The path of a ray cast into volumetric data. The gray voxels are the samples.

Parallelizing this technique has the same problems as for the voxels technique. Every render node would require a high-end graphics card, and the data would need to be read back from the graphics card after rendering, which is normally a slow operation.

4.1.3 Raytracing

The physics of light suggests the simulation of light rays to produce the final image. In the real world, light passes within an object and then enters the eye or camera. In our virtual world, we use the same principle, but only examine the light rays that actually touch the camera, by tracing the rays in the opposite direction, from the camera to the object. A virtual screen is placed in front of the volumetric data, and rays of light are cast from the camera towards the virtual screen, and from there into the data. As the ray traverses the data, samples are taken as illustrated in figure 6 which allow the calculation of the final color at that location on the virtual screen.

For example, in our case, the values in the volumetric data represent the density of the fluid, and we can simply sum the values to determine the transparency of the cube in the following manner:

$$pixel = transparency \times \sum_{ray} voxel$$

The number of samples to take is a parameter in this algorithm. In the real world, the color of a ray in a transparent material is proportional to the thickness of the material and its transparency. This means that the color becomes stronger as the ray becomes longer. In order to mimic this, we take a number of samples proportional to the length of the ray.

$$samples = k_1 \times dim \times length(ray)$$

It is also necessary to determine the number of rays to cast, in other words to determine the size of the virtual screen. The virtual screen is a square of size dim_{screen}. Our raytracer calculates this size in proportion to the size of the volumetric cube in voxels:

$$dim_{screen} = k_2 \times dim_{cube}$$

When the cube faces the camera directly with $k_2 = 1$, there is exactly one ray per voxel in the plane of the cube that is closest to the camera. The cube's shape and orientation fits the virtual screen exactly. Each ray cast from the camera to the virtual screen intersects the cube.

However, if the cube turns and an edge or corner faces the camera, many of the rays that are cast miss the cube entirely because the cube's shape and orientation no longer match the virtual screen (see Figure 7), which remains an axis-oriented square facing the camera. The number of rays cast remains the same, but fewer of them pass within the cube. This means that the speed of rendering increases, but the quality of the resulting image decreases.

The value of k_1 is not fixed. When k_1 is small, each ray takes fewer samples, making the rendering process faster but reducing the quality of the resulting image. When k_1 is large, each ray takes more samples, taking more time but increasing the resulting quality. By varying this parameter, it is possible to exchange quality for speed or vice versa, which we call the rendering's level of detail (LOD). This parameter changes the number of samples which are taken along the depth of the cube, so we call this parameter LOD_{depth}.

With the same idea, the value of k_2 is also a LOD parameter. When k_2 is small, the number of rays cast decreases, making the rendering faster. When k_2 is large, more rays are cast and the rendering becomes slower, but produces a higher-quality image. This parameter changes the number of rays cast along the plane of the virtual screen, so we call this parameter LOD_{planar}.

In order to keep the image's quality and rendering speed constant, it is necessary to vary the LOD_{planar} parameter depending on the orientation of the volumetric cube. When the cube is not directly facing the camera, the computer should cast more rays towards the virtual screen in order to have the same number which intersect the cube. It would be possible to mathematically analyze the relationship between the cube's orientation and the proportion of rays which intersect it, which would allow the creation of a function that would determine a proper LOD_{planar} for an arbitrary orientation that would keep the image quality and speed constant. Another possibility would be to give a target speed, and have a small routine which varies LOD_{planar} in order to make the actual speed match the target speed as closely as possible. Our implementation simply keeps a constant LOD_{planar}.

5 Optimization of the Fluid Simulation

The fluid simulation proposed by Stam [7] requires a great deal of computation, particularly in 3D. If N is the linear size of the cube, the amount of data

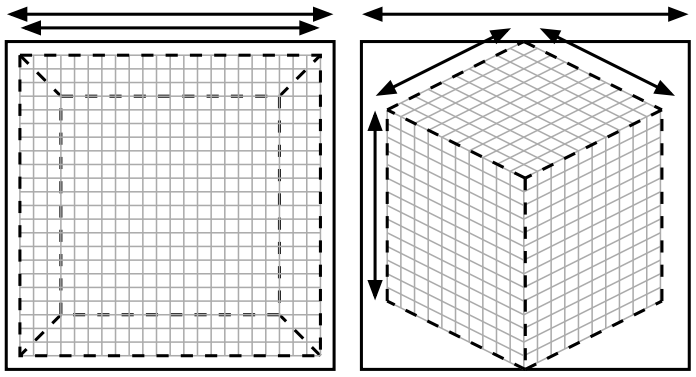


Figure 7: Left: the cube faces the camera. The number of rays equals the number of voxels in the plane of the virtual screen. Right: the cube turns. The number of rays which intersect the cube decreases greatly, and many voxels are skipped by the remaining rays which still intersect the cube.

processed by the simulation is proportional to N^3. The simulation requires dozens of passes within the fluid data in order to complete a single step of the simulation.

People are always looking to process more data in less time. The basic simulation performs a great deal of conversions between integer and floating-point types, and it accesses memory in a fairly unordered fashion. By optimizing these two weaknesses, the simulation is able to process much more data.

5.1 Int-float Conversions

On modern processors, conversion between integer and floating-point values is relatively slow. For example, converting from a `float` to an `int` on an Intel or AMD x86 processor forces a pipeline flush, which causes a large speed penalty [16]. On a Motorola PowerPC G4 800MHz, such a conversion takes over 43 nanoseconds, or 35 execution cycles [17]. In comparison, most floating-point instructions on this processor execute in one cycle in the best case, or five cycles in the worst case [**?**].

Our simulation code contains a lot of `int` variables whose values are invariant, or which change in a straightforward fashion, but which are used in calculations with floating-point values. This requires a conversion each time the calculation is performed, which is redundant but slow.

For example, the variable `N` in the `advect()` function is frequently used to compute floating-point values. By adding an `Nfloat` variable and performing the conversion before the loop, the conversion is only done once, saving a great deal of computation time.

In the same function, the variable `i` is similar, except its value changes frequently. However, the change is simple, and so it is possible to create a variable

`ifloat` which contains the same value. At the beginning of the loop, it is set to the same value as `i`, and then both of them are incremented simultaneously. This eliminates all conversions of the value of `i`.

Similarly, multiplications are preferred over divisions. The `lin_solve()` function performs one division by the variable `c` in each iteration of the loop. Since `c` never changes within the loop, it's possible to calculate its reciprocal, and replace the division by `c` with a multiplication by `cRecip`.

5.2 Memory Accesses

The order in which data is stored and accessed in memory is extremely important for performance optimization. Modern computer memory systems make various assumptions about memory access patterns, and a program which conforms to these assumptions will gain a great deal of performance compared to one which does not.

Most memory accesses in most programs are either to nearby addresses, or linear. In other words, either the program accesses memory in a relatively random fashion in a small region of memory, or it accesses addresses X, $X + 1$, $X + 2$, ... Because of these common patterns, the hardware attempts to optimize these two cases.

Modern memory systems have several levels of caches, pieces of memory which sit close to the CPU and provide rapid access to a small amount of data. When a piece of data is loaded, this data and its neighbors are loaded into the cache. When the processor then loads a neighboring piece of data, it finds it already present in the cache, which takes much less time.

Modern processors also have a prefetching module. This module snoops the addresses being requested by the CPU, and when it appears that the processor is starting to access memory in a sequential manner, it begins preloading the subsequent data. This way, while the CPU is performing computations on the value at address $X + n$, the prefetching module is simultaneously loading data from $X + n + 1$, which can greatly increase performance.

For architectures with cache hierarchies and a prefetching module, which is the standard architecture for modern high-performance computing, it is extremely important to organize a program's memory accesses to match the capabilities of the hardware.

5.2.1 Loop Ordering

In the case of a fluid simulation and many other algorithms, the program works with multidimensional data, but computer memory is one-dimensional. For two-dimensional data such as a matrix, the data is traditionally arranged by row (although arranging by row is sometimes used as well, for example in Fortran). With this arrangement, in an X by Y grid, where X is the number of columns and Y the number of rows, the index of the cell (x, y) is therefore $y \times X + x$. For three dimensions, a similar technique is used, with the planes being stored consecutively.

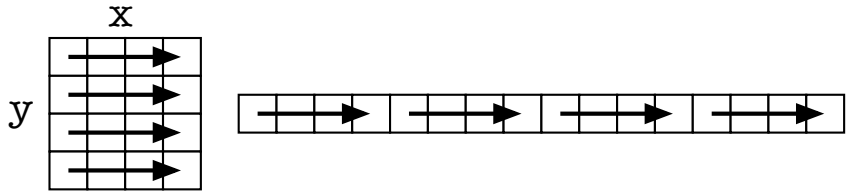


Figure 8: When `x` is incremented by the inner loop, the order in which values are loaded matches the order which is fastes for the hardware.

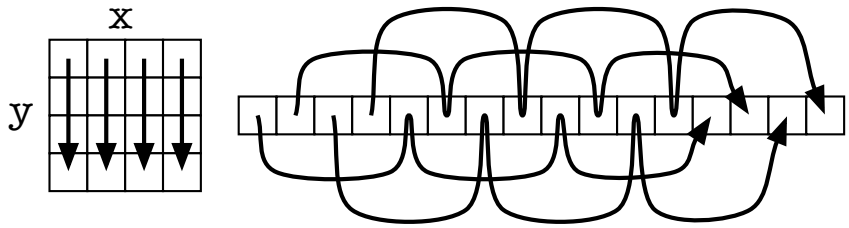


Figure 9: When `x` is in the outer loop, values are loaded in a fashion which fails to take advantage of the memory system.

Memory access ordering is determined by which variable is incremented the most often, which is in turn determined by the ordering of the loops which increment them. If x is incremented by the outer loop, and y is incremented by the inner loop, y changes more often, and x changes by 1 every time y completes a full loop. The loops work with the values at $(0,0)$, $(0,1)$, $(0,2)$, $(0,3)$, ..., $(1,0)$, $(1,1)$, $(1,2)$, $(1,3)$, ... These coordinates correspond to addresses 0, X, $2X$, $3X$, ..., 1, $X+1$, $2X+1$, $3X+1$, ... (See Figure 9). By switching the loops and putting x in the inner loop, the addresses are loaded in a linear fashion, as the cells load values from addresses 0, 1, 2, 3, ..., X, $X+1$, $X+2$, ... This sequence matches the hardware's capabilites much more closely (See Figure 8).

For three-dimensional data, the same techniques can be generalized with three coordinates. x should be incremented first, followed by y, followed by z in the outermost loop.

Therefore, it is extremely important to put loops in the proper order.

```
for(x = 0; x < N; x++) {
    for(y = 0; y < N; y++) {
        for(z = 0; z < N; z++) {
```

Instead, the order of the loops should be changed to put x on the inside:

```
for(z = 0; z < N; z++) {
    for(y = 0; y < N; y++) {
        for(x = 0; x < N; x++) {
```

This order will load data in a linear fashion, giving better performance.

5.3 Results

We tested these optimizations on an Apple PowerBook G4 with a 1.5GHz processor and a $64x64x64$ fluid cube. Before optimizing, the speed of the simulation was 0.22 steps per second. With all of the proposed optimizations, the simulation was able to perform 1.52 steps per second, giving an increase in speed of 591%.

5.4 Other Possibilities

Frequently, a program will access a single piece of data multiple times during a calculation. In this case, it's best to place the data accesses as close together to each other as possible. This increases the chance that the data will still be found in the cache when it is next requested. For example, loop `2` in the following code will run much faster on a large quantity of data compared to loop `1`:

```
// 1
for(n = 0; n < 3; n++)
    for(i = 0; i < size; i++)
        A[i] = f(B[i]);

// 2
for(i = 0; i < size; i++)
    for(n = 0; n < 3; n++)
        A[i] = f(B[i]);
```

This case is simple, because there are no dependencies between the calculations on each piece of data, and so it is trivial to rearrange the loops to make all of the accesses of a value be consecutive.

We can imagine performing a calculation in a table where the calculation of each value depends on the previous value in the table, and this calculation is performed three times. The simplest algorithm would be to simply repeat this loop three times:

```
for(n = 0; n < 3; n++)
    for(i = 1; i < size; i++)
        A[i] = f(A[i-1], A[i], A[i+1]);
```

These loops will traverse the table three times in a linear fashion. If the table were larger than the computer's cache, the computer will be forced to load the table from main memory three times.

The algorithm can be changed to perform the calculations in an order which optimizes memory access. Instead of a simple linear loop, the algorithm will only calculate far enough ahead to perform the complete sequence of three calculations for each value:

```
A[1] = f(A[0], A[1], A[2]); // 1

A[2] = f(A[1], A[2], A[3]); // 2
A[1] = f(A[0], A[1], A[2]);

for(i = 1; i < size - 2; i++) // 3
    for(n = 2; n >= 0; n--)
        A[i+n] = f(A[i+n-1], A[i+n], A[i+n+1]);

A[size-2] = f(A[size-3], A[size-2], A[size-1]); // 4
A[size-1] = f(A[size-2], A[size-1], A[size]);

A[size-1] = f(A[size-2], A[size-1], A[size]); // 5
```

This algorithm becomes more complicated because of the beginning and end of the table. To begin, it must perform the first calculation on the first entry in the table (1). Afterwards, it must calculate the intermediate values of the first two entries (2). Once the loop terminates, it has to do the same thing for the last two entries (4 and 5).

The loop itself (3) is the interesting portion of this example. The calculation is performed backwards in order to ensure that there are enough intermediate values present in the table to calculate the final value for each entry in the table. Memory is accessed in this order: $X+2$, $X+1$, X, $X+3$, $X+2$, $X+1$, $X+4$, ... The addresses are close together and the following accesses to each address are close to the first, making it likely that each piece of data will stay in the cache until all three calculations have been performed.

The `lin_solve` function in the fluid simulation is similar, but in three dimensions instead of one. For the Gauss-Seidel solver, the calculation for each cell in the fluid depends on the result of the calculation of three neigbors (See Figure 10).

For two or three dimensions, it is possible to use a similar technique as described for the one-dimensional table above. To begin, the program calculates the intermediate values in a triangle or pyramid. Next, it calculates the values for an entire line. Following this, it performs the calculation on a number of lines equal to the number of iterations performed by the solver (See Figure 11).

For a 3D solvel, the calculation has to be performed on several lines simultaneously. If i is the number of iterations, the calculation must be performed

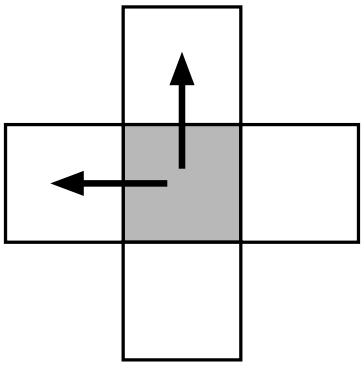


Figure 10: Data dependencies in a Gauss-Seidel solver. The calculation for each cell depends on the calculation of its neighbors in only one direction.

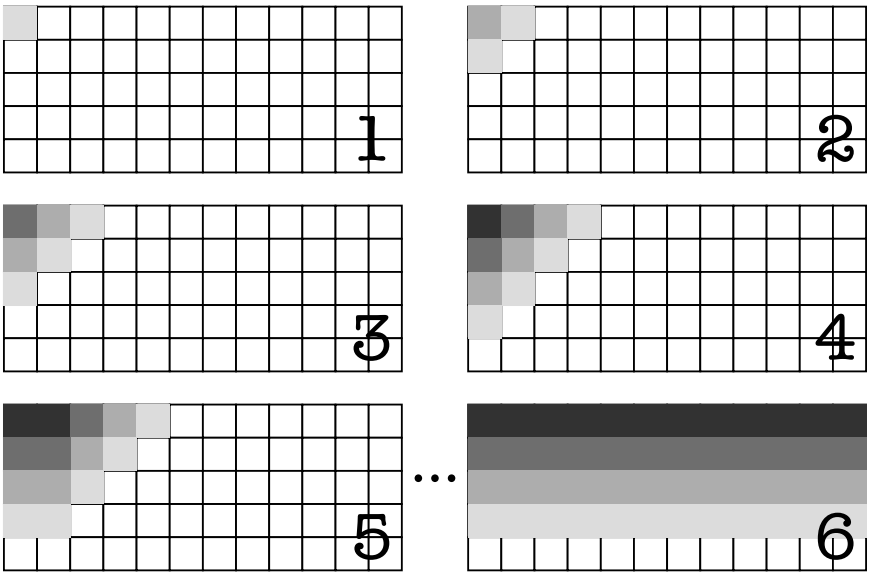


Figure 11: Order of calculation in a Gauss-Seidel solver written to match the capabilities of the memory system. 1) The first intermediate value is calculated in the corner. 2) The first intermediate value is calculated in its two neighbors, followed by calculating the second intermediate value in the corner. 3) The first intermediate value is calculated in the neighbors of the neighbors, which allows the calculation of the second intermediate value in the neighbors, and finally the third intermediate value in the corner. 4) Intermediate values have been calculated in a range of four cells, which allows the calculation of the fourth and final value in the corner. 5) Having calculated all four steps in the corner, the calculation proceeds on the first four lines. 6) The calculation terminates on the first line, and the program begins on lines 2 through 5.

on $1 + 2 + \cdots + i - 1 + i = i \times (i + 1)/2$ lines. For $i = 4$ and a $64x64x64$ cube containing 4-byte floating-point values, the amount of data being processed at any one time is:

$$\frac{4 \times 5}{2} \times 64 \times 4B = 2560B$$

This is smaller than the level-one cache of a typical processor, which is normally 32kB. Performing the calculation on the entire cube for each pass, as with the simple algorithm, 1MB of data must be loaded for each pass, which is much larger than the level-one cache, and likely larger than the level-two cache as well.

6 Optimization of the Raytracer

The basic raytracing algorithm is simple and slow. The simplest method is to work with floating-point values, but these must be converted to integers in order to calculate the address of the target voxel. The algorithm also requires a great deal of relatively random memory accesses, which limit the performance of the memory system.

6.1 Address Calculation

In section 5.1, we described how minimizing integer/floating-point conversions can aid performance. The main raytracing loop traces a single ray within the volumetric data. Using the size of the cube, the entry and exit points, and the LOD parameters, it calculates deltas dx, dx, and dz, as well as a number of samples to take. Starting from the entry point, each iteration of the loop takes a sample and increments the current position by these deltas. None of these deltas or coordinates can be integers, since they frequently have fractional values. This forces three `float-int` conversions for each iteration of the loop.

The index of the current voxel is calculated using the current coordinates:

$$index = x_i + y_i \times size_x + z_i \times size_x \times size_y$$

The multiplications in this expression are slow. It is also necessary to bound the coordinates to avoid problems relating to rounding errors, which requires costly comparison operations.

When the cube is of size $2^n \times 2^n \times 2^n$, it is simpler to convert a position within the cube into an index within the cube's data. Each dimension uses a whole number of bits in the address. For example, for a $32 \times 32 \times 32$ cube, each coordinate consists of exactly five bits in the index of the voxel:

$$\begin{aligned} index &= x + y \times 32 + z \times 32 \times 32 \\ &= x + y \times 2^5 + z \times 2^{10} \\ &= x|(y << 5)|(z << 10) \end{aligned}$$

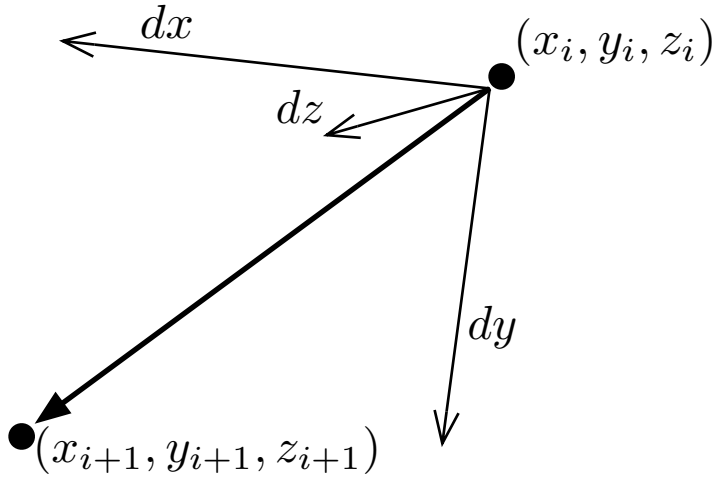


Figure 12: One step of the inner raytracing loop.

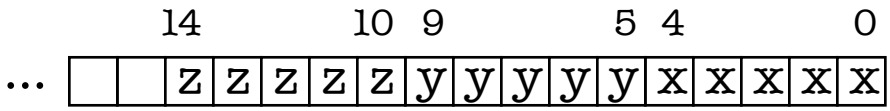


Figure 13: Bitwise organization of the coordinates in the index of a voxel in a 32x32x32 cube.

This computation only uses C bitwise operations, which are normally very fast. See Figure 13 for a graphical representation of the bitwise representation of a voxel's index in.

To avoid `float-int` conversions, fixed-point calculations are used. Our implementation uses a 20-bit mantissa and thus a fixed exponent of 2^{-20}. This representation can be rapidly converted into an integer [19].

These two operations can be combined. For each coordinate, there is a right shift by 20 to convert it to an integer, followed by a left shift to calculate the index. These two operations can be combined into one by using a mask:

```
// 1
y_int = y >> 20
y_int = y_int % ysize
index = index | (y << log2(xsize))

// 2
xshift = log2(xsize)
yimask = (y - 1) << xshift
yishift = 20 - xshift

// 3
index = index | ((y >> yishift) & yimask)
```

The calculation in 1 is replaced by the calculations in 2 and 3. Step 2 is performed only once when the program's data structures are initialized, and step 3 is performed for each iteration.

The number of operations is the same as before, but this version adds functionality by forcing values which become too large or small to be within the acceptable range. The full calculation of an index can be performed using two perations per coordinate, plus two more operations to combine the results:

```
index = ((sx >> xishift) & ximask) |
        ((sy >> yishift) & yimask) |
        ((sz >> zishift) & zimask);
```

6.2 Memory Access

In section 5.2, we saw how properly organizing memory access can result in significant performance gains. Unfortunately, the raytracing algorithm, tracing a ray within the cube accesses memory in a relatively random manner, and it is not possible to correct this.

```
while (x, y, z) in cube
    calculate index of (x, y, z)
    load value at index // 1
    total = total + value // 2
    if total > max
        stop
    calculate next (x, y, z)
```

Each step depends on the result of the previous step. The most time-consuming line is line 1 because of the memory access. The only line which

depends directly on its result is line 2. A loop which puts all of the other lines between them can run much faster. The CPU will frequently be able to execute the intervening lines while it waits for the memory access to complete.

```
calculate index of (x, y, z)
while (x, y, z) in cube
    load value at index // 1
    calculate next (x, y, z)
    if total > max
        stop
    calculate index of (x, y, z)
    total = total + value
```

This loop performs the same calculation but is much faster. The loop performs slightly more calculation than strictly necessary when it terminates, but this is more than compensated for by the better overall performance.

In tests on our sample data with all of the proposed optimizations, the speed of the raytracer compared to the initial implementation was increased by a factor of approximately three.

7 Parallelization

For this step of the work, we chose a modular approach with:

- A fluid simulation module which executes the 3D fluid solver and sends the results to the render module.
- A render module which takes the simulation data and executes the raytracing algorithm to generate pixel data.
- A display module which presents a user interface and displays the pixel data generated by the render module.

7.1 Parallel Fluid Simulation

In order to parallelize the fluid simulation, we cut the cube into pieces and run the simulation for each piece on its own calculation node.

To minimize communications between the nodes, each piece should have the smallest possible surface area. The optimal division is to cut the cube into subcubes. This division causes a problem with the number of simulation nodes. The number of subcubes is itself always cube, and so this division only works with certain numbers of nodes. On a small PC cluster, this means that the number of simulation nodes is limited to 1, 8, and possibly 27. It would be possible to distribute multiple one subcube per CPU, but this creates difficulties for keeping the workload evenly distributed, and requires more communications than would otherwise be necessary. The communications between the subcubes

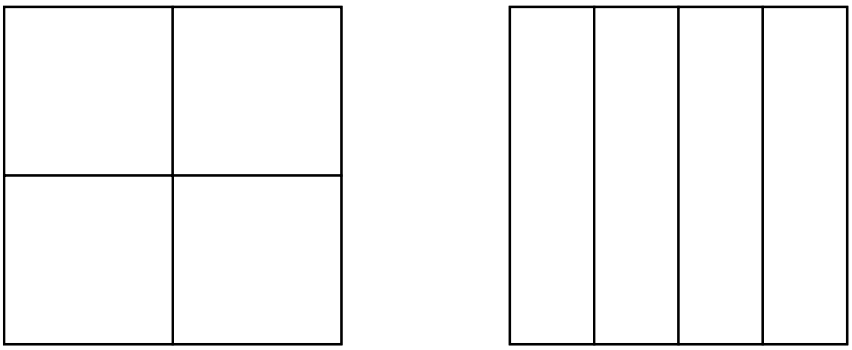


Figure 14: Left: the cube is divided into subcubes. Right: the cube is divided into slices.

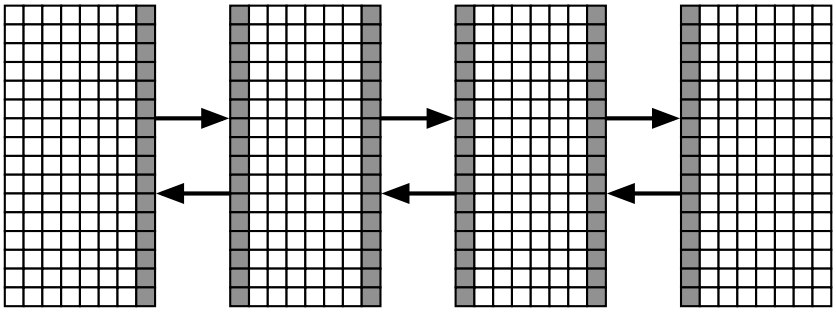


Figure 15: Fluid simulation on four CPUs. The cells in gray are the ghosts which are communicated after each substep.

is also difficult to manage, since any given subcube can have anywhere from three to six neighbors.

Another possibility is to cut the cube into slices and place each slice on a separate CPU. This division is not optimal in terms of the amount of communications required, but it works with any number of CPUs and the communications are much less difficult to manage, with a maximum of two neighbors per slice.

These two approaches are illustrated in Figure 14.

We decided to cut the cube into slices along the Z axis. We chose the Z axis because of how the data is organized: recombining the slices is a simple concatenation of each slice's data.

To handle communications between the slices, a layer of ghosts is created. The ghosts are cells in a neighboring slice which are copied into the current slice. These copies must be updated after each substep of the calculation which modifies them. The two nodes on the ends only have ghosts on one side, and the others have ghosts on two sides. (See Figure 15.)

There are actually four layers of ghosts, one for the density and one for each

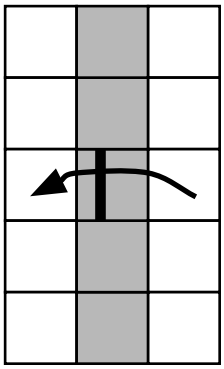


Figure 16: A cell containing a large velocity cannot traverse the border between two nodes.

of the three velocity components.

The ghosts are exchanged after each iteration of `lin_solve`, after each `advect`, and in the middle of and after each `project`. In total, the ghosts are exchanged 38 times per simulation step.

The size of the ghost layer can affect the accuracy of the simulation. In the `advect` step, a cell which contains a large velocity can modify a cell which is non-adjacent, but it is impossible to modify a cell on the other side of the ghost layer. If the simulation contains speeds which cause effects at a greater distance than the thickness of the ghost layer, the simulation will lose accuracy (see Figure 16). Normally, velocities present in the cells are much lower than what is required to affect non-neighboring cells, and so a ghost layer that is only one cell thick is sufficient.

If N is the size of the cube, the ghost layer is therefore N^2 cells. If each cell contains a four-byte value, the total quantity of data sent and received in each internal node is:

$$data = 2 \times 38N^2 \times 4B$$

The factor of two is due to the fact that each internal node has two layers of ghosts, one for each neighbor. For a $32x32x32$ cube, 304kB of data is transferred per step. For a $64x64x64$ cube, this increases to 1.18MB of data, and for a $128x128x128$ cube, 4.75MB. This quantity of communications can cause a bottleneck for the speed of the simulation.

7.2 Parallel Raytracing

In section 4.1, we saw that raytracing is the only volumetric rendering technique which is suitable for parallel processing.

We propose two complimentary parallelization schemes, one which divides the virtual screen and puts a complete copy of the volumetric cube on each CPU,

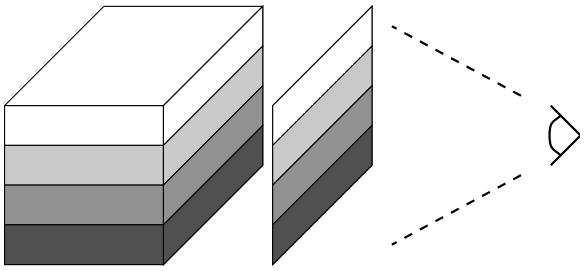


Figure 17: Parallel raytracing: the cube is divided into slices, and each slice is given to a different CPU. Each level of gray in the image is on a different processor.

and one which divides the volumetric cube into pieces and puts each piece on a separate CPU. It would also be possible to combine these two approaches, where the cube is divided into pieces, and each piece is distributed to several CPUs which divide the virtual screen for that piece amongst themselves.

7.2.1 Division of the Cube

If the cube is cut into pieces, there is a large advantage in terms of the amount of communications required. Instead of sending the entire cube to all render nodes, only one part of the cube is sent to each render node, which greatly diminishes the amount of data transmitted. However, the communications from the render nodes to the display node increases, because each render node has to transmit an entire virtual screen to the display node, which then computes a final composite image from the images of the render node. This division causes a problem with equal distribution of work among the nodes. It is possible that one region of the cube requires more computation time to render compared to another. With our raytracing algorithm, regions with more fluid take less time to render, as each ray which traverses a region of dense fluid will rapidly reach saturation, at which point the tracing of the ray can be aborted early. If the work is not evenly distributed, then performance can drop greatly as some nodes continue to work while others sit idle, which fails to take advantage of all available hardware.

This division of data is illustrated in Figure 17.

7.2.2 Division of the Virtual Screen

If we divide the virtual screen among the render nodes, the communications become much simpler. It's possible to divide the screen into slices or squares, but the same problem arises with an unequal distribution of work. However, it is possible to take advantage of the fact that the calculation of each ray is completely independent of the others, and divide the virtual screen in a cyclic manner, where the ray n is given to the CPU $n \bmod k$, where k is the total

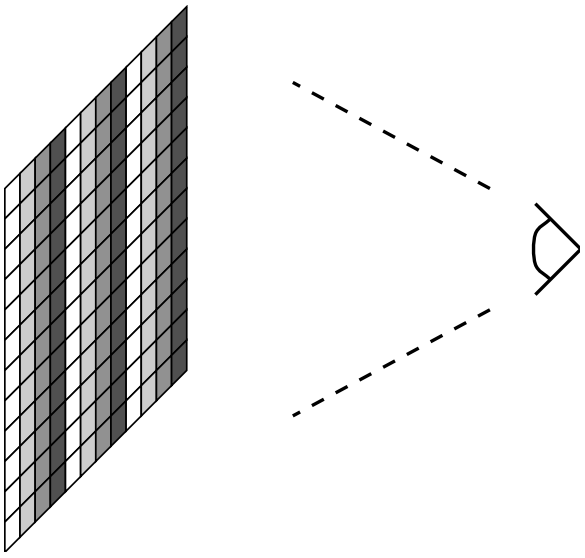


Figure 18: Parallel raytracing: the pixels of the virtual screen are divided in a cyclic manner. Each level of gray in the image is on a different processor.

number of CPUs. This division guarantees an equal distribution of work, since each CPU participates in the calculation of each region, and in general all of the CPUs perform similar calculations.

This division of data is illustrated in Figure 18.

Due to the simplicity of the communications an the equal distribution of work, we implemented this second distribution for our application.

7.3 Complete Application Network

There are three kinds of modules that are deployed on the cluster: the fluid modules, the render modules, and the display module. It is necessary to choose a protocol for their communications and create a schema for the entire application network.

7.3.1 Communications Schema

In section 7.1, we saw that the fluid modules communicate between each other. We also saw in section 7.2 that the render nodes do not communicate with each other. Besides the inter-fluid communications, there is also the communication of simulation data from the fluid nodes to the render nodes, and the communication of pixel data from the render nodes to the display node.

Each fluid node has to communicate its ghosts to its neighbors, so there is a connection to the neighbors. It also has to communicate with all of the render nodes, and so there is a connection from each fluid node to each render node.

The render nodes require two sets of data. They need the volumetric data which comes from the simulation run on the fluid nodes, and they also need data which describes the various parameters of the virtual camera.

The simulation and render processes are decoupled, meaning that there is no link between the speed of the simulation and the speed of the render. If the simulation is relatively fast, then there can be several simulation steps for each rendered frame. On the other hand, if the simulation is relatively slow, then there can be several rendered frames for a single simulation step. To accomodate this, we put a "greedy" filter on this connection, which always accepts data from the fluid nodes, and always gives the latest available data to the render nodes.

In order to execute the raytracing algorithm, the render nodes need the matrix which describes the position, orientation, and other parameters of the virtual camera. In order to provide this, there is a connection from the display node to all of the render nodes. The display and render nodes are tightly coupled, where one rendered frame corresponds exactly to a single displayed frame, and the transmission of a matrix serves as the trigger to start the process of rendering a new frame. This is done by making the connection use a standard FIFO queueing process.

The render nodes send the pixel data that they produce to the display node, and so there is also a connection here. Like the matrix connection, this connection is a standard FIFO connection.

The display node then has two connections to each render node, one for the transmission of matrix data, and the other for the reception of pixel data.

This communications schema as applied to a network with four fluid nodes and two render nodes is shown in Figure 19.

As seen from the rest of the network, the fluid module is just a module that produces arbitrary volumetric data from an unknown source. This makes it possible to easily replace this module with another module that produces its data in a different way. For example, a static data module would be useful for displaying volumetric data stored in a file. The static data module loads the data from a file and sends it directly to the render module. We implemented a static data module which allowed us to test the performance of the render module without interference from the fluid simulation, and to view these types of files.

7.3.2 Protocols

Traditional protocols designed for distributed applications, such as the well-known MPI, work well for homogeneous applications, but are not designed for heterogeneous applications such as this one. Various libraries designed for heterogeneous applications exist, but we experienced difficulties in integrating them into this application. Since our communications schema is not too complicated, we decided to create a custom communications library for our application.

This library is a small wrapper around standard POSIX sockets using TCP. It provides functions for connecting to a host, listening for connections, and

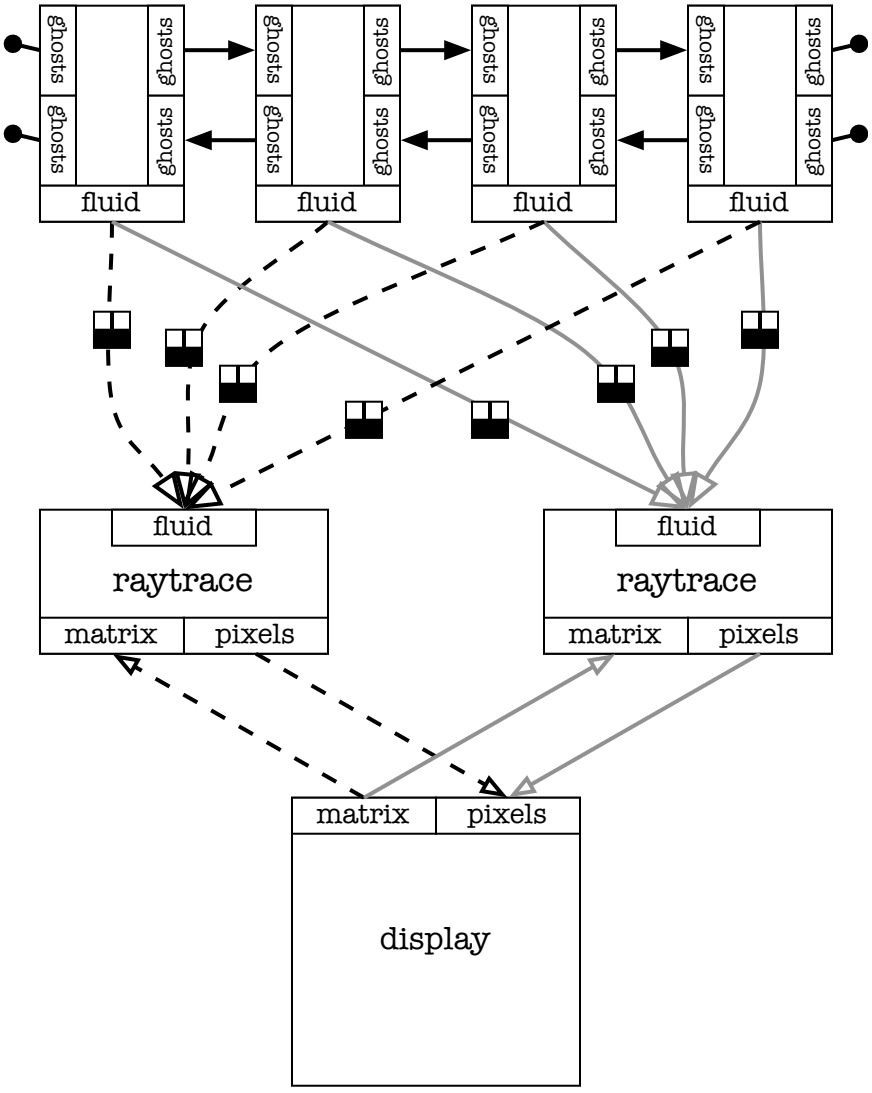


Figure 19: Communications schema for a network with four fluid nodes, two render nodes, and one display node. The boxes on the connections between the fluid nodes and the render nodes indicate a "greedy" filter.

sending or receiving atomic packets of data on multiple connections with a single function call.

All of the fluid modules in the network are basically a single large homogeneous parallel application, and so it is possible to use MPI for its internal communications. MPI implementations are typically optimized for data transfers between homogeneous nodes, and particularly for transfers between two nodes running on a single dual-processor computer, making MPI a good choice for this section of the communications network. Since the fluid module uses only the most basic functionality provided by MPI, we keep the ability to use our custom library instead of an MPI library.

Our custom communications library has no facility for constructing the entire network. We created a Python script which knows the configuration of the cluster and launches all of the various modules on different nodes of the cluster. The script stores a list of all available nodes for the fluid and render modules, and constructs a network based on the number of modules to launch, and the list of available nodes.

8 Performance Analysis

8.1 Theoretical Analysis of the Fluid Simulation

In order to analyze the simulator, we separate the program into two phases, a calculation phase followed by a communications phase. In section 7.1, we saw that the amount of data transferred in each step was equal to $2 \times 38N^2 \times 4B$ for all internal nodes. Therefore, if l is the latency of the network and B the bandwidth, we can calculate the total communications time:

$$t_1 = 38l + \frac{2 \times 38N^2 \times 4B}{G}$$

For dual-processor computer, we can remove the factor of two. It is possible to ensure that two fluid nodes placed on the same computer are also neighbors in the simulation. This causes the communications between these two nodes to remain completely internal to the computer. Given these conditions, each node communicates with at most one other node through the network.

After this first communications step, the fluid nodes send data to the render nodes. Each fluid node sends its data to all render nodes, and each render node receives all of the cube's data.

Let f be the number of fluid nodes, r the number of render nodes, and N the size of the cube, then the amount of data d_f sent by each fluid node is:

$$d_f = \frac{rN^3}{f} \times 4B$$

The amount of data d_r received by each render node is:

$$d_r = N^3 \times 4B$$

The amount of data d which limits this communications step is the maximum of these two values. If we have more render nodes than fluid nodes, $d = d_f$, otherwise $d = d_r$. The fluid simulation generally requires much more computational power, and so the application normally has more fluid nodes than render nodes, making it so $d = d_r$. Taking latency into account, the total time for this communications step is therefore:

$$t_2 = l + \frac{N^3}{G} \times 4B$$

Combining both communications steps, the total time for the communications phase is:

$$t = t_1 + t_2 = 39l + \frac{76N^2 + N^3}{G} \times 4B$$

8.2 Theoretical Analysis of the Raytracer

Compared to the fluid module, the render module communicates very little. It receives a great deal of volumetric data from the fluid module, but this data is managed by the "greedy" filter on the connection, and its communications time is counted in the fluid module's communications. The only communication which counts for the render module is its communication with the display module. For each render step, the current display matrix is sent from the display node. Next, pixel data is generated from the volumetric data and the display matrix. Finally, this pixel data is sent back to the display node. For our analysis, we can combine the two communications steps into one, by adding all of the data together and doubling the latency to account for the fact that it is actually two separate steps.

The matrix contains 16 elements of four bytes each. It is sent to all render nodes by the display node, and so we must count the bandwidth of the display node. Each pixel is a one-byte gray level. If N be the size of the cube, the number of pixels is $LOD_{planar} \times N^2$. The amount of data d transmitted is:

$$d = 64rB + LOD_{planar} \times N^2 B$$

Where r is the number of render nodes. The total communications time is therefore:

$$t = 2l + \frac{64rB + LOD_{planar} \times N^2 B}{G}$$

9 Performance Data

9.1 Scalability

Performance tests were carried out on two PC clusters, one cluster at North Dakota State University (NDSU) in the United States, and the cluster at the LIFO at the Université d'Orléans.

The NDSU cluster is not very powerful, particularly its interconnect, and this allows us to examine the computational requirements of this application. This cluster consists of 15 Power Mac G4s at 533MHz with 256MB of memory, running Mac OS X, with a 100Mbit ethernet switch as the interconnect.

The LIFO cluster was in the middle of maintenance and upgrades during these tests. The piece of the cluster which we used for these tests consisted of a cluster of 7 biprocessor PCs with 1GB of memory, running Linux, with a Gigabit Ethernet switch as the interconnect.

The fluid tests were performed on cubes of size $32x32x32$, $64x64x64$, and $128x128x128$. For each size, the number of fluid nodes was varied from 1 to the maximum possible number on the given cluster without putting two fluid nodes on the same CPU. The speeds for these tests are given in simulation steps per second.

For the tests on small static datasets, using the file bonsai256x256x256.raw [1], we started one static data node on an arbitrary node, and then we varied the number of render nodes from 1 to the maximum possible on the cluster without putting two render nodes on the same CPU.

For the tests on data from the Commissariat à l'Energie Atomique (CEA, the French atomic energy agency), we followed the same plan, but the size of the data created strong limitations on the tests that could be performed. They were too large to be tested on the NDSU cluster, both because of bandwidth problems when transporting the data, and because of the limited amount of memory in the NDSU computers. For the LIFO cluster, we were finally limited to a single render node per computer. With the CEA data, each render node required more than 512MB of memory in its working set. Even though each computer had two CPUs, putting two render nodes on the same computer caused the computer's working set to pass the 1GB of installed memory, causing the machine to thrash and its performance to fall greatly.

The Level of Detail (LOD) tests were carried out using the CEA data on the LIFO cluster with seven render nodes, the maximum possible on this cluster with the CEA data. These tests are divided into two parts, one part covering planar LOD, and one part covering depth LOD.

9.1.1 Scalability Analysis of the Fluid Simulation

In this section, we calculate the theoretical communications time for the fluid simulation using real-world network performance data, in order to validate the theoretical analysis and to compare with the performance obtained in testing. We assume that the bandwidth for a 100Mbit network is $10MB/sec$, and a 1Gbit network is $100MB/sec$. Typical latency on an ethernet network is $350\mu sec$ [20]. Using these numbers, we calculate the communications time with a single render node and with different values of N, the size of the cube. We also calculate the time required for only the second communications step, transmitting data from the fluid module to the render module.

Bandwidth	N	Latency	Time → render	Total Comms. Time
10	32	$350\mu sec$	$13.1ms$	$58ms$
10	64	$350\mu sec$	$105ms$	$243ms$
10	128	$350\mu sec$	$839ms$	$1351ms$
100	32	$350\mu sec$	$1.3ms$	$17ms$
100	64	$350\mu sec$	$10.4ms$	$30ms$
100	128	$350\mu sec$	$84ms$	$122ms$

This gives us a theoretical communications time. We also have the total time required for each simulation step which was measured in performance tests on a single node. With these two values, we can calculate the total computation time required for a single simulation step by taking the difference between the total time and the communications time. The time required for the computation phase on an arbitrary number of nodes is inversely proportional to the number of nodes.

Bandwidth	N	Total Time	Comms. Time	Computation Time
10	32	$103ms$	$13.1ms$	$90ms$
10	64	$1205ms$	$105ms$	$1100ms$
10	128	$11111ms$	$839ms$	$10272ms$
100	32	$71ms$	$1.3ms$	$69ms$
100	64	$1010ms$	$10.5ms$	$1000ms$
100	128	$20000ms$	$84ms$	$19916ms$

In this table, `Total Time` is the performance measured with a single fluid node, `Comms. Time` is the theoretical communications time with a single node, and `Computation Time` is the difference between the two, the amount of time spent in the computation phase.

Having this computation time, we can then perform a theoretical calculation of the scalability of the parallel fluid simulation. If t_c is the computation time on a single node, and f is the number of fluid nodes, we can calculate the total time required for a full simulation step, with both computation and communication phases:

$$t = \frac{t_c}{f} + 39l + \frac{76N^2 + N^3}{G} \times 4B$$

This produces times which are very close to actual real-world data. Comparisons between the theoretical results computed using the above equation and real-world test results are provided in sections 9.3.1 and 9.3.2.

9.2 Scalability Analysis of the Raytracer

We calculate the theoretical communications time for the render module in the same fashion as for the fluid module. We use the same network performance values as for the fluid module analysis: bandwidth at $10MB/sec$ and $100MB/sec$, and a latency of $350\mu sec$. The LOD_{planar} is set to 1.

Bandwidth	N	Latency	Total Time	Communication	Computation
10	256	$350\mu sec$	$2439ms$	$7ms$	$2432ms$
100	256	$350\mu sec$	$403ms$	$1ms$	$402ms$
100	512	$350\mu sec$	$4000ms$	$3ms$	$3997ms$

Immediately, we see that the computation time is enormous compared to the communication time, as the computation phase takes two to three orders of magnitude longer. The predicted speeds are therefore almost exactly proportional to the number of render nodes.

Real-world performance test data for the render module is given in parts 9.3.3, 9.3.4, and 9.3.5. The actual performance of the render module is far from linear. On the maximum number of nodes, the difference is between $40 - 100\%$.

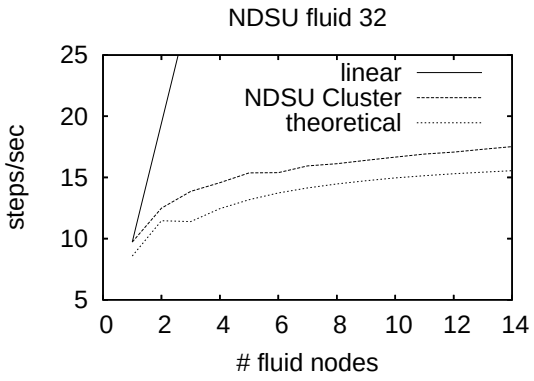
The reasons for this difference are not clear. The raytracing algorithm is extremely parallel with no dependencies between the various raytracing nodes.

9.3 Performance Tests

9.3.1 Fluid Simulation on the NDSU Cluster

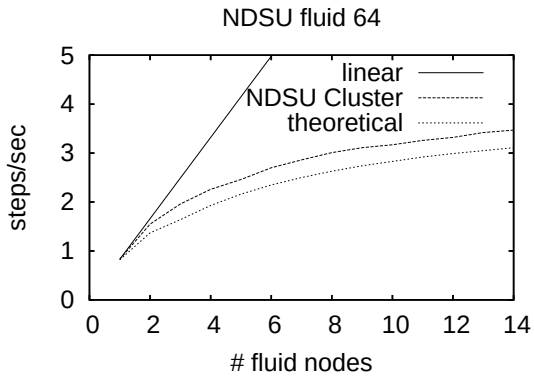
The following table contains test results for fluid simulation in a size 32 cube on the NDSU cluster:

Nodes	Real	Theoretical
1	9.72	8.58
2	12.49	11.46
3	13.86	11.39
4	14.57	12.45
5	15.37	13.18
6	15.39	13.73
7	15.94	14.14
8	16.12	14.47
9	16.40	14.73
10	16.66	14.96
11	16.91	15.14
12	17.07	15.30
13	17.30	15.43
14	17.52	15.55



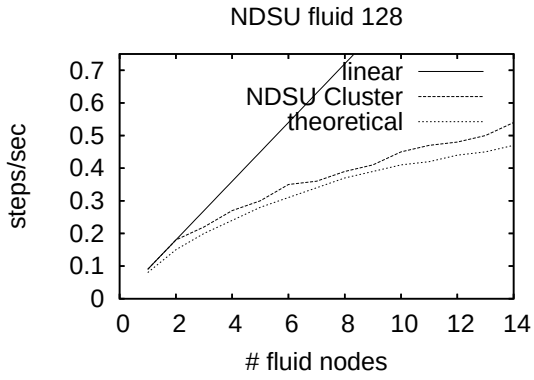
The following table contains test results for fluid simulation in a size 64 cube on the NDSU cluster:

Nodes	Real	Theoretical
1	0.83	0.82
2	1.55	1.37
3	1.96	1.64
4	2.26	1.93
5	2.46	2.16
6	2.70	2.35
7	2.86	2.50
8	3.01	2.63
9	3.11	2.74
10	3.17	2.83
11	3.26	2.92
12	3.32	2.99
13	3.42	3.05
14	3.47	3.11



The following table contains test results for fluid simulation in a size 128 cube on the NDSU cluster:

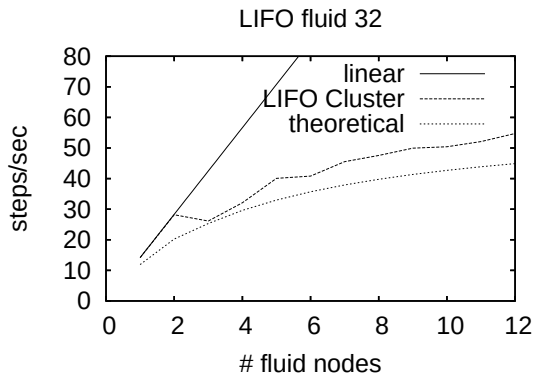
Nodes	Real	Theoretical
1	0.09	0.08
2	0.18	0.15
3	0.22	0.20
4	0.27	0.24
5	0.30	0.28
6	0.35	0.31
7	0.36	0.34
8	0.39	0.37
9	0.41	0.39
10	0.45	0.41
11	0.47	0.42
12	0.48	0.44
13	0.50	0.45
14	0.54	0.47



9.3.2 Fluid Simulation on the LIFO Cluster

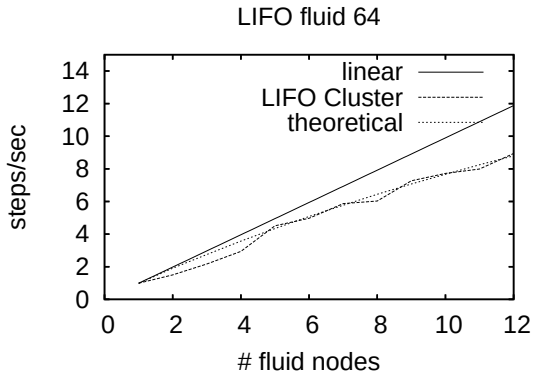
The following table contains test results for fluid simulation in a size 32 cube on the LIFO cluster:

Nodes	Real	Theoretical
1	14.16	11.91
2	28.17	20.22
3	26.14	25.31
4	32.11	29.61
5	40.08	32.98
6	40.81	35.69
7	45.55	37.92
8	47.57	39.77
9	49.94	41.35
10	50.40	42.70
11	52.11	43.88
12	54.74	44.91



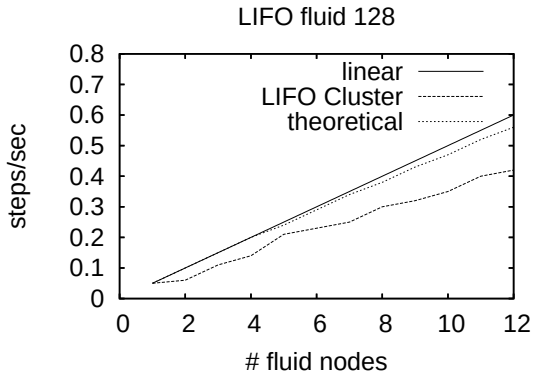
The following table contains test results for fluid simulation in a size 64 cube on the LIFO cluster:

Nodes	Real	Theoretical
1	0.99	0.98
2	1.50	1.91
3	2.17	2.75
4	2.94	3.57
5	4.51	4.34
6	4.97	5.08
7	5.87	5.77
8	6.03	6.44
9	7.27	7.07
10	7.72	7.67
11	7.99	8.25
12	8.94	8.80



The following table contains test results for fluid simulation in a size 128 cube on the LIFO cluster:

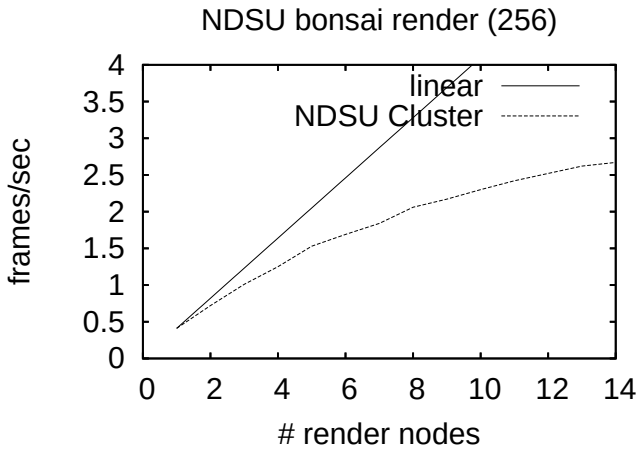
Nodes	Real	Theoretical
1	0.05	0.05
2	0.06	0.10
3	0.11	0.15
4	0.14	0.20
5	0.21	0.24
6	0.23	0.29
7	0.25	0.34
8	0.30	0.38
9	0.32	0.43
10	0.35	0.47
11	0.40	0.52
12	0.42	0.56



9.3.3 Static Data on the NDSU Cluster

The following table contains test results for static data from the file bonsai256x256x256.raw on the NDSU cluster:

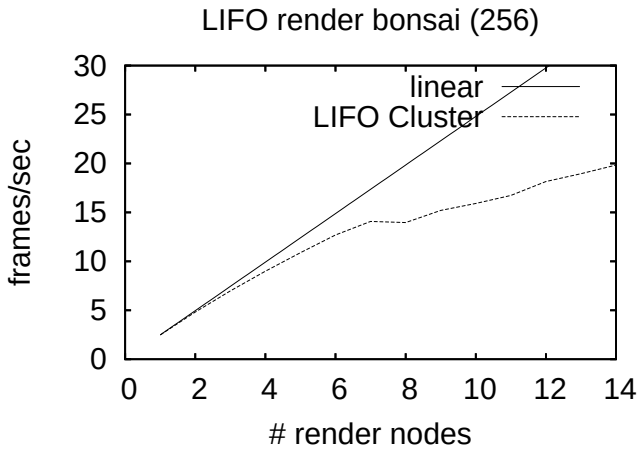
Nodes	FPS
1	0.41
2	0.72
3	1.01
4	1.25
5	1.53
6	1.69
7	1.84
8	2.06
9	2.17
10	2.30
11	2.42
12	2.52
13	2.62
14	2.67



9.3.4 Static Data on the NDSU Cluster

The following table contains test results for static data from the file bonsai256x256x256.raw on the LIFO cluster:

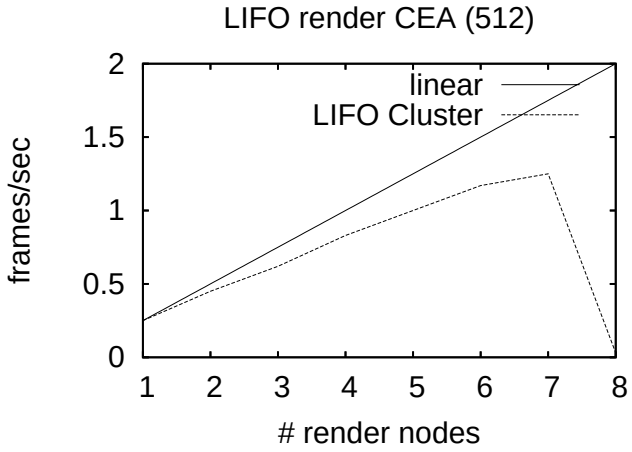
Nodes	FPS
1	2.48
2	4.82
3	6.98
4	8.99
5	10.88
6	12.69
7	14.08
8	13.96
9	15.20
10	15.91
11	16.73
12	18.15
13	18.95
14	19.84



9.3.5 CEA Data on the LIFO Cluster

The following table contains test results for static data from the CEA of size $512x512x512$ on the LIFO cluster. Performance drops to near zero with eight nodes due to the working set growing larger than available memory on one node:

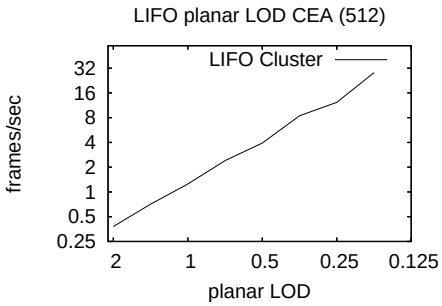
Nodes	FPS
1	0.25
2	0.45
3	0.62
4	0.83
5	1.00
6	1.17
7	1.25
8	0.03



9.3.6 LOD Tests Using CEA Data on the LIFO Cluster

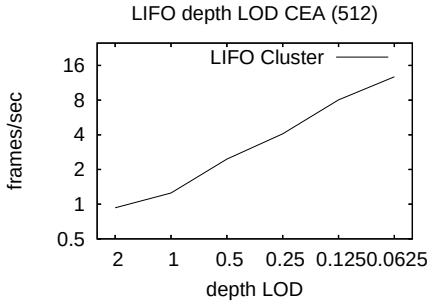
The following table contains test results for the planar LOD algorithm using static data from the CEA of size $512x512x512$ on the LIFO cluster. The data is graphed using a logarithmic scale, and the optimal performance is a straight line:

Nodes	FPS	LOD_{planar}
7	0.38	2.0
7	0.71	1.414
7	1.25	1.0
7	2.40	0.707
7	3.93	0.5
7	8.44	0.3536
7	12.28	0.25
7	28.30	0.1765



The following table contains test results for the depth LOD algorithm using static data from the CEA of size $512x512x512$ on the LIFO cluster. The data is graphed using a logarithmic scale, and the optimal performance is a straight line:

Nodes	FPS	LOD_{depth}
7	0.93	2.0
7	1.25	1.0
7	2.46	0.5
7	4.08	0.25
7	8.04	0.125
7	12.75	0.0625



10 Conclusion

The objective of this work was to create an interactive 3D fluid simulation application and to explore various ways to process large quantities of data by more efficiently using computer hardware, and by deploying the application on PC clusters.

The fluid simulation is based on a 2D simulation proposed by Jos Stam. The goal of this simulation is to extend Stam's proposal into three dimensions while keeping its real-time properties and the quality of the output.

A 3D fluid simulation produces volumetric data. To display this data to the user, we studied current volumetric rendering techniques, and we implemented

three techniques in order to perform comparisons. The raytracing algorithm was chosen for its flexibility and in particular its suitability for parallelization.

Optimizing the components of the application allowed us to increase the speed of the raytracer by a factor of three, and the fluid simulation by a factor of nearly seven, on a single CPU.

For more speed, we parallelized the application in order to deploy it on a PC cluster. We created a custom networking library to manager the communications between the various components of the application. For the communications within the fluid module, the standard MPI interface gave good results.

We performed a theoretical performance analysis on the fluid and render modules, and tested the application on two PC clusters in order to verify the analysis. The real-world performance of the fluid simulation was close to the predicted performance, with a difference of no more than 20%. The maximum speedup attained was a factor of nine, on 12 CPUs of the LIFO cluster. The real-world performance of the raytracer was much lower than the theoretical analysis predicted, with a maximum ppeedup of eight on 14 CPUs, where the analysis suggested a speedup of almost 14. The reasons for this difference are not clear.

With the simulation in place on the LIFO cluster, the linear dimensions of the simulation were able to be doubled from 32 on a single CPU to 64 on the cluster without any performance loss. Doubling the linear dimensions increases the amount of data by a factor of eight, showing how the parallelization provided significant gains. The parallel raytracer was able to display extremely large datasets with a good image quality at nearly 20 frames/second, greater than the minimum considered necessary for scientific visualization, which is 15 frames/second, and the speed obtained was eight times faster than the single-CPU version. We also demonstrated the effectiveness of the Level of Detail (LOD) variables, which allow a tradeoff between image quality and rendering speed. The result is a high-performance fluid simulation and volumetric data display application.

10.1 Future Directions

The final application still has room for improvement. With the limited time for this project and the desire to create a working application, there are some refinements which were not implemented.

10.1.1 Links Between Simulation and Rendering

Frequently in the fluid simulation, there are large regions of space which remain empty. By discovering the location of these empty spaces and skipping them, the render module would be able to decrease the amount of data processed and increase the speed of its results.

The simplest way to accomplish this is to search for the minimum and maximum coordinate along each axis which contains fluid, which then describes a bounding box containing all of the fluid. While simple, the effectiveness of

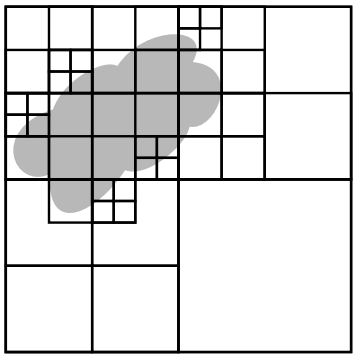


Figure 20: An octtree describing a fluid cube.

this approach depends on the conditions of the simulation. If the simulated space contains small bits of fluid that are far apart, the bounding box could be excessively large, limiting the potential speed gain.

Another possibility would be to create an octtree (tree with eight children per node) for the cube. The root of the tree contains a single node containing the entire cube. For each node, it can either be divided into eight subcubes, or it can be left intact. With a simple value of *empty* or *non-empty* on each leaf node, the entire state of the cube can be described to the render module with very little data. The resolution of the tree could be limited in order to trade off between tree construction time and final render time. This tree could also be used to limit the amount of data that must be transmitted to the render nodes. An example of an octtree used to describe a fluid cube is illustrated in Figure 20.

10.1.2 Simulation Calculation Strategies

In section 5.4, we saw one possibility for improving the fluid simulation's memory access patterns. It would require a complete reworking of the fluid solver in a difficult-to-manage fashion, but with a good potential for improved performance.

10.1.3 Asynchronous Communications

Traditional analysis of parallel applications considers communication and calculation to be completely separate phases, where the processor does nothing during the communication phase, and the network does nothing during the calculation phase. However, most systems allow asynchronous communication, where the processor continues to calculate while the network transmits data.

In the fluid simulation, there is a great deal of opportunity for this kind of optimization. For example, the simulation contains three successive calls to the `advect` function on three completely different sets of data. It would therefore

be possible to transmit the results of one call while the second call was executing, rather than waiting for the transmission to complete. If the communication were faster than the computation, then this would effectively remove two communications substeps from each simulation step. If the computation is faster than the communication, then it effectively removes two computation substeps. Other possibilities include the last communication substep in `lin_solve` and the last communication substep in `project`. Managing these communications becomes much more difficult. It is necessary to analyze all of the dependencies between the various calculations and insert code which blocks execution until the necessary data has been fully received for each substep that requires it. The simulation's speed is greatly limited by the time spent communicating, which makes this a promising improvement.

Transmission of data to the render module is also promising. The fluid module sends density data which is calculated at the end of the simulation step, and which are not modified anywhere else. Changing this communication substep to be asynchronous would allow almost the entirety of the following simulation step to transmit this data, making it likely that this communication would become effectively free.

The render module is less limited by its communication, and it is coupled to the display module. It would be possible to shift the rendering process by introducing a one-step latency between the display and render modules. This way, when the render module finishes one frame, the matrix needed to produce the next frame would have already been received, and so the render module could begin rendering the next frame while it was transmitting the previous one to the display module.

References

[1] Dirk Bartz, Stefan Gumhold Universität Tübingen http://www.gris.uni-tuebingen.de/areas/scivis/volren/datasets/datasets.html

[2] Scientific Visualization Laboratory, Georgia Tech "Scientific Visualization Tutorial" http://www.cc.gatech.edu/scivis/tutorial/tutorial.html

[3] Dan Baker, Charles Boyd "Volumetric Rendering in Realtime" *Proceedings of the 2001 Game Developers Conference*, 2001 http://www.gamasutra.com/features/20011003/boyd_pfv.htm

[4] Paul S. Calhoun, BFA, Brian S. Kuszyk, MD, David G. Heath, PhD, Jennifer C. Carley, BS, Elliot K. Fishman, MD "Three-dimensional Volume Rendering of Spiral CT Data: Theory and Method" *Radiographics*, 1999;19:745-764. http://radiographics.rsnajnls.org/cgi/content/full/19/3/745

[5] Chuck Hansen, Michael Krogh, James Painter, Guillaume Colin de Verdière, Roy Troutman "Binary-Swap Volumetric Rendering on the T3D" *Proceedings of the Cray Users Group 1995 Spring Conference*, 1995 http://www.ccs.lanl.gov/ccs1/projects/Viz/pdfs/95-cug95.pdf

[6] David S. Ebert, Roni Yagel, Jim Scott, Yair Kurzion "Volume Rendering Methods for Computational Fluid Dynamics Visualization" *IEEE Visualization*, 1994 http://dynamo.ecn.purdue.edu/~ebertd/papers/vis94.ps.Z

[7] Jos Stam "Real-time Fluid Dynamics for Games" *Proceedings of the Game Developers' Conference*, 2003 http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/GDC03.pdf

[8] Wei Hong, Feng Qiu, Arie Kaufman Stony Brook University "Hybrid Volumetric Ray-Casting" http://www.cs.sunysb.edu/~vislab/projects/gpgpu/hybrid.pdf

[9] Jens Krüger, Rüdiger Westermann TU-München "GPU Simulation and Rendering of Volumetric Effects for Computer Games and Virtual Environments" http://wwwcg.in.tum.de/Research/data/Publications/eg05.pdf

[10] Timothy J. Purcell, Ian Buck, William R. Mark, Pat Hanrahan Stanford University "Ray Tracing on Programmable Graphics Hardware" http://graphics.stanford.edu/papers/rtongfx/rtongfx.pdf

[11] Peter Shirley, Allan Tuchman University of Illinois "Polygonal Approximation to Direct Scalar Volume Rendering" http://www.cs.utah.edu/~shirley/papers/polygonal.pdf

[12] William E. Lorensen, Harvey E. Cline "Marching Cubes: A High Resolution 3D Surface Construction Algorithm" *Computer Graphics (Proceedings of SIGGRAPH '87)*, Vol. 21, No. 4, pp. 163-169

[13] James Sharman "The Marching Cubes Algorithm" http://www.exaflop.org/docs/marchcubes/ind.html

[14] Robert Fraser SGI Computer Systems Advanced Systems Division "Interactive Volume Rendering Using Advanced Graphics Architectures" http://web.archive.org/web/20001022092548/http://www.sgi.com/Technology/volume/VolumeRendering.html

[15] Rändy Osborne, Hanspeter Pfister, Hugh Lauer, Neil McKenzie, Sarah Gibson, Wally Hiatt, TakaHide Ohkami Mitsubishi Electric Research Lab "EM-Cube: An Architecture for Low-Cost Real-Time Volume Rendering" *Proceedings of the 1998 IEEE symposium on Volume visualization*, pp. 31-38 http://online.cs.nps.navy.mil/DistanceEducation/online.siggraph.org/2001/Courses/cd1/courses/13/Ws97.pdf

[16] Erik de Castro Lopo "Faster Floating Point to Integer Conversions" http://mega-nerd.com/FPcast/

[17] Apple Computer, Inc. "Software Pipelining" http://developer.apple.com/hardware/ve/software_pipelining.html

[18] Motorola, Inc., Freescale Semiconductor, Inc. "MPC7450 RISC Microprocessor Family User's Manual" http://www.freescale.com/files/32bit/doc/ref_manual/MPC7450UM_ZIP.zip

[19] Alexei Lebedev "Fixed Point Math For Speed Freaks" *MacTech*, Vol. 10, No. 3. http://www.mactech.com/articles/mactech/Vol.10/10.03/FixedPointMath/

[20] "Performance and Capacity Management Planning Guide for Solaris OS" *Sun Microsystems Blueprints*, CD 3, 2005-08-07

[21] Sylvain Jubertie "Techniques de parallélisation de pré-rendu pour la réalité virtuelle sur grappe de PC" *Laboratoire d'Informatique Fondamentale d'Orléans, Université d'Orléans*, Septembre 2003

[22] Jérémie Allard, Valérie Gouranton, Loïck Lecointre, Sébastien Limet, Emmaneul Melin, Bruno Raffin, Sophie Robert "FlowVR: a Middleware for Large Scale Virtual Reality Applications" *Euro-Par 2004 Parallel Processing. Proceedings.*, 2004 http://www-id.imag.fr/~raffin/papers/ID/europar04.pdf

[23] Jérémie Allard, Valérie Gouranton, Loïck Lecointre, Sébastien Limet, Emmaneul Melin, Bruno Raffin, Sophie Robert "FlowVR : Coupling Distributed Codes for High Performance Interactive Applications" http://flowvr.sourceforge.net/doc/flowvr/flowvr-manual.pdf