Real-time Visualization of Smoke through Parallelizations

Torbjørn Vik[a], Anne C. Elster[a] and Torbjørn Hallgren[a]

[a]Norwegian University of Science and Technology, Department of Computer and Information Science, NO-7491 Trondheim, Norway

Animation/visualization of natural phenomena such as clouds, water, fire and smoke has long been a topic of interest in computer graphics, and following the increase in common processing power researchers have started using physically based simulations to generate these animations. Smoke, and other fluids, is extremely hard to animate by hand, and simulation techniques producing realistic results are highly applicable and desirable in areas ranging from computer games to weather forecasts.

Our work combines mathematical, graphical and technical techniques including several parallel ization techniques in order to produce a *real-time* code that visualizes rolling and twirling smoke in three dimensions. The resulting 3D parallel implementation exploits dual cpu hardware in order to produce more than 20 image frames per second on today's modern desktop computers.

Keywords:*real-time, parallel, visualization smoke, CFD, CG, pipelining, SIMD instructions*

1. INTRODUCTION

Early researchers in the field of visual fluid simulations produced realistic results by applying mathematically generated textures to simple geometric structures [4,9]. This approach did not count for interaction with solid objects and left the animators with the tedious work of parameter tweaking.

The models and algorithms presented in [3,11,2], all demonstrate a clear evolution within visual simulations of fluids: In 1997, Foster & Metaxas published their paper, *Modeling the Motion of a Hot, Turbulent Gas* [2]. They extracted the parts of Navier-Stokes' equations that affect the interesting visual features of smoke, such as convection and vorticity, thus creating a model capable of simulating realistic smoke on relatively coarse grids. Their model was only stable for small time steps because of an explicit integration scheme; large velocities required small timesteps, and the result was animations evolving too slowly.

Stam was the first to present a stable algorithm solving the full Navier-Stokes equations [11], independent of the time steps and implemented a version that *"allows an animator to design fluid-like motions in real time"*. This was achieved using a semi-Lagrangian approach together with implicit solvers, allowing for much larger time steps and a faster evolving simulation. Fedkiw, Stam and Jensen [3] pointed out that the first-order integration scheme used by Stam caused too much numerical dissipation and small-scale details just tended to die out. Fedkiw et al. presented a model aiming for visual simulation of *smoke*, compared to the general visual fluid simulator of Stam. Their model was based on the work of Foster & Metaxas [2] and Stam [11], but featured some modifications for improved visual quality. They reported frame rates from 1-10 fps for their fastest, low-quality simulations.

For a real-time environment aiming for a visual imitation of the real world, these frame rates are, however, still not sufficient. We present a configuration of parameters, sub-algorithms and numerical solvers, all analyzed with respect to visual quality, computational efficiency and parallelism, eventually aiming for frame rates of 20 and above on today's modern desktop hardware.

2. SIMULATION MODEL

Our simulation model is based on the one presented by Fedkiw et al. [3], and assumes that smoke can be modelled as an inviscid incompressible fluid. The algorithm dices the computational doimain

into uniform sub-cubes known as *voxels* and uses two 3D grids – one grid storing the data for the previous frame, the other the data for the current frame. Each grid defines the velocity vector u, and values such as density ρ, temperature T, and user-defined forces (e.g. wind).

The algorithm consists of 5 numerical steps followed by two steps handling the lighting and rendering of the simulation data. The numerical equations for these steps are found in [3,16].

1. **Update Velocities** by scaling the forces, including the bouyancy force (gravity and effects of hot air rising) and user-defined forces, with the timestep Δt.
2. **Self-Advect velocities** using a semi-Lagrangian (SL) method [11].
3. **Solve Poisson's Equation** for the pressure using CG
4. **Update (project) temporary velocities** by subtracting the pressure gradient scaled by the time step in order to conserve mass.
5. **Advect scalar values**, i.e. update density ρ and temperature T using SL similarly as step 2
6. **Calculate lighting**
7. **Render image**

3. MODEL OPTIMIZATIONS

In order to get the best possible performance, both optimizations of the original model and parallelizations are needed. To achieve this, a 2D prototype that was analyzed and profiled in order to pinpoint possible processing hotspots in the algorithm (See Figure 1). This version defined the velocities at voxel centers as in Fedkiw et al. [3], and showed that the two advection steps occupied about 90% of the CPU time.

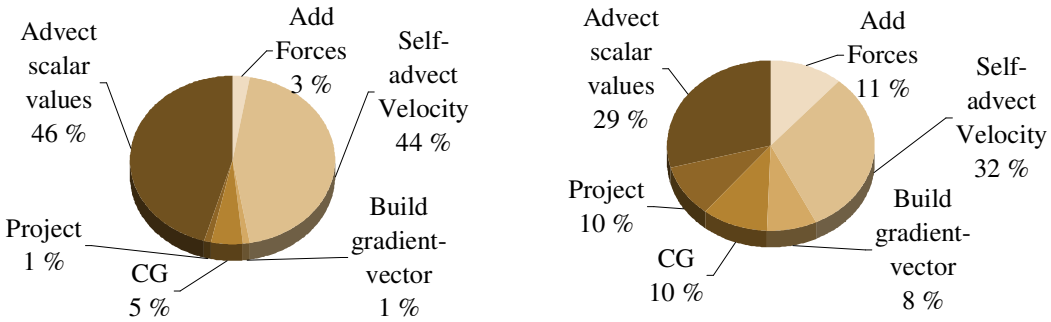

Figure 1. Profiling results of 2D (left) and 3D (right) serial implementation. Rendering disabled, no SIMD extension used, 5 CG iterations.

Our 3D implementation adopted the approach of Stam[11] and Foster et al.[2], with velocities defined at voxel centers, thus reducing the number of linearly interpolated off-grid values. This also helped our parallel implementation.

3.1. Poisson's Solver

Solving Poisson's equation requires a linear solver handling sparse Symmetric Positive Definite (SPD) matrices in an efficient manner, e.g. relaxation methods [2,14] or iterative solvers [6,5,3,11,15, 12].

Fedk et. al uses a incomplete Cholesky preconditioner as part of their Conjugate Gradients (CG) solver [**?**]. However, since the time-steps and changes between each frame in real-time smoke simulations are relatively small, we skip the pre-conditioner and use the solution from the previous frame as an initial guess. As a result, our code only uses 5 iterations to converge to an acceptable level. A detailed comparison of various PETSc [10] iterative solvers and our solver is presented in [16].

4. PARALLELIZATION TECHNIQUES

Each computational step described in the previous section, consists of one or more iterations over all the voxels in the domain. Iterative operations are usually very well suited for parallism, simply by dividing the iterative constructs and assigning each part to a node. In addition, the calculation of one frame is dependent on the results from Steps 1-6 (described in Section 2) of the previous frame. However, the rendering step - Step 7 - of the previous frame may overlap with the computations of the current frame.

We implemented the algorithm as one data-parallel and one modified data parallel version, where the latter also supports pipelining techniques.

4.1. Data Parallel Implementation

This version divides the different iterative constructs of the algorithm into smaller domains and distributes them to the different nodes. All administration and synchronization were performed using Win32 API calls, and does add some overhead, but when running the simulation on relatively small number of CPUs (2 - 4), this overhead should be negligible.

Section 5.1 presents benchmarks and analysis of the data-parallel approach, and shows a significant speedup, but also uncovers that one of the CPUs is left idle while the other is running the OpenGL rendering code.

4.2. Pipelined Implementation

As mentioned, steps from two different frames can be interleaved, as their dependencies do not require them to be processed in exact sequential order. Our implementation exploits this independence, by using a round-robin queue to distribute jobs among the worker threads and administrate the dependencies.

The efficiency of this technique depends on the performance of the graphics card, or to be more specific, how much time that is spent performing the rendering compared to the time spent performing the numerical simulation and lighting. Optimal performance is thus achieved when these two time slots are equal. If the rendering step completes too quickly, the other CPU will not be able to process anything of the following frame. In this case, the pipelined approach will probably slow down the system due to the overhead added through the queueing system.

4.3. Vectorization on Intel CPUs

The recent Intel families of processors include a set of SIMD instructions, known as SSE (Streaming SIMD Extension) which process values in groups of four. Hence, these processors can theoretically perform up to four times as many calculations per clock cycle.

We modified most of the simulator to exploit these SSE vector instructions, and experienced about 10% average speedup. Some parts of the algorithm proved to be better suited for vectorization than others, especially the CG method. The more complex steps, such as the advections, require linear interpolation of off-grid values using eight different on-grid values not placed contiguously in memory. An SSE version of these steps hence requires the values to be gathered into special vector data types in groups of four, and adds overhead to the code, thus reducing the speedup.

SIMD instructions may increase the memory bus usage significantly compared to standard floating-point operations since the bus must deliver more data to keep up with the parallel computations. In other words, the memory bus may end up being an even tighter bottleneck than before.

4.4. Parallelization of lighting calculations

The lighting calculations cannot be treated as ordinary iterative operations, since the order of traversal (of the voxels) depends on the light's direction and where the rays enter the volume. We parallelized Fedkiw et al's lighting model [3] by grouping the light rays into distinct sets and assigning each set to a thread. Our model supports parallel light rays entering the volume from an arbitrary angle.

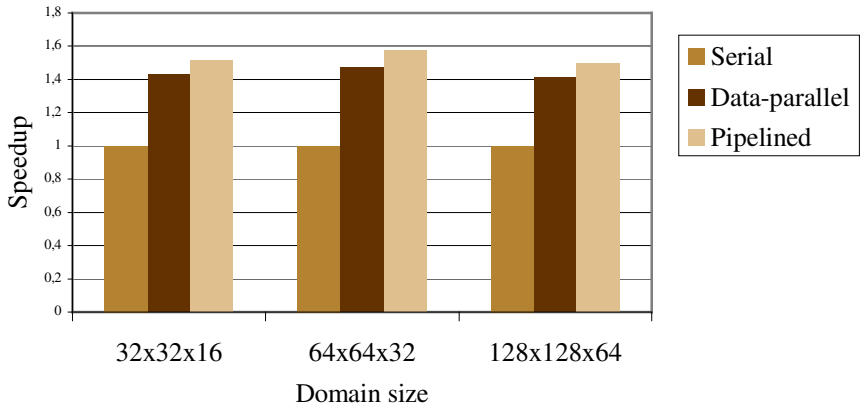

Figure 2. The speedup of data parallel and pipelined implementation vs. serial.

5. RESULTS

5.1. Results for parallel implementations

Figure 2 shows the speedup gained by our data-parallel implementation, compared to the serial version and pipelined version. The chart shows that there is indeed a speedup with the data parallel implementation, varying from 20% to 60% depending on the simulation parameters. Still, 60% speedup when moving from one to two CPUs is far from linear scaling. A partial explanation is that one thread is performing rendering while the other threads stall for some time, waiting for the first to finish. This is confirmed by the numbers reported by the MS Task Manager, giving a CPU utilization ranging from 60% for coarse grids to 90% for fine grids. However, even with rendering disabled, the speedup is still not higher than about 66%, indicating that there are other, more substantial, explanations. Some of the processing power is of course lost due to overhead added when our code administrates and distributes tasks among the worker threads. Most likely, the poor scaling is caused by race conditions for a shared resource somewhere in the system, and tests point out the memory bus as being the probable cause. The data required by a CPU to fulfill its work is way larger than its local cache, and the memory bus is unable to deliver data fast enough for the program to scale linearly.

Figure 2 also illustrates the evident speedup gained by the pipelined approach, compared to the data-parallel, but the efficiency is most likely still limited by memory bus bandwidth.

5.2. Visuals

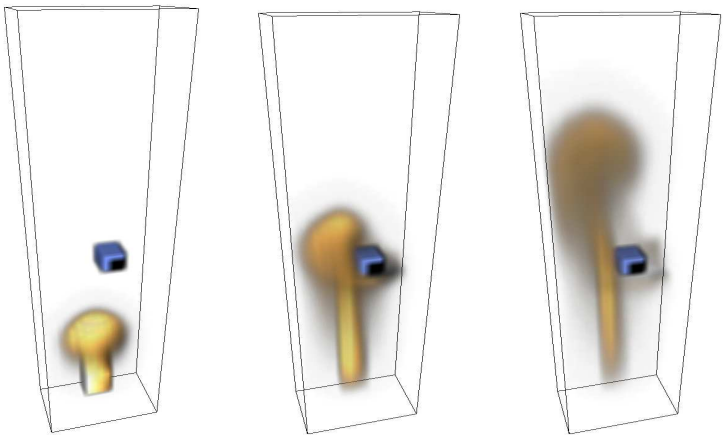

Figure 3. Frames [60, 110, 160] of our parallel simulation, 16x16x64 grid, 5 CG iterations, 40 fps.

Figure 3 presents a series of pictures from our implementation. It runs about 40 frames per second on a Dual Intel Xeon 1.7 GHz, simulating 16 384 voxels. The smoke moves in a realistic fashion and

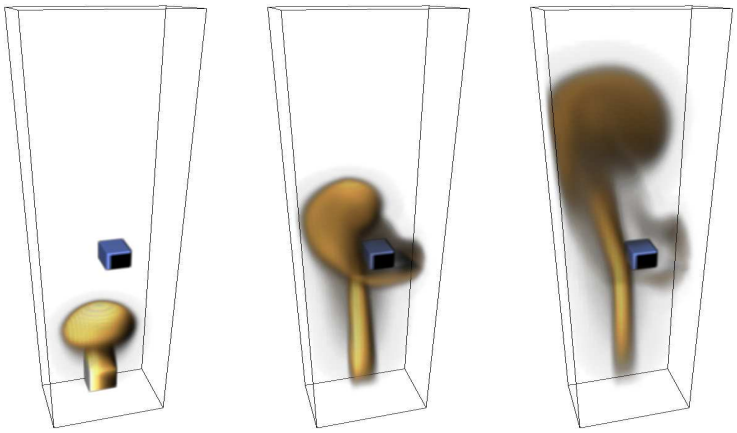

Figure 4. Frames [60, 110, 160] of our parallel simulation, 32x32x128 grid, 5 CG iterations, 4.1 fps.

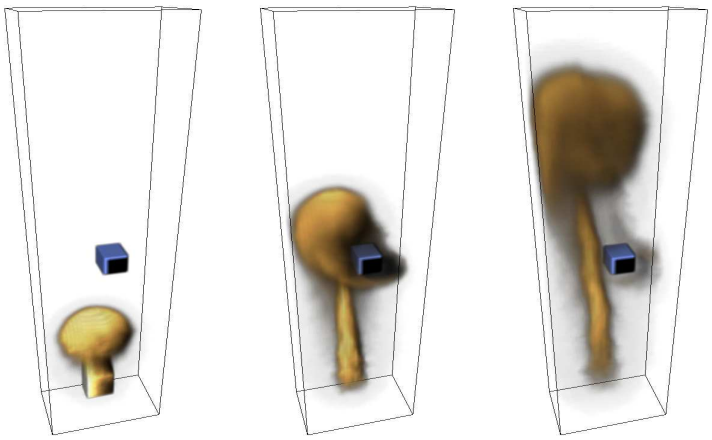

Figure 5. Frame 60, 110 and 160 of a smoke simulation, grid sized 32x32x128, 5 CG iterations, 4.1 fps. Random forces.

interacts correctly with the obstacle present in the domain, but suffers from the numerical dissipation and lack of detail pointed out by Fedkiw et al. [3].

Figure 4 presents a simulation with the exact same parameters as Figure 3, but features eight times as many voxels (131 072). The detail is greatly improved, but the result is only 4 frames per second.

Figure 5 illustrates the effect of introducing small random forces into the domain, and show, in comparison with Figure 4, that the smoke looks less "smooth" and for some situations more realistic.

6. CONCLUSION

Our results showed that it is possible to run an optimized implementation of the algorithm outlined by Fedkiw et al.[3] that yields 5 to 40 frames per second on today's modern PC hardware. Algorithmic optimizations, parameter tuning and efficient programming techniques made the implementation capable of animating smoke, still with a satisfactory level of realism and speed in real-time.

First we modified the algorithm by sacrificing accuracy for speed through the following:

- *Defining the velocity at voxel centers instead of voxel faces*
- *Fast matrix-free CG (no preconditioner)*
- *Leaving out the vorticity confinement*

These three decisions unquestionably reduced the visual quality of the smoke, a fact supported by a comparison of the visual results presented by Fedkiw et al.[3] with the visual results of this our implementation (see section 5.2). Nevertheless, the modifications also reduced the computational complexity of the algorithm, thus making it possible to calculate significantly more frames per second.

It is interesting to notice that the number of CG iterations made little or no difference on the visual simulations; a surprising result when seen in the light of earlier discussions [3]. This effect should be investigated further.

The speedups gained by parallelizing our simulation model proved it very suitable for parallelism, but also revealed the memory bus as being a limitation for memory intensive tasks. The pipelined approach presented in 4.2 introduced a technique for dynamic job distribution and pipelining, thus reducing the penalty caused by serial sub-tasks such as rendering. The technique of pipelining is well-known, especially within computer hardware engineering, but our results showed its relevance within the field of CFD visualization. Since most of today's SMP systems still share a common graphics card, the technique should also apply to a wide range of visualization software. The results also showed that certain parts of the algorithm gained significant speedups as a result of Intel's SSE instructions, in particular the CG method.

Whether it is possible or not to run a visual CFD simulation in real-time, depends on the desired visual accuracy and quality. Higher software and algorithmic efficiency will allow for higher accuracy on a on a given hardware platform, but the numerical simulations will always have a potential for improvement.

REFERENCES

[1] Peter Bartello and Stephen J. Thomas. The Cost-Effectiveness of Semi-Lagrangian Advection. Dr. P. Bartello, RPN, 2121, voie de Service nord, Route Transcanadienne, Dorval (Quebec) H9P 1J3.

[2] N. Foster and D. Metaxas. Modeling the Motion of a Hot, Turbulent Gas. In SIGGRAPH 97 Conference Proceedings, Annual Conference Series, pages 181-188, August 1997.

[3] R. Fedkiw, J. Stam, and H. W. Jensen. Visual Simulation of Smoke. In SIGGRAPH 2001 Conference Proceedings, Annual Conference Series , pages 15-22, August 2001

[4] G. Y. Gardner. Visual Simulation of Clouds. Computer Graphics (SIGGRAPH 85 Conference Proceedings), 19(3):297-384, July 1985.

[5] Louis A. Hageman and David M. Young, *Applied Iterative Methods*, Academic Press, New York, 1981.

[6] David R. Kincaid and Anne C. Elster co-editors. *Iterative Methods in Scientific Computation II*, IMACS Series in Computational and Applied Mathematics Volume 5, IMACS, 1999.

[7] OpenGL, http://www.opengl.org

[8] OpenMP, http://www.openmp.org

[9] K. Perlin. An Image Synthesizer. Computer Graphics (SIGGRAPH85 Conference Proceedings), 19(3):287-296, July 1985.

[10] PETSc, http://www-unix.mcs.anl.gov/petsc/petsc-2/

[11] J. Stam. Stable Fluids. In SIGGRAPH 99 Conference Proceedings, Annual Conference Series, pages 121-128, August 1999.

[12] An Introduction to the Conjugate Gradient Method Without the Agonizing Pain Edition 1.25. Jonathan Richard Shewchuk, August 4., 1994, http://www-2.cs.cmu.edu/ jrs/jrspapers.html

[13] C. M. Stein, N L Max. A Particle-Based Model for Water Simulation. (SIGGRAPH 98 Conference, july 19-24).

[14] Fred T. Tracy. A Comparison of a Relaxation Solver and a Preconditioned Conjugate Gradient Solver in Parallel Finite Element Groundwater Computations. Engineer Research and Development Center, Major Shared Resource Center.

[15] Henk A. Van Der Vorst. Lecture Notes on iterative methods. Mathematical Institute, University of Utrecht, Budapestlaan, the Netherlands. June 4, 1994. http://www.hpcmo.hpc.mil/Htdocs/UGC/UGC01/paper/fred_tracy_paper.pdf

[16] Torbjørn Vik, Real-time visual simulation of Smoke, Master thesis at department of Computer and Information Science, NTNU, http://www.tihlde.org/~ torbjorv/diplom15.pdf