Ron Henderson - Dreamworks

Title: Parallelism in tool development for simulation

Summary

In this section we look at the practical issues involved in introducing parallel computing for tool development in a studio environment with a focus on simulation. We talk about the hardware models that programmers must now target for tool delivery, common visual effects platforms, libraries and other tools that developers should be familiar with in order to be more productive, and considerations for key algorithms and how those relate to important application areas.

MOTIVATION

Parallel computing is a requirement. Since microprocessor clock rates have stopped increasing, the only way to improve performance (outside of regular optimization or algorithmic improvements) is by exploiting parallelism. The good news is that parallel computing can offer dramatic speedups over existing serial code, and with modern programming models and a little practice you can quickly see impressive results.

The two dominant hardware platforms for deploying parallel programs at the moment are shared memory multicore CPUs and massively parallel GPUs. The following table shows the hardware in use at DreamWorks Animation for our production tools over the past several years:

Model	Deployed	Cores	RAM	Speed	Video Card	VRAM
HP 9300	2005	4	4 GB	2.2 GHz	Nvidia FX 3450	512 MB
HP 9400	2006	4	4/8 GB	2.6 GHz	Nvidia FX 3500	512 MB
HP 8600	2008	8	8 GB	3.2 GHz	Nvidia FX 5600	1.5 GB
HP z800 (white)	2009	8	8 GB	2.93 GHz	Nvidia FX 4800	1.5 GB
HP z800+ (gold)	2010	12	12 GB	2.93 GHz	Nvidia FX 4800	1.5 GB
HP z800+ (red)	2011	12	24 GB	2.93 GHz	Nvidia FX 5000	2.5 GB
HP z820	2012	16	32 GB	3.10 GHz	Nvidia FX 5000	2.5 GB

This reflects the industry trends of increasing parallelism, increasing memory capacity, and flat or decreasing processor clock rates. These numbers are for desktop hardware, but the hardware deployed in our data centers for batch simulation and rendering has followed a similar evolution. In addition, we maintain special clusters of machines with up to 32 cores and 96 GB memory as part of our "simulation farm" dedicated to running parallel jobs.

When developing software tools for production, obviously we have to consider the hardware available to run those tools. By raw count CPU cores represent 98% of the available "compute" capacity at the studio, and GPUs about 2% (although a more reasonable comparison is probably based on FLOPs or some similar measure of computing power). For this reason, along with the flexibility to write tools that perform well across a wide variety of problem sizes, shared memory multiprocessors are by far the dominant hardware platform for internal development.

These course notes are organized as follows. First we look at programming models for writing parallel programs, focusing on OpenMP and Threading Building Blocks (TBB). Next we look at issues around understanding and measuring performance. And finally we look at case studies for fluid simulation and liquid simulation.

PROGRAMMING MODELS

Two programming models have dominated development at DWA since we started pushing parallel programming into a much wider portion of the toolset: OpenMP and Threading Building Blocks (TBB).

The first wave of software changes in the areas of simulation and volume processing used OpenMP, a tasking model that supports execution by a team of threads. OpenMP is very easy to incorporate into an existing code base with minimal effort and disruption, making it attractive for legacy applications and libraries. It requires compiler support, but is currently available for most modern compilers and even multiple languages (C, C++ and Fortran).

As we developed more complex applications and incorporated more complex data structures, the limitations of OpenMP became more problematic and we moved to Threading Building Blocks (TBB). TBB is a C++ library that supports both regular and irregular parallelism. TBB has several advantages, in particular superior support for dynamic load balancing and nested parallelism. It should work with any modern C++ compiler and does not require any special language support.

Here is a quick summary of these two programming models from [McCool et al. 2012]:

OpenMP

- Creation of teams of threads that jointly execute a block of code
- Support for parallel loops with a simple annotation syntax
- Support for atomic operations and locks
- Support for reductions with a predefined set of operations (but others are easy to program)

Threading Building Blocks (TBB)

- Template library supporting both regular and irregular parallelism
- Support for a variety of parallel patterns (map, fork-join, task graphs, reduction, scan and pipelines)
- Efficient work-stealing load balancing
- Collection of thread-safe data structures
- Efficient low-level primitives for atomic operations and memory allocation

Early versions of TBB were fairly intrusive because they required writing functors and specialized classes in order to schedule work with the TBB task model. However, with the addition of lambda expressions in the C++11 standard the syntax for writing such expressions is much easier and dramatically simplifies the task of incorporating TBB into an application.

We also make extensive use of the Intel Math Kernel Library (MKL) [Intel 2011], a collection of high-performance kernels for linear algebra, partial differential equations, FFTs and vector math. Note that MKL uses OpenMP as its internal threading model and has basic support for controlling the number of threads used by its internal functions.

In general there are no major problems mixing these programming models in the same library or even the same application as long as you avoid **nested parallelism** that might lead to oversubscription. However, because of the growing need to compose parallel algorithms in both third-party and proprietary applications, we have adopted TBB as our standard parallel programming model.

Everything you need to get started

The vast majority of changes required to introduce parallelism using OpenMP are covered by the following pragmas:

#pragma omp parallel for
#pragma omp parallel reduction(op : variable)
#pragma omp critical
#pragma omp flush

The first two are for specifying parallel loops and reductions, and the second two are for handling critical sections and reductions that are not covered by one of the built-in operators. A critical section will be executed by at most one thread at a time and is useful for avoiding race conditions. A flush forces synchronization of a thread-local variable across all threads and can be useful before a critical section that requires updates between thread local and global variables.

The most common usage patterns in TBB are also around parallel loops and reductions. One of the nice benefits of writing algorithms with TBB is that the need for critical sections and atomic variables largely disappears, even for production code, but support is provided just in case. The roughly analogous functions in TBB are the following:

tbb::parallel_for
tbb::parallel_reduce

The syntax for calling these functions can be a little odd if you have not used them before, but is easily explained with a few examples. TBB also offers hooks for critical sections and synchronization using:

tbb::mutex
tbb::atomic

If you're a good parallel programmer with TBB you probably won't need these.

Example: Over

The first example is for a simple parallel loop to compute a linear combination of two vectors with an alpha channel for blending. This is a common operation in image and volume processing given by:

$$u \leftarrow (1 - \alpha) u + \alpha v$$

Here is an implementation in OpenMP:

It is easy to read the serial implementation and all we added was the pragma to specify the loop to execute in parallel. OpenMP will split this loop and execute a subrange using a team of threads.

Here is the same function implemented in TBB:

```
inline void
tbb_over(const size_t n, float* u,
         const float* v, const float* alpha)
{
    tbb::parallel_for(
        tbb::blocked_range<size_t>(0, n),
        [=](const tbb::blocked_range<size_t>& r)
        {
            for (size_t i = r.begin(); i < r.end(); ++i) {
                u[i] = (1.f - alpha[i]) * u[i] + alpha[i] * v[i];
            }
        }
        );
}
```

This is little more complicated but still quite readable. This form of tbb::parallel_for takes a range of work to be scheduled and a function representing the task to be executed. We're using a **lambda expression** to keep the syntax compact, but you could also use a function pointer. These two implementations should have similar performance.

Example: Dot Product

The next example is for a simple reduction common to linear algebra and iterative methods like conjugate gradient iteration. The dot product is defined as:

$$u \cdot v = \sum_{i=0}^{n-1} u_i v_i$$

The implementation in OpenMP can be written using one of the built-in reduction operators:

```
inline float
omp_dot(const size_t n, const float* u, const float* v)
{
    float result(0.f);
```

```
#pragma omp parallel reduction(+ : result)
    {
#pragma omp for
        for (size_t i = 0; i < n; ++i) {
            result += u[i] * v[i];
        }
    }
    return result;
}
```

In this example a private copy of `result` is created for each thread. After all threads execute they will add their value to the copy of `result` in the master thread. There is a relatively small number of built-in reduction operators, but we will look at how to code around this in the last example below.

Here is the equivalent function in TBB:

```
inline float
tbb_dot(const size_t n, const float* x, const float* y)
{
    return tbb::parallel_reduce(
      tbb::blocked_range<size_t>(0, n),
        float(0),
        [=](tbb::blocked_range<size_t>& r, float sum)->float
        {
            for (size_t i = r.begin(); i < r.end(); ++i) {
                sum += x[i] * y[i];
            }
            return sum;
        },
        std::plus<float>()
        );
}
```

Note that tbb::parallel_reduce requires two functors: one for the task to be executed, and a combiner function that combines results to create a final value. TBB can execute using a binary tree to reduce each subrange and then combine results, or it can chain subranges if one task picks up additional work and combines it with intermediate results.

Finally, here is the corresponding call to the MKL implementation of this function:

```
float mkl_dot(const size_t n, const float* x, const float* y) {
    return cblas_sdot(n, x, 1, y, 1);
}
```

Important: the value of sum in the functor executed by each task could be zero or could be the result from reducing another subrange, but you cannot assume it is zero!

TBB combines the results of each subrange depending on the order that threads complete. For operations like the dot product that are sensitive to accumulated roundoff error this can lead to non-deterministic results. In this case you can accumulate the sum in double precision (which might help). TBB also includes an alternative **tbb::parallel_deterministic_reduce** function that always combines subranges using a binary tree. It may have slightly lower performance.

Example: Maximum Absolute Value

Let's look at one final example of a reduction that cannot be implemented with an OpenMP built-in operator: the maximum absolute value of all elements in an array. Here is one possible implementation in OpenMP:

```
inline float
omp_absmax(const size_t n, const float* u)
{
    float vmax(0.f);
#pragma omp parallel
    {
        float tmax(0.f);
#pragma omp for
        for (size_t i = 0; i < n; ++i) {
            const float value = std::abs(u[i]);
            if (value > tmax) tmax = value;
        }
#pragma omp flush(vmax)
#pragma omp critical
        {
            if (tmax > vmax) vmax = tmax;
        }
    }
    return vmax;
}
```

Here we have a thread-local variable (`tmax`) that is used to compute the maximum value for each thread, and a global maximum (`vmax`) that is updated inside a critical section. The pragma `flush(vmax)` forces a synchronization before the start of the final reduction.

You might be tempted to write this using a shared array where each thread stores its result in a unique element of the array and the main thread does a final calculation of the max once all worker threads complete. To set this up you need to know the number of threads to allocate the array and an ID for each thread to know what index you should use to store the thread-local result. **Avoid patterns like this**. Anything requiring a call to omp_get_num_threads() or omp_get_thread_id() can almost certainly be written more efficiently. TBB does not even offer such a feature.

The implementation of maximum absolute value in TBB is essentially the same as the dot product example but with the specific methods for the reduction and combiner functions:

```
inline float
tbb_absmax(const size_t n, const float* u)
{
    return tbb::parallel_reduce(
        tbb::blocked_range<size_t>(0, n)
        float(0),
        [=](tbb::blocked_range<size_t>& r, float vmax)->float
        {
            for (size_t i = r.begin(); i < r.end(); ++i) {
                const float value = std::abs(u[i]);
                if (value > vmax) vmax = value;
            }
            return vmax;
        },
        [](const float x, const float y){ return x > y ? x : y; }
        );
}
```

We can take advantage in the combiner of the fact that all intermediate results are >= 0, so there is no need for a further check of the absolute value. In general this method will have better scaling than the OpenMP version because it avoids the synchronization.

Finally, here is the call to the corresponding function in MKL:

```
float
mkl_absmax(const size_t n, const float* x)
{
    size_t index = cblas_isamax(n, x, 1);
    return x[index];
}
```

Platform Considerations

Tools used within the visual effects industry rarely run as stand-alone systems. Instead they are generally integrated into one or more extensible Digital Content Creation (DCC) platforms where they run in conjunction with other tools for animation, model generation, texture painting, compositing, simulation and rendering. Software developers need to be aware of the difference between writing a stand-alone application that controls all memory and threading behavior, and writing a plugin for a larger environment.

The two common commercial platforms used at DWA are Houdini (SideFX Software) and Maya (Autodesk). The good news is that generally speaking there is no problem writing plugins for either system that take advantage of threading using either of the programming models discussed above. In fact, both Maya and Houdini use TBB internally for threading and so this programming model integrates particularly well.

Houdini 12 introduced a number of convenience functions that wrap TBB to make parallel programming directly using the Houdini Development Kit (HDK) more developer friendly. SideFX also reorganized their geometry library to be more parallel and SIMD-friendly, with the result that parallel programming with the HDK can be quite efficient. For more details see the Houdini parallel programming considerations elsewhere in these course notes.

In general we structure code into libraries that are independent of any particular DCC application and plugins that isolate platform-specific changes. **Library code should never impose specific decisions about the number of threads to execute**. We had problems with early library development using TBB where the scheduler was being re-initialized to a different thread pool size inside library functions. Since the scheduler is shared, this had the unintended side-effect of changing performance of other unrelated code. Keep library code independent of thread count and leave control of resources to the enclosing application.

Performance

There are a few important concepts related to measuring and understanding performance for parallel applications.

Speedup compares the execution time of solving a problem with one “worker” versus P workers:

$$\text{speedup} = S_P = \frac{T_1}{T_P},$$

where T_1 is the execution time with a single worker.

Efficiency is the speedup divided by the number of workers:

$$\text{efficiency} = \frac{S_P}{P} = \frac{T_1}{PT_P}.$$

Ideal efficiency would be 1 and we would get the full impact of adding more workers to a problem. With real problems this is never the case, and people often vastly overestimate parallel efficiency. Efficiency is a way to quantify and talk about the diminishing returns of using additional hardware for a given problem size.

Amdahl's Law relates the theoretical maximum speedup for a problem where a given fraction, f, of the work is inherently serial and cannot be parallelized, placing an upper bound on the possible speedup:

$$S_P \leq \frac{1}{f+(1-f)/P}.$$

In real applications this can be severely limiting since it implies that the maximum speedup possible is 1/f. For example, an application with f = 0.1 (90% of the work is perfectly parallel) has a maximum speedup of 10. Practical considerations like I/O can often be crippling to performance unless they can be handled asynchronously. To get good parallel efficiency we try to reduce f to the smallest value possible.

Gustafson-Barsis' Law is the related observation that large speedups are still possible as long as problem sizes grow as computers become more powerful, and the serial fraction is only a weak function of problem size. This is often the case, and in visual effects it is not uncommon to solve problems that are 1000x larger than what was considered practical just five years ago.

Asymptotic complexity is an approach to estimating how both memory and execution time vary with problem size. This is a key technique for comparing how various algorithms are expected to perform independent of any specific hardware.

Arithmetic intensity is a measure of the ratio of computation to communication (memory access) for a given algorithm. Memory continues to be a source of performance limitations for hardware and so algorithms with low arithmetic intensity will exhibit poor speedup even if they are perfectly "parallel". The examples shown earlier of dot products and simple vector operations all have low arithmetic intensity, while algorithms like FFTs and dense

matrix operations tend to have higher arithmetic intensity.

CASE STUDIES

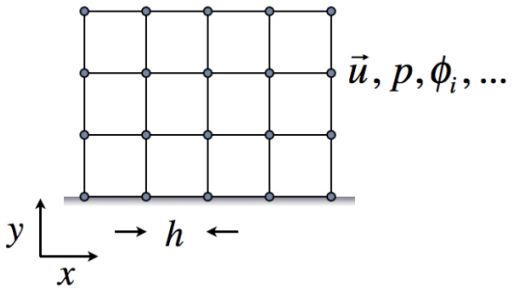
For the remainder of these course notes we'll look at case studies for two simulation problems: fluid simulation (smoke, dust, fire, explosions) and liquid simulation (water). These are closely related but use different algorithms and present different challenges for scalability.

Data Structures

Data structures can have a big impact on the complexity and scalability of a given algorithm. For the case studies in these course notes there are three data structures of particular importance: particles, grids and sparse grids.

Particles are one of the most general ways to represent spatial information. In general particles will have common physical properties such as position and velocity, but they are often used to carry a large number of additional properties around for book keeping. Houdini 12 provides a very flexible API for managing general point attributes organized into a paged data structure for each attribute type. It is a great implementation to study as a balance of flexibility and performance.

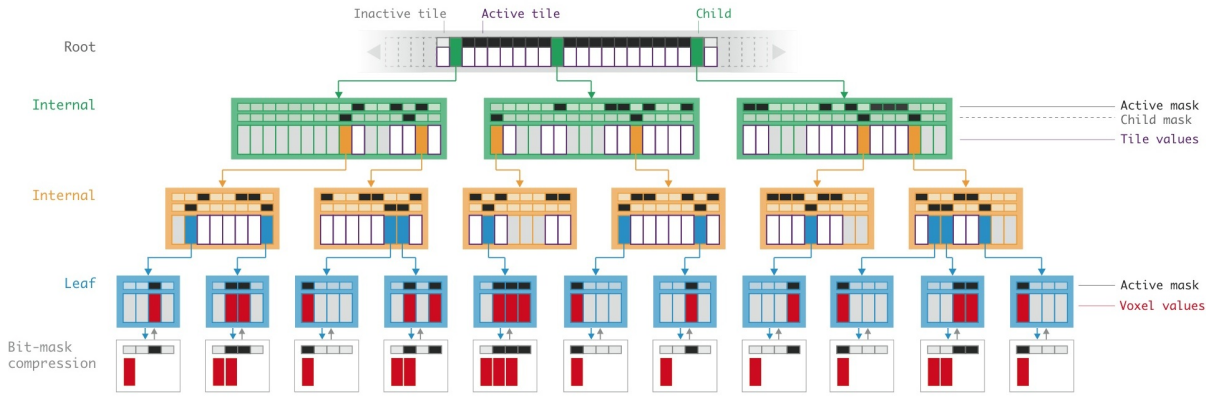
Volumes represent an organization of data into a contiguous N = (N_x * N_y * N_z) segment of memory. Data is stored in volume elements or *voxels*. Data can be read and written to the volume at an arbitrary (i,j,k)-coordinate as long as the indices are in bounds. Volumes can be implemented using simple data structures in C++ or using common libraries such as boost::multiarray. Normally a volume will have a spatial transform associated with it to place the voxels into world space, allowing the world space position of any grid point to be computed without explicitly storing any position data. If memory is allocated contiguously then operations on a volume can take place in an (i,j,k)-index space or by treating the voxel data like a single array of length N. Volumes support a simple form of data decomposition by partitioning work along any dimension (i, j or k) or the entire index range N.



Schematic of a data structure for storing volumetric simulation data. In a typical allocation one or more yz-planes representing contiguous data would be assigned to a thread for processing.

Volumes can cause problems with data locality since points close in index space may be far apart in physical memory, but this is often still the most convenient memory layout for simple problems.

Sparse Grids are a hybrid of particles and volumes. A sparse grid is a data structure that can store information at an arbitrary (i,j,k)-coordinate but only consumes memory proportional the the number of stored values. In order to achieve this a sparse grid might be organized into tiles that are allocated on demand, typically tracked with some other data structure (octree, B-tree or hash table). In these course notes we will use OpenVDB [Museth 2013] as a reference sparse grid data structure.



OpenVDB data structure [Museth 2013].

OpenVDB uses a B+tree to encode the structure of the grid. This is a shallow binary tree with a large fan-out factor between levels. Voxel data is stored in **leaf nodes** organized into 8^3 tiles with a bitmask indicating which voxels have been set. **Internal nodes** store bitmasks to indicate whether any of their children contain filled values. The **root node** can

grow as large as memory requires in order to accommodate an effectively infinite index space. Data in an OpenVDB grid is spatially coherent, meaning that neighbors in index space are generally stored close together in physical memory simply because of the organization into tiles. See [Museth 2013] for more details of the advantages of this data structure.

Sparse grids are important for achieving the memory savings required for high-resolution simulations and volumetric models. However, they present some interesting challenges and opportunities for parallel computing. Writes to an OpenVDB grid are inherently **not thread safe** since each write requires (potentially) allocating tiles and updating bitmasks for tracking. In practice we can implement many parallel operations either by pre-allocating result grids on a single thread or by writing results into a separate grid per thread and then merging the final results using a parallel reduction. This is more efficient than it sounds in practice because often entire branches of the B+tree can be moved from one grid to another during a reduction with simple pointer copies (but not data copies). If branches of the tree do collide then a traversal is triggered that ultimately may result in merging individual leaf nodes. Reads from an OpenVDB grid are always thread safe and parallel computations can be scheduled by iterating over active voxels or active tiles. The latter is generally most efficient since each tile represents up to 512 voxels. High-performance implementations of these operations are supported in the OpenVDB library along with many examples of basic kernels for simulation and volumetric processing.

Fluid Simulation

Fluid simulation is an important category of effect, representing as much as 25% - 35% of the shot work in our feature films from 2008 - 2013. Dense grids are used to store simulation variables such as velocity, forces and active scalars representing smoke, temperature, fuel and so forth.

The following is the basic time integration algorithm for our production fluid solver [Henderson 2012]:

1. advect velocity and integrate body forces
2. project velocity
    a. composite collision volumes into current velocity
    b. update divergence
    c. solve discrete Poisson equation for pressure
    d. update velocity
3. diffuse velocity

4. transport advected scalars
    a. composite any source terms
    b. solve scalar transport

Computational time in steps 1 and 4 is dominated by advection. We provide support for first order semi-Lagrangian [Stam 1999], semi-Lagrangian with 2-stage and 3-stage Runge-Kutta schemes for path tracing, a modified MacCormack scheme with a local min-max limiter [Selle et al. 2008], and Back and Forth Error Compensation and Correction (BFECC) [Dupont and Liu 2003, Kim et al. 2005].

All of these methods have linear complexity in the number of grid points, but may differ in overall cost by a factor of 2-5 depending on the number of interpolations required. They are easily parallelizable and show linear speedup with increasing numbers of workers for sufficiently high resolution. Methods like BFECC are attractive for fluid simulation because they are unconditionally stable and simple to implement.

Here is a straightforward implementation of BFECC using OpenMP. We start with a method that implements first-order semi-Lagrangian advection:

```
void advectSL(const float dt,
              const VectorVolume& v,
              const ScalarVolume& phiN,
              ScalarGrid& phiN1)
{
    const Vec3i res = v.resolution();

    // phi_n+1 = L phi_n

#pragma omp parallel for
    for (size_t x = 0; x < res.x(); ++x) {
        for (size_t y = 0; y < res.y(); ++y) {
            for (size_t z = 0; z < res.z(); ++z) {
                const Vec3i coord(x, y, z);
                const Vec3s pos = Vec3s(coord) - dt * v.getValue(coord);
                phiN1.setValue(coord, sample(phiN, pos);
            }
        }
    }
}
```

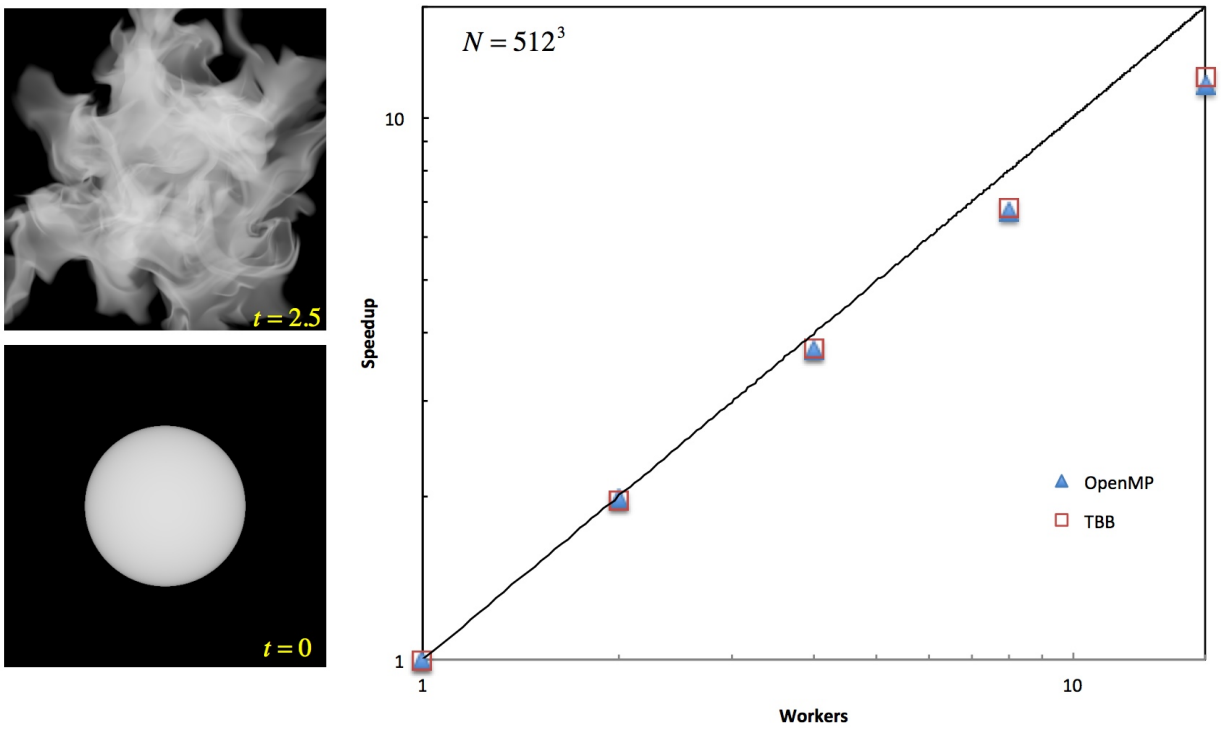
We interpolate values from one grid and write them to another. Clearly there is no contention on writes and we can thread over the grid dimension that varies slowest in memory. BFECC is then built from three applications of this function:

```
void advectBFECC(const float dt,
                 const VectorVolume& v,
                 const ScalarVolume& phiN,
                 ScalarVolume& phiN1,
                 ScalarVolume& work1,
                 ScalarVolume& work2)
{
    // notation

    ScalarVolume& phiHatN1 = work1;
    ScalarVolume& phiHatN  = work2;
    ScalarVolume& phiBarN  = work2;

    // phi^_n+1 = L phi_n

    advect(dt, v, phiN, phiHatN1);

    // phi^_n = L^R phi^_n+1

    advect(-dt, v, phiHatN1, phiHatN);

    // phiBar_n = (3 phi_n - phi^n ) / 2

#pragma omp parallel for
    for (size_t i = 0; i <  phiN.size(); ++i) {
        phiBarN.setValue(i, 1.5f*phiN.getValue(i) - 0.5*phiHatN.getValue(i)));
    }

    // phi_n+1 = L phiBar_n

    advect(dt, v, phiBarN, phiN1);

    // Apply limiter

    limit(v, phiN, phiN1);
}
```

The function `limiter()` applies some rule to deal with any newly created extrema in the advected scalar in order to guarantee that all new values are bounded by values at the previous time step. The middle step can treat the grids like flat arrays since the operation is independent of position.

In the following figure we show speedup measurements for a 3D advection benchmark problem. We advect an initial scalar field in the shape of a smoothed sphere through a velocity field computed from curl noise [Bridson et al. 2007]. Most of the work is in the

`sample()` method, and here we just use trilinear interpolation. This equivalent implementation in TBB is straightforward, and in this figure we compare the speedup behavior of the two programming models to confirm that for large problems they have essentially identical performance.



Speedup curve for scalar advection on a grid with N=512^3 grid points using a BFECC advection kernel. Performance measured on a desktop system with dual Intel Xeon Processors E5-2687W (20M Cache, 3.10 GHz) using up to 16 computational threads.

Comparing Parallel Methods for the Discrete Poisson Equation

Steps 2 and 3 in the above time integration algorithm require solving the discrete Poisson equation for the pressure, or the discrete Helmholtz equation in the case of diffusion. Solving this equation on a grid amounts to solving a large, sparse linear system, and selecting the method to use for this step has a critical impact on performance of the overall framework.

The optimal method depends on the hardware that will be used for simulation. Our target hardware platform for this system is a shared-memory multiprocessor. Current hardware trends indicate that the number of processing units available on such systems will increase at a steady rate over the next several years, and therefore we want to select solution techniques that scale well on such systems.

The solution techniques most commonly invoked in the computer graphics literature are the conjugate gradient method (CG) and multigrid (MG). Both are iterative techniques that require no explicit representation of the matrix. For constant-coefficient problems like the ones required here, we can also consider techniques based on the Fast Fourier Transform (FFT). This is a direct method that takes advantage of the fact that the Helmholtz equation can be solved independently for each Fourier mode. A common misconception about FFT-based methods is that they are restricted to problems with periodic boundary conditions, but by using appropriate sine- or cosine- expansions they can be used to solve problems with general boundary conditions. A good overview of all three methods is available in the scientific computing literature [Press et al. 1996], with additional details on MG available in [Briggs et al. 2000].

Estimates of computational complexity for shared memory multiprocessors are typically made using an idealized computing platform called a Parallel Random Access Machine (PRAM). This is a model of an abstract shared memory machine that assumes an arbitrary number of processing units and ignores the cost of synchronization or communication. The following table shows the theoretical running time on a serial machine and a PRAM for these methods, reproduced partially from [Demmel 1996]:

Method	**Serial Time**	**PRAM Time**
SOR	N^(3/2)	N^(1/2)
CG	N^(3/2)	N^(1/2) log N
Multigrid	N	(log N)^2
FFT	N log N	log N
Lower Bound	N	log N

Asymptotic complexity for several common methods used to solve the discrete Poisson problem [Demmel 1996].

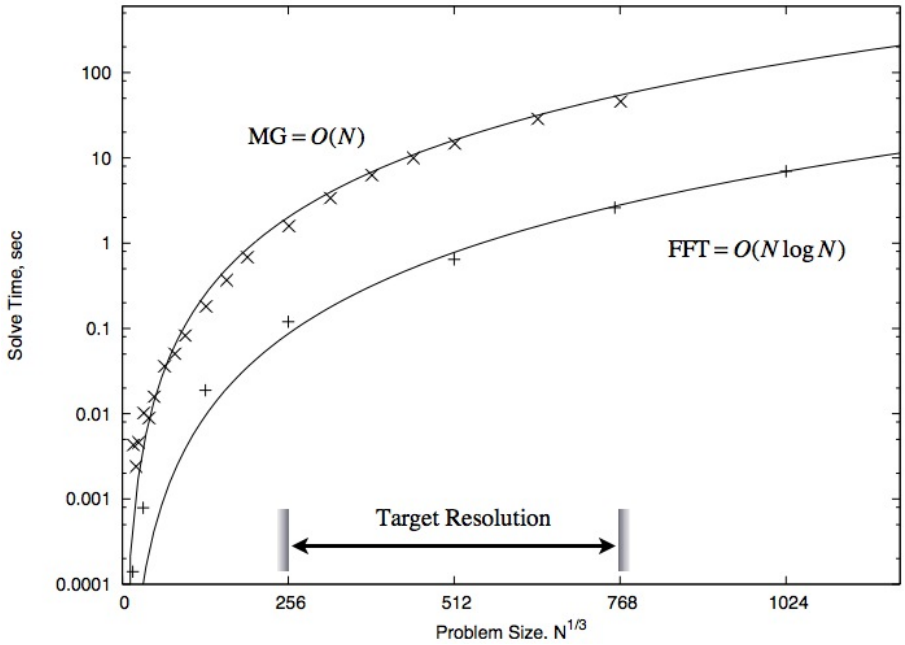
No serial method can be faster that O(N) since it must visit each grid point at least once. MG is the only serial method that achieves this scaling. On an idealized machine with P = N processors scaling depends on the amount of parallelism that can be exploited in the algorithm. The lower bound is O(log N) because this is the minimal time to propagate information across the domain via parallel reduction. The FFT-based solver is the only

method that achieves this scaling. CG is slower than the best method by a factor of N^(1/2) in both cases and will not be considered further. The key observation is this: for large problems the optimal serial algorithm is MG, while the optimal parallel algorithm is an FFT-based solver.

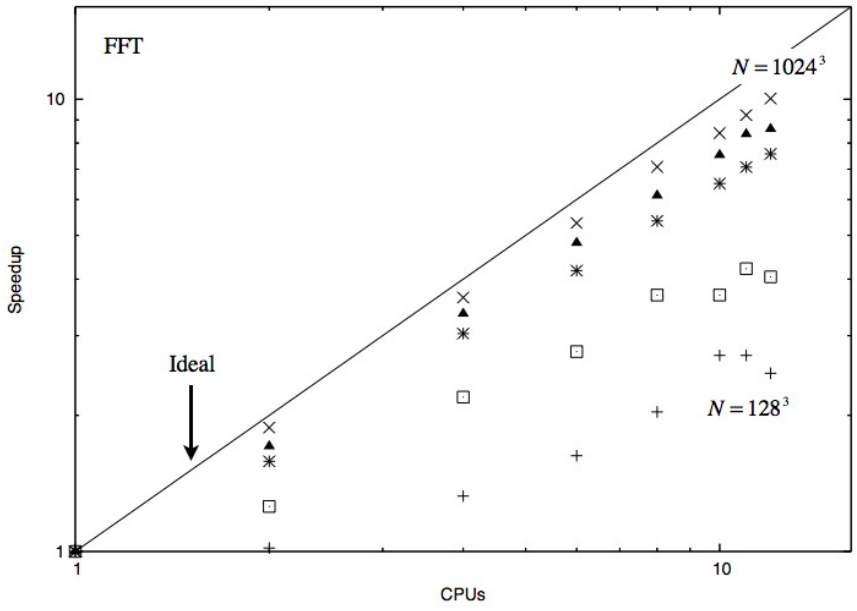
However, these estimates are only for the overall scaling. Actual performance will depend on the details of the hardware platform. As a benchmark problem we compute the solution to the Poisson problem with zero Dirichlet boundary conditions on the unit cube. Times are measured on a workstation with dual Intel Processors X5670 (12M cache, 2.93 GHz) which support up to 12 hardware threads.

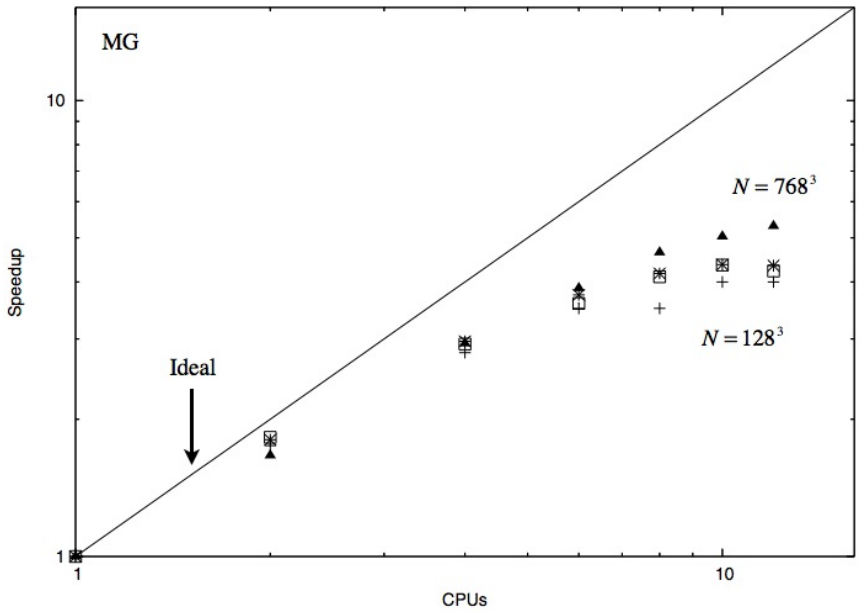
The MG solver is compiled using the Intel 12 vectorizing Fortran compiler with optimization level 3. The MG solver uses a W(2,1) cycle with a vectorized Gauss-Sidel iteration for residual relaxation at each grid level [Adams 1999]. MG iterations are limited to five complete cycles which is enough to reduce the relative change between fine grid iterations to less than 1×10^{-5} in all cases reported. Fine grid resolution is restricted to powers-of-two multiples of a fixed coarse grid resolution. FFT-based solution timings use the Helmholtz solver interface in the Intel Math Kernel Library [Intel 2011]. This interface provides a “black-box” solver for the Helmholtz equation on grids with arbitrary resolutions using prime factorization, but is most efficient for powers-of-two grid points. This is a direct technique that produces the solution to machine precision in a fixed number of operations independent of the right-hand-side and therefore has no iteration parameters. Both implementations use OpenMP for shared memory parallelism [OpenMP 2011].

In the following figure we show the measured solve times for the above benchmark case for problem sizes from N = 16^3 to N = 1024^3, which covers the problem sizes of practical interest for visual effects production. Both methods scale as expected with problem size, but the FFT-based solver is almost an order of magnitude faster for a given number of grid points. In fact, the FFT-based solver produces a solution to machine precision in less time than a single MG cycle. The next figure shows the parallel speedup for both solvers, and as expected the FFT-based solver also has much better parallel efficiency. Note that the MG solve times and parallel efficiencies reported here are consistent with the recent work of [McAdams et al. 2010] for similar problem sizes and hardware.



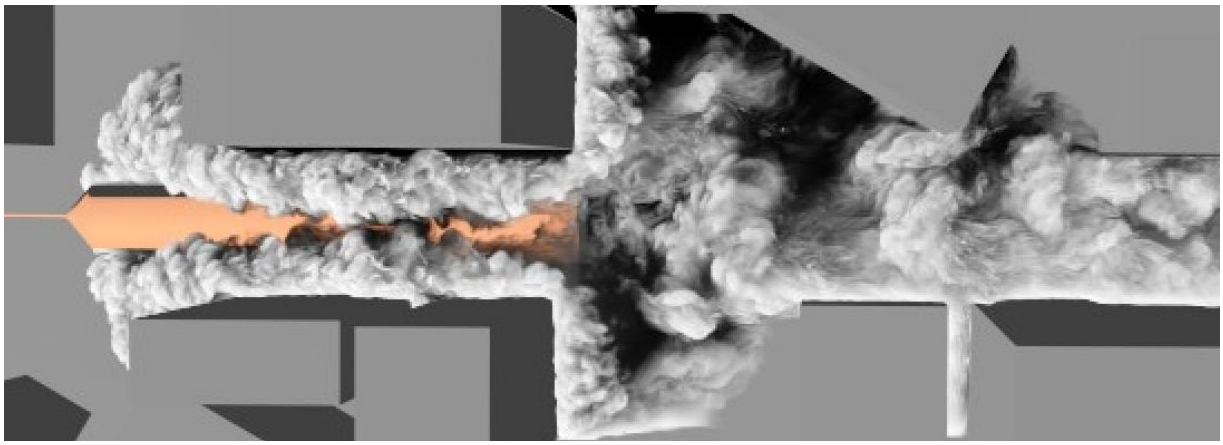
Comparison of solve times for the Poisson equation with N total grid points using MG and FFT solution techniques. The solid curves confirm the expected asymptotic scaling. Run times are measured on a workstation with dual Intel Xeon Processors X5670 using 12 computational threads.





Parallel speedup for the FFT-based Poisson solver (top) and MG Poisson solver (bottom) for various grid resolutions.

These measurements confirm that even for a relatively small number of processing cores there is a significant performance advantage to using the FFT-based solver over MG. Although the MG solver can be applied to a much more general class of elliptic problems, it is an order of magnitude more expensive. They also confirm that the FFT-based technique has better parallel scaling and efficiency, as expected. Since all of the elliptic solves required in this framework are by design suitable for the use of FFT-based techniques, this is the method used exclusively. This includes the solution to the pressure Poisson problem, and the implicit diffusion step for both fluid velocity and scalar transport. This is the single most important choice affecting performance of the fluid integration algorithm.



Volume rendering of a large dust cloud simulated with a resolution of N=1200x195x500 along with a final frame from the movie Megamind [Vroeijenstijn and Henderson 2011].

Liquid Simulation

Liquid simulation is closely related to the techniques that we use for volumetric fluids, but has the additional complication of free surface boundary conditions and the need to explicitly track a surface. Multiple approaches are possible, but the most popular is to use to a hybrid particle and grid based method. Particles are used to represent the liquid, and a sparse grid is used as an intermediate data structure to perform operations such as pressure projection that run more efficiently on the grid. Sparse grids are important because the grid data representing a liquid typically covers a highly irregular region of space.

Here is a basic time integration method for liquids based on the Fluid-Implicit-Particle (FLIP) method [Brackbill and Ruppel 1986; Bridson 2008]:

1. transfer particle velocity information to grid
2. integrate body forces and accelerations on grid
3. project velocity on grid
    a. set free surface boundary conditions
    b. solve discrete Poisson equation for pressure
    c. update velocity
4. add interpolated grid velocity difference to particle velocity
5. integrate particle motion to get final position

Steps 1, 3, 5 and 6 in the above integration scheme are easily parallelizable as vector operations over independent particles or grid points. Scalability may be limited by low arithmetic complexity.

Step 4 requires solving a discrete Poisson equation of the same mathematical form as the example above, but in this case with a highly irregular domain corresponding to the current shape of the liquid based on the location of particles. In our performance testing this step only represents about 15% of the overall time to integrate the equations of motion, so we solve it using an Incomplete Cholesky preconditioner combined with conjugate gradient iteration (ICCG) (see e.g. [Bridson 2008]). This has limited parallelism but the method is good enough for moderate grid resolution. Other techniques are possible based on parallel ICCG or MG but we leave these for future discussion.

Example of a liquid effect in the form of a splash element with some surrounding open water [Budsberg et al. 2013].

Parallel Point Rasterization

Step 2 in the above algorithm turns out to be the performance bottleneck in our fluid solver, representing about 40% of the overall simulation time. In this step we rasterize particle velocities into a sparse grid in preparation for solving for the pressure and correcting for divergence. We refer to this problem as *point rasterization* or *point splatting*. The general problem is to compute the interpolation of some field value u_p defined at irregular particle locations x_p onto the regular grid points u_m:

$$u_m = \sum_{p=1}^{N} W(x_m - x_p) u_p \,,$$

where W is a smoothing kernel. In practice W will be compact so each particle will only influence a small set of neighboring grid points.

We can implement this method by iterating over grid points, finding all particles that overlap with that grid point, and then summing their contributions according to the above equation. This requires some acceleration structure to quickly find all particles in the neighborhood of a given grid point.

But we can also invert this operation and iterate over the particles while summing their contributions into each grid point. This removes the need for a particle acceleration structure. A serial implementation of this method is straightforward:

```
void paToGrid(const PaList& pa, VectorGrid& v)
{
    // rasterize particle velocities into grid v

    for (size_t i = 0; i < pa.size(); ++i) {
        const Vec3s pos  = pa.getPos(i);
        const Vec3s vel  = pa.getVel(i);
        scatter(v, pos, vel);
    }
}
```

The function `scatter()` is effectively our implementation of the summary kernel. We'll consider a low-order B-spline (BSP2) and the M_4' function (MP4) as smoothing kernels [MONAGHAN 1985]. Note that BSP2 affects 2x2x2=8 mesh points around each particle, and MP4 affects 4x4x4 = 64 mesh points around each particle, so these are interesting to compare because of the differences in arithmetic complexity.

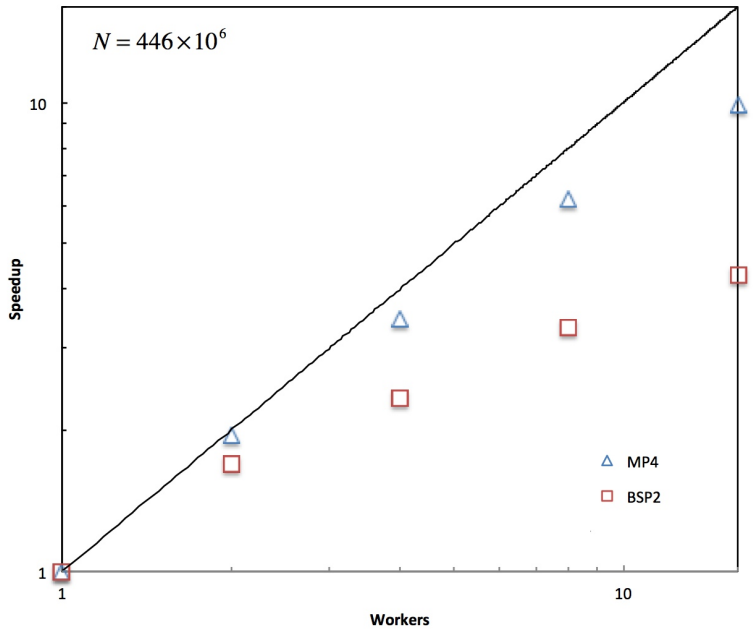
The problem with parallelizing the above loop is that successive calls to scatter may write to the same grid points, so we need some way to coordinate the writes to avoid contention.

Here we have an elegant solution with TBB and OpenVDB. The basic strategy is to create a separate sparse grid for each thread and rasterize a subrange of particle velocities into that grid. As mentioned above, OpenVDB supports fast operations to combine grids that take advantage of its tree structure. We can express the entire operation using a TBB reduction. Here is the parallel implementation of the serial rasterization algorithm:

```
class PaToGrid
```

```
{
    const PaList& mPa;
    VectorGrid& mV;

public:
    PaToGrid(const PaList& pa, VectorGrid& v)
        : mPa(pa), mV(v) {}
    PaToGrid(const PaToGrid& other, tbb::split)
        : mPa(other.mPa), mV(other.mV.copy()) {}

    // rasterize particle velocities into grid v

    void operator()(const tbb::blocked_range<size_t>& range)
    {
      for (size_t i = range.begin(); i < range.end(); ++i) {
            const Vec3s pos = mPa.getPos(i);
            const Vec3s vel = mPa.getVel(i);
            scatter(mV, pos, vel);
      }
    }

    // merge grids from separate threads

    void join(const PaToGrid& other) {
        openvdb::tools::compSum(mV, other.mV);
    }
};

// Execute particle rasterization
PaToGrid op(pa, v);
tbb::parallel_reduce(tbb::blocked_range<size_t>(0, pa.size()), op);
```

This allows the operation to execute without the need for any locks, and provides overall good performance. The following figure shows speedups for a benchmark problem with N=446 million points rasterized into a grid with a final resolution of more than 800^3. Note that the scaling is significantly better for the MP4 kernel. This kernel is more expensive, but it has a higher ratio of compute to communication and therefore much better scaling. The BSP2 kernel is limited by memory bandwidth and achieves only a speedup of S ~ 5 with P=16 threads.



Speedup curve for velocity rasterization from N=446 million points into a N=836^3 (effective resolution) sparse grid using a high-order (MP4) and low-order (BSP2) kernel. Run times are measured on a workstation with dual Intel Xeon Processors E5-2687W (20M Cache, 3.10 GHz) using up to 16 computational threads.

SUMMARY

We have reviewed some of the history and current hardware choices driving algorithm and tool development at DreamWorks Animation. We presented case studies of kernels for fluid simulation and liquids to illustrate the relationship between hardware, algorithms and data structures.

MORE INFORMATION

Several of the technologies discussed in these course notes are available online. For more information see the following web sites:

OpenMP web site:
http://openmp.org/

Threading Building Blocks (TBB) web site:
http://threadingbuildingblocks.org

OpenVDB web site:
http://www.openvdb.org

REFERENCES

ADAMS, J. C. 1999. MUDPACK: Multigrid software for elliptic partial differential equations. NCAR. Version 5.0.1.

BRIGGS, W. L., HENSON, V. E., AND MCCORMICK, S. F. 2000. A multigrid tutorial, 2nd ed. ed. Society for Industrial and Applied Mathematics, Philadelphia, PA.

BRIDSON, R., HOURIHAM, J., AND NORDENSTAM, M. 2007. Curl-noise for procedural fluid flow. ACM Trans. Graph. 26.

BRIDSON, R. 2008. Fluid simulation for computer graphics. A K Peters, Wellesley, Mass.

BUDSBERG, J., LOSURE, M., MUSETH, K. and BAER, M. 2013. Liquids in *The Croods*. ACM Digital Production Symposium.

DEMMEL, J. 1996. Applications of parallel computers. Retrieved from U.C. Berkeley CS267 Web site: http://www. cs.berkeley.edu/˜ demmel/cs267/.

DUPONT, T. F., and LIU, Y. 2003. Back and forth error compensation and correction methods for removing errors induced by uneven gradients of the level set function. Journal of Computational Physics, vol. 190, no. 1, pp. 311–324.

HENDERSON, R. 2012. Scalable fluid simulation in linear time on shared memory multiprocessors. ACM Digital Production Symposium (DigiPro).

INTEL. 2011. Intel Math Kernel Library Reference Manual. Intel Corporation. Document number 630813-041US.

KIM, B., LIU, Y., LLAMAS, I., and ROSSIGNAC, J. 2005. Flowfixer: Using BFECC for fluid simulation. Eurographics Workshop on Natural Phenomena.

MCCOOL, M., ROBISON, A., and REINDERS, J. 2012. Structured Parallel Programming. Morgan Kaufmann.

MCADAMS, A., SIFAKIS, E., AND TERAN, J. 2010. A parallel multigrid Poisson solver for fluids simulation on large grids. In Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, SCA ’10, 65–74.

MUSETH, K. 2013. VDB: High-resolution sparse volumes with dynamic topology. ACM Trans. Graph. 32, 3.

OPENMP. 2011. The OpenMP API specification for parallel programming. OpenMP. Version 3.1.

PRESS, W., TEUKOLSKY, S., VETTERLING, W., AND FLANNERY, B. 1996. Numerical Recipes in C: The Art of Scientific Computing, second ed. Cambridge University Press.

STAM, J. 1999. Stable fluids. In Proceedings of the 26th annual conference on Computer graphics and interactive techniques, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH ’99, 121–128.

VROEIJENSTIJN, K., AND HENDERSON, R. D. 2011. Simulating massive dust in Megamind. In ACM SIGGRAPH 2011 Talks, ACM, New York, NY, USA, SIGGRAPH ’11.

MONAGHAN, J. J. 1985. Extrapolating B splines for interpolation, J. Comput. Phys. 60 (2) 253–262.