SPGrid: A Sparse Paged Grid structure applied to adaptive smoke simulation

Rajsekhar Setaluri Mridul Aanjaneya Sean Bauer Eftychios Sifakis
University of Wisconsin - Madison

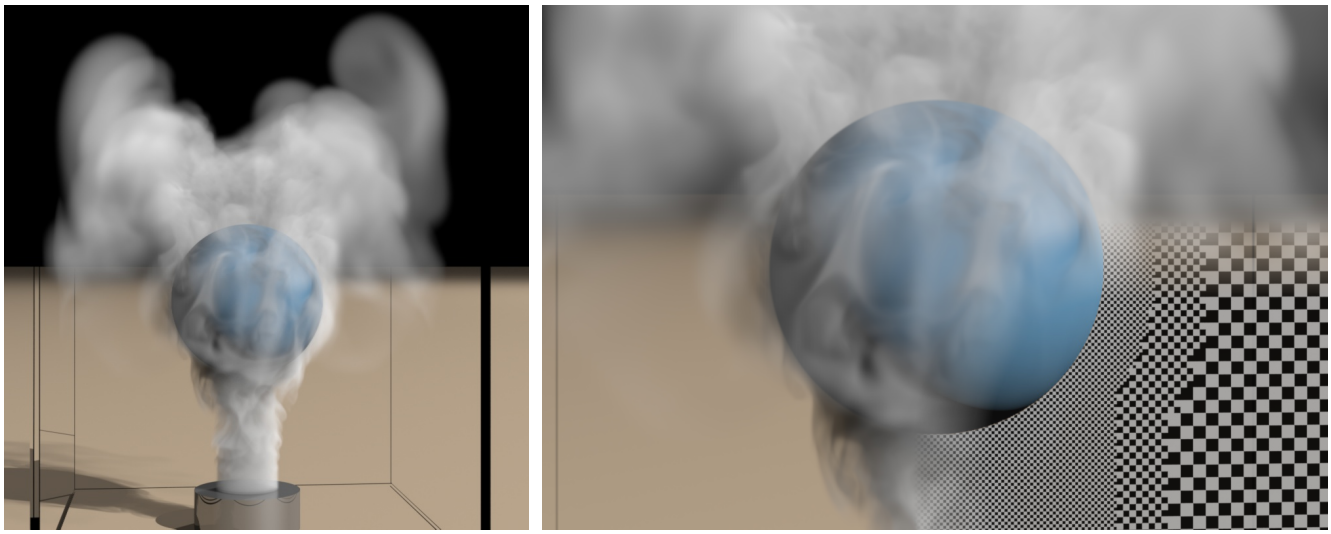


Figure 1: *Smoke flow past sphere with* $135M$ *active voxels,* $1K \times 1K \times 2K$ *maximum resolution. Adaptive grid shown on the right.*

Abstract

We introduce a new method for fluid simulation on high-resolution adaptive grids which rivals the throughput and parallelism potential of methods based on uniform grids. Our enabling contribution is *SPGrid*, a new data structure for compact storage and efficient stream processing of sparsely populated uniform Cartesian grids. SPGrid leverages the extensive hardware acceleration mechanisms inherent in the `x86` Virtual Memory Management system to deliver sequential and stencil access bandwidth comparable to dense uniform grids. Second, we eschew tree-based adaptive data structures in favor of storing simulation variables in a pyramid of sparsely populated uniform grids, thus avoiding the cost of indirect memory access associated with pointer-based representations. We show how the costliest algorithmic kernels of fluid simulation can be implemented as a composition of two kernel types: (a) stencil operations on a single sparse uniform grid, and (b) structured data transfers between *adjacent* levels of resolution, even when modeling non-graded octrees. Finally, we demonstrate an adaptive multigrid-preconditioned Conjugate Gradient solver that achieves resolution-independent convergence rates while admitting a lightweight implementation with a modest memory footprint. Our method is complemented by a new interpolation scheme that reduces dissipative effects and simplifies dynamic grid adaptation. We demonstrate the efficacy of our method in end-to-end simulations of smoke flow.

CR Categories: I.3.7 [Computer Graphics]: Computational Geometry and Object Modeling—Physically based modeling;

Keywords: fluid simulation, sparse grids, adaptive discretizations

Links: DL PDF WEB

1 Introduction

Computer graphics research has explored opportunities for spatial adaptation in fluid simulation using a broad spectrum of techniques, including adaptive Cartesian grids [Losasso et al. 2004; Zhu et al. 2013; Ferstl et al. 2014], adaptive tetrahedral meshes [Klingner et al. 2006; Ando et al. 2013] and particle based approaches [Adams et al. 2007; Solenthaler and Gross 2011]. For the same number of degrees of freedom, the visual richness of adaptive techniques is clearly superior to uniform grids. However, the computational cost is markedly higher. Common operations such as stencil computations and sparse linear algebra are significantly more efficient on uniform grids, as they can leverage cache-optimized memory access patterns, maximize prefetching efficiency and simplify balanced domain partitioning for parallelism.

This has prompted many authors to pursue balanced compromises between highly adaptive representations and purely uniform grids, to achieve the best of both worlds in terms of both detail and performance. RLE-based techniques [Houston et al. 2006; Irving et al. 2006; Chentanez and Müller 2011] propose a hybrid between a 2D uniform grid and a 1D run-length encoding scheme. Adaptive mesh refinement (AMR) and chimera grid techniques [Berger and Colella 1989; Sussman et al. 1999; Patel et al. 2005; Dobashi et al. 2008; Tan et al. 2008; Cohen et al. 2010; Golas et al. 2012; English et al. 2013] achieve spatial adaptation by patching together overlapping uniform grids of different resolutions. It has also been proposed that dimension-by-dimension adaptation can focus resolution on areas of interest, while maintaining a topologically uniform underlying grid [Zhu et al. 2013]. Finally, one of the most popular and

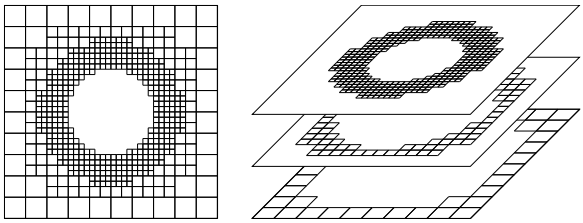


Figure 2: *Quadtree modeled as a pyramid of sparse uniform grids.*

performant recent techniques for storing adaptive volumetric data, OpenVDB [Museth 2013], builds on the performance potential of uniform grids by using a shallow tree with a high branching factor, storing a uniform grid in every node.

We introduce *SPGrid* (short for Sparse Paged Grid), a novel data structure for the storage and stream processing of sparsely populated uniform grids. SPGrid replicates both the throughput potential and programming experience of using a uniform grid, while having a storage footprint that only exceeds the actual sparse grid content by a very modest margin. We use this data structure to construct an adaptive discretization for fluid simulation where the conventional tree- and pointer-based representation has been replaced (see Figure 2) by a pyramid of sparsely populated uniform grids. We show that all costly algorithmic kernels encountered in an adaptive CFD simulation can be implemented in our scheme without explicit tree traversal, by combining efficient streaming kernels operating on uniform grids. Finally, we present an adaptive multigrid preconditioned Krylov solver for the pressure Poisson equation which has a very lightweight footprint, and outperforms alternatives based on incomplete factorizations in terms of convergence and scalability.

Contributions Our core contributions are summarized below:

- A novel lightweight data structure ("SPGrid") for large, sparsely populated Cartesian grids, which leverages established *hardware* acceleration mechanisms of the virtual memory subsystem to achieve competitive performance to dense uniform grids, and supports coordinated storage of multiple data channels with non-identical (albeit similar) domains of support. SPGrid uses Morton encoding to map 3D data sets into a linear memory span; however we show how stencil operations can be performed natively with linearized (encoded) addresses without translating them to geometric coordinates.
- A reformulation of conventional pointer-based octree-style adaptive fluid discretizations using an SPGrid pyramid, where the expensive algorithmic kernels (e.g. advection, discrete gradient, divergence and Laplacian) are composed from simple, parallelizable uniform grid kernels. This property holds even for non-graded octrees, i.e. cases where neighboring cells may differ by more than one level of resolution.
- An adaptive multigrid-preconditioned Krylov solver that achieves resolution-independent convergence rates and can be natively implemented, in a matrix-free fashion, in the same SPGrid pyramid used for all other fluid simulation kernels.
- An interpolation scheme that reduces dissipation effects in advection, and simplifies dynamic adaptation of grid topology.

Although these contributions are quite complementary and align nicely in our final product, they are also modular and reusable in other contexts. For example, a narrow-band level set application could leverage SPGrid alone and forego the adaptive embellishments. Similarly, one could replace SPGrid with OpenVDB (e.g. in scenarios that demand out-of-core processing) to store sparsely populated grids, and retain the pyramid adaptivity concept. In our supplemental material we also explain how the SPGrid structure can even be used for optimized storage and application of an Incomplete Cholesky preconditioner, if desired, instead of multigrid.

2 Related work

Cartesian grids are very popular for fluid simulation, typically using a MAC grid discretization [Harlow and Welch 1965], the unconditionally stable semi-Lagrangian advection scheme [Stam 1999] and a pressure Poisson projection step for incompressibility, commonly solved using Preconditioned Conjugate Gradients [Fedkiw et al. 2001]. These fundamental concepts have also been employed in octree discretizations [Losasso et al. 2004], and recently in more efficient RLE representations [Houston et al. 2006; Irving et al. 2006; Chentanez and Müller 2011]. Grid-based discretizations are hardly the only option, and a number of alternatives are based on tetrahedral meshes [Feldman et al. 2005; Klingner et al. 2006; Chentanez et al. 2007; Batty et al. 2010; Ando et al. 2013] which may also provide the flexibility for spatial adaptation. Other authors have also proposed solvers using hybrid Particle-Eulerian methods [Zhu and Bridson 2005] as well as purely SPH-based methods [Desbrun and Cani 1996; Müller et al. 2003; Premoze et al. 2003; Adams et al. 2007], which can also support spatial adaptivity [Solenthaler and Gross 2011]. However, these methods may exhibit suboptimal cache performance as the data layout in memory is non-sequential. To leverage cache-optimized memory access patterns, Morton indexing has been used for various CFD applications [Aftosmis et al. 2004; Goswami et al. 2010]. Methods based on grids conforming to the liquid surface [Clausen et al. 2013], Voronoi diagrams [Sin et al. 2009; Brochu et al. 2010], and velocity-vorticity domain decomposition [Golas et al. 2012] have also been proposed. Finally, others have investigated guiding high resolution simulation from a coarser resolution version [Nielsen and Bridson 2011], or adapting the per-axis spacing of uniform grids [Zhu et al. 2013].

Independent of CFD applications, a rich literature exists in using adaptive and compressed representations for volumetric data such as distance fields [Frisken et al. 2000], voxelized geometry [Crassin et al. 2009], and exceptionally refined level sets [Nielsen et al. 2007]. Various compressed storage structures have been proposed, including hash-based approaches [Teschner et al. 2003; Brun et al. 2012] and Run-length encodings [Houston et al. 2006]. Finally, the VDB data structure [Museth 2013], which evolved from Dynamic Tubular Grids [Nielsen and Museth 2006] and the DB+Grid data structure [Museth 2011] enjoys broad production use and is publicly available via the open source OpenVDB software library.

Relation to OpenVDB Our proposed data structure, SPGrid, has a significant conceptual affinity to the OpenVDB library [Museth 2013], which merits special mention. In fact, the availability of an open-source implementation was extremely beneficial for our work, as we had a mature, performance-conscious solution to benchmark against. Both SPGrid and VDB seek to accelerate sequential and stencil access. SPGrid places even higher emphasis on optimizing throughput, especially in a multithreaded setting, to levels directly comparable with dense uniform grids. In this vein, SPGrid relies on hardware mechanisms for address translation and caching; for example, VDB uses a software cache to store address translations of tree nodes, while SPGrid effectively uses the Translation Lookaside Buffer (TLB) for this purpose. The emphasis on maximizing throughput necessitates certain compromises in scope and functionality. For example, SPGrid only stores sparse subsets of a *single* uniform grid, while VDB can also store values at coarser tree nodes. SPGrid relies on an additional software layer (see section 4) to accommodate multiresolution storage. VDB is natively designed to accommodate out-of-core processing, while SPGrid is exclusively targetting cases where all data can remain resident in physical memory. The effective resolution of VDB grids is only limited by available space, while SPGrid has finite (albeit, gigantic) maximum grid size. Table 1 compares key features of OpenVDB and SPGrid.

	SPGrid	OpenVDB
Random Access	Very fast (Haswell) Moderate (Ivy Bridge)	Fast
Sequential Access	Very fast ($\approx$ Uniform)	Fast
Stencil Access	Very fast	Fast
Max. Grid Size	16K$\times$32K$\times$32K (1 channel) 8K$\times$16K$\times$16K (16 channels)	Unlimited
Out-of-core	Not supported	Supported
OS	Linux tested (currently)	Platform Independent
Caching	Hardware	Software
Multi-resolution	Single grid only	Yes

Table 1: *Comparison of SPGrid and OpenVDB.*

3 A Sparse Paged Grid structure (SPGrid)

Different application areas may adopt one of many sparse and adaptive storage schemes (octrees, hashed grids, RLE, etc) based on their particular needs. CFD workloads, in particular, carry specific design demands: the most fundamental observation is that **sequential and stencil access** are by far the most common use scenarios; even "random" access typically exhibits spatial coherence (e.g. semi-Lagrangian advection). Both iterative solvers (e.g. Conjugate Gradients) and direct methods or components (e.g. Incomplete Cholesky factorization) are memory-bound, mandating that data storage schemes should optimize for **memory bandwidth and footprint**. This is even more crucial for multiprocessor (or SIMD) platforms that can be vastly underexploited if memory utilization is not kept to a minimum. Additionally, fluid simulations need to operate on **multiple data channels** of scalar and vector-valued quantities (pressure, density, velocities, auxiliaries of Krylov solvers, etc.), many of which are not geometrically coincident with one another, but occupy very similar spatial extents (see Figure 6).

We propose a data structure for sparsely-populated *uniform* grids, as one might use for example to solve a narrow-band level set problem. We note that if memory footprint was not a concern, one could in principle satisfy the aforementioned algorithmic specifications by simply allocating a *dense* uniform grid, and only using its sparse relevant subset. Stencil access is particularly straightforward to optimize in this context, since a specific neighbor of a grid location is always stored a constant memory offset away from the address of the reference data point. Although using dense storage as a means to streamline access to sparse subsets is clearly an impractical proposition (grids in our work may have an occupancy as low as 0.1%) this concept is similar to how Virtual Memory operates. CPUs allow a process access to a vast Virtual Memory address space, provided that the sparse subset that is actually touched (quantized to 4KB pages) is small enough to fit in physical memory. Our proposed data structure uses a locality-preserving mapping from 2D/3D grid locations to a 1D memory span, and uses Virtual Memory mechanisms to avoid allocation of unused regions.

3.1 Array layout

Our proposed SPGrid structure provides an abstraction for a multidimensional (2D/3D) array which only contains data in a sparse subset of its entries. Similar to C++ static arrays, every array coordinate, whether populated or not, has a *specific location* in (virtual) memory where its contents would be stored. This is in contrast to pointer-based octrees, VDB or hashed grids, which will heap-allocate storage on first access, and maintain an explicit translation between geometric array coordinates and the actual memory location of a given entry. Although SPGrid reserves a gigantic virtual memory address span for the sparse grid, only a tiny subset of array entries will ever be touched and occupy space in physical memory.

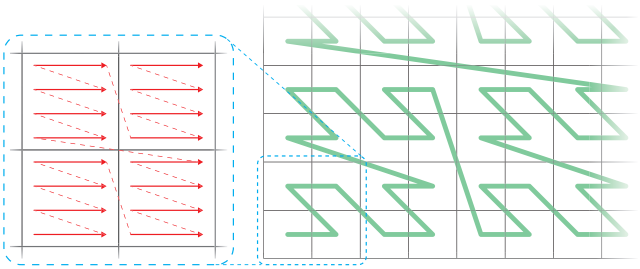


Figure 3: *Illustration of the traversal order by which SPGrid maps 2D/3D array data to a linear memory span. Lexicographical order is used to traverse the interior of small-sized blocks, while the blocks themselves are laid out along a space-filling Morton curve.*

Unlike the lexicographical ordering of C++ arrays, SPGrid uses a custom traversal pattern to map 2D/3D geometric coordinates into linearized memory offsets. We first partition the array into rectangular blocks of a small, carefully chosen size (our fluid simulation examples use $4 \times 4 \times 4$ blocks). Array coordinates are then mapped to 1D memory offsets by traversing the contents of each block lexicographically, and arranging blocks along a space-filling Morton curve as shown in Figure 3. Morton ordering is a locality-promoting map, ensuring that geometrically proximate array entries will, with high probability, be stored in nearby memory locations. As a consequence, if the sparse occupancy pattern of the array exhibits high spatial coherency, the corresponding data entries will be highly clustered in memory. The next step is to reserve a virtual memory address span, for the entire array, but without occupying space in physical memory until its contents need to be accessed.

Modern operating systems provide mechanisms for reserving virtual memory space *without* reserving physical memory beforehand. In Linux, this functionality is embodied in the `mmap` system call, which is used in our implementation as follows:

```
void *ptr=mmap(0,size,PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS|MAP_NORESERVE,-1,0)
```

Invoked as such, `mmap` will promptly reserve even a vast virtual memory span (up to the architectural limit of 128TB per process) and return immediately without reserving any physical memory. The `mmap` arguments `MAP_ANONYMOUS|MAP_NORESERVE` further instruct the operating system to *never* swap out any page in this span, ensuring that any page in this virtual address range is either physical-memory resident, or has never been touched before (an out-of-memory error will be triggered before ever violating this property). After this initial call, any subsequent access to a memory location within the span reserved by `mmap` has one of the following effects: (a) If the virtual page containing the requested address does not have a translation in the virtual page table then a page fault occurs, prompting the operating system to reserve a physical page, record the mapping in the page table, and zero-fill the physical page contents prior to making it available for normal use. (b) If the page of the requested address has already been mapped (i.e. it has been touched before), address translation (and handing of any TLB misses, triggering a page table walk) is handled completely in hardware, without operating system intervention. The latter case is no different to the handling of a memory request for an address that has been allocated with any other mechanism. Note that, in order for our scheme to work, the reservation of the host array memory span must be done with the `mmap` mechanism (or a mechanism like VirtualAlloc under Windows), and not with the `malloc` library function, or the default C++ operator `new`, as those will automatically map the requested range to physical memory. Finally, we assume that the allocated space for the entire array is always aligned to 4KB page boundaries, which will be automatic if `mmap` is used.

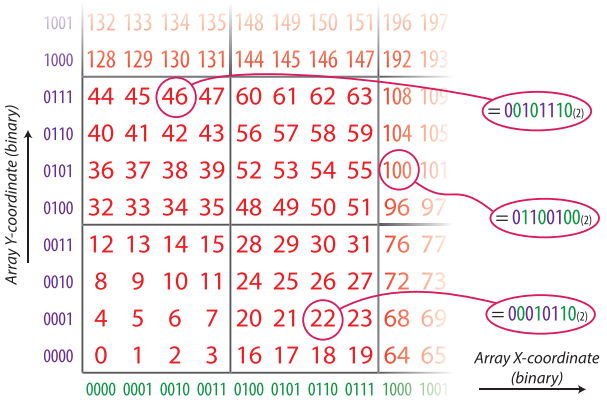


Figure 4: *The 1-dimensional index of every array entry is computed by bit-interleaving the binary expansions of the array coordinates.*

3.2 Address computation and multi-channel arrays

The relation between Cartesian coordinates and the linearized index suggested by our traversal scheme becomes apparent if we examine their binary expansion. As seen in Figure 4 the linearized index of an array entry is computed by interleaving the bits of the multidimensional Cartesian coordinates according to a specific pattern. We also specified that SPGrid should accommodate storage of multiple data channels (corresponding to different simulation variables). Instead of dedicating a separate SPGrid for each channel, we choose to interleave their contents in memory at block granularity. In fact, we choose the geometric block size such that all channels of a block will fit exactly in a 4KB memory page. In the example of Figure 5, 4KB pages are used to store four distinct float-valued data channels (u, v, w, p) allowing 1KB or 256 entries per channel; this suggests a geometric block size of $8 \times 8 \times 4$. Our smoke simulator uses 16 channels, each storing either a single-precision floating point value or a $32-$bit integer, allowing for storage of 64 entries per channel, and resulting in a block size of $4\times4\times4$. Note that it is not necessary that all data channels have the same width; it is perfectly possible to mix channels containing floats and doubles, for example.

For quick access, we precompute a *base pointer* for each data channel, indicating the first memory location where an entry of this channel (in essence, the entry corresponding to zero Cartesian coordinates) would be stored. The *byte offset* between the location of an arbitrary entry (i, j, k) of a channel and the respective base pointer is computed as shown in the lower part of Figure 5. We note that:

- The interleaving pattern is known at compile time since it only depends on the number and size of channels. Since the spread pattern of each coordinate (i, j or k) preserves the ordering of its bits, we encode it by storing (per coordinate) a **64-bit mask** of the bit locations that each coordinate is spread into.
- The zeros inserted into the bit pattern are necessary both for aligning data according to their size, as well as "jumping over" the other channels before moving on to the next block.
- The bit interleaving pattern need not be concerned with the maximum array size. We construct the spread masks to use as many of the coordinate bits as possible when assembling the $64-$bit offset, even if the populated array size is smaller.

The bit interleaving operation is not trivial in terms of cost, and certainly more costly than the offset translation in lexicographically ordered static C++ arrays (which would just require 2 additions and 2 multiplications in 3D). The cost is significantly lower on CPUs of the Haswell architecture however (16 cycles for translation of a 3D coordinate), since they include hardware instructions (`pdep`) for distributing bits according to a mask. On Ivy Bridge processors the cost of a full 3D translation is 85 cycles in our optimized implementation. Note that, in cases of true random access with no spatial locality, this translation cost would actually not be the bottleneck, as cache/TLB miss penalties would likely be even larger. However, the translation cost is unacceptable for streaming (sequential or stencil) access, for which we propose an alternative, efficient methodology.

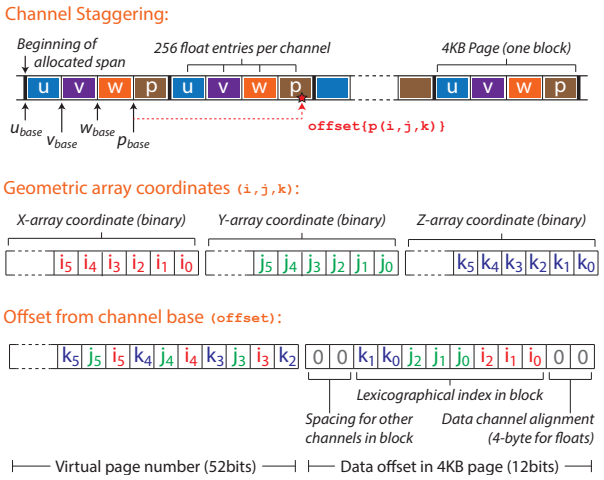


Figure 5: *Channel interleaving within a memory page, and computation of memory data offsets from geometric array coordinates.*

3.3 Access modes

True random access is very uncommon in CFD workloads. The bulk of the computational load in such applications is spent in streaming kernels and performing sequential or stencil access on data. We proceed to describe our methodology for executing such kernels at an efficiency comparable to dense arrays.

Sequential access We use the term sequential access to refer to kernels that stream through the contents of one or more data channels in a sequential fashion (e.g. following the traversal order in Figure 3) and perform operations only between data entries (from different channels) with the same array index. Scaling a data channel by a constant, copying one channel to another one, or the BLAS `Saxpy` subroutine are examples of sequential access. Reduction operations (e.g. dot products, max/min computations) also fit in this category as they follow the same access pattern. Incidentally, those are exactly the kernels used in iterative solvers such as Conjugate Gradients (in addition to a Matrix-Vector multiply, which is a *stencil* operation if implemented in a matrix-free fashion).

Within each geometric block, implementing a sequential kernel is trivial, as the data of each channel are *sequentially arranged* in memory (and also aligned, allowing the compiler to automatically issue SIMD instructions). Furthermore, TLB misses are reduced since all channels for a given block reside in the same memory page. The only question is how we iterate over the *sparse* collection of populated blocks. For this reason we use two acceleration structures: (a) When initializing the topology of our sparse array, we use a bitmap to record all geometric blocks that have been touched. Since we dedicate only 1 bit for every 4KB, or 2^{15} bits of useful data (assuming that the sparsity patterns of different channels are highly correlated) the size of this bitmap is just 0.003% of the reserved virtual memory space. For example an SPGrid storing 16 float-valued channels at an ambient resolution of $2K \times 2K \times 2K$, corresponding to 512GB of "virtual" SPGrid size, and 5GB of real data payload (assuming 1% occupancy) would only necessitate 16MB (0.3% of the useful data footprint) for this bitmap. (b) For efficient traversal of the sparse set of populated blocks, we construct a flattened array of linearized $64-$bit offsets, for the first entry of

each populated block. This array of block offsets is updated in bulk after any operation that changes the SPGrid sparsity pattern. Note that these offsets are *not channel specific*; since they point to the first entry of each block, the lower $12-$bits will be equal to zero regardless of the channel, and thus can be used even for channels with different data size (i.e. float vs. double). We note that instead of flattened block offsets, one could alternatively store the geometric coordinates of the first element of each occupied block; in this case we would pay the cost of translation just once (for the root of the block) which would easily amortize over the processing time over the remaining block entries which can be accessed sequentially. We try to avoid this, however, and seek to operate in "flattened" offset space for as many of our kernels as possible.

Stencil access Perhaps the most significant access mode from a performance standpoint involves sequential iteration over the sparse index set, and application of a geometric stencil (e.g. a 7-point Laplacian) at each location. As previously described, sequentially iterating over the locations where the stencil is to be centered is accommodated via the block offset array. The main challenge is the computation of memory locations of *geometric neighbors* that are pulled into the evaluation of the stencil. While an on-the-fly translation of geometric neighbor coordinates into memory offsets is possible, this would require 6 translations for a 7-point Laplacian in 3D; even on a Haswell processor, this limits us to about 5% of the processing throughput that the same operation would exhibit on a lexicographically ordered dense array (where stencil neighbor offsets would translate into constant strides in memory).

We propose a method for computing stencil neighbor locations *natively in linearized offset space*, without ever explicitly translating into geometric coordinates. For example, consider a stencil which includes a geometric neighbor with offset $(-1, 2, 1)$ from the reference point of the stencil. We proceed to compute the flattened version of this relative coordinate by translating the integer coordinates into a packed $64-$bit offset as in Figure 5. Negative values are encoded as 2's complement and bit-multiplexed in the usual fashion. Our key observation is that it is possible to directly add this packed offset to a base (also packed) flattened index, in a very efficient fashion. More concretely, given the packed translations of a location (i, j, k) and an offset $(\delta i, \delta j, \delta k)$ it is possible to directly evaluate the packed translation of the sum $(i + \delta i, j + \delta j, k + \delta k)$ without converting to geometric (unpacked) coordinates.

As a first step, we describe a simpler operation, which we label `MaskedAdd` which extracts the same collection of bits (according to a bit mask) from its two $64-$bit inputs I and J, performs normal addition on the extracted values, and then distributes the bits of the addition result in a $64-$bit final value according to the same mask:

```
template<uint64_t mask>
uint64_t MaskedAdd(uint64_t I,uint64_t J)
{return ((I & mask) + (J | ~mask)) & mask;}
```

In essence, `MaskedAdd` inserts zeros in I on each bit location where the `mask` is zero, and respectively inserts ones in the same bit locations for J. This train of 1-bits injected between the bit locations that are specified by the mask acts as a vehicle to propagate any carries to the next effective bit location. Three `MaskedAdd` invocations can be combined to add two bit-packed 3D coordinates, using the three respective bit masks:

```
template<uint64_t xm,uint64_t ym,uint64_t zm>
uint64_t PackedAdd(uint64_t I,uint64_t J)
{return MaskedAdd<xm>(I,J)|
    MaskedAdd<ym>(I,J)|MaskedAdd<zm>(I,J);}
```

When using constant offset stencils (e.g. uniform grid Laplacians) we pre-translate the geometric neighbor offsets into the respective packed offsets. We then fetch neighbors in a stencil application by executing the `PackedAdd` routine, which is very efficient; on a Haswell processor it takes approximately 5 cycles, while on Ivy Bridge systems it executes in 7 cycles (further savings are possible if either I or J are compile-time constants, which is frequently the case). Notably, the assembly code generated for `PackedAdd` uses 21 instructions. However the specific instruction mix of integer operations distributes extremely well on the multiple functional units of these CPUs, enabling superscalar execution.

Bulk stencil operations and the offset shadow-grid The `PackedAdd` mechanism is very efficient and achieves almost 50% of the dense array throughput on some Haswell processors (Intel Xeon E3-1200 v3 family). On some Ivy Bridge platforms, however, which offer a higher ratio of memory bandwidth to compute capacity (e.g. Intel Xeon E5-2600 v2 family) dense arrays can still achieve about $3\times$ higher throughput. A further optimization is possible, in scenarios where we expect *active* SPGrid blocks to have a rather high occupancy (50% or more). This is certainly the case for our CFD application, as our SPGrids store narrow-bands with a width of 8-16 voxels, and our $4 \times 4 \times 4$ geometric blocks (for 16 data channels) are consequently very well utilized. The main idea is that within a block `PackedAdd` is invoked in many instances where it is not strictly needed (for example, the "right" neighbor of the first voxel of each block can just be found at the adjacent memory location), and many results of `PackedAdd` are multiply computed as the same voxel is reached from many neighboring locations. When we know our stencil has narrow support (e.g. Laplacian), we can precompute in-bulk all linearized offsets in an extended window (sized $6 \times 6 \times 6$ in our case) reaching one voxel outside our geometric block. Due to the very specific nature of this computation, further optimizations are possible: For example, offsets within the $4 \times 4 \times 4$ region of this window (corresponding to the block itself) are always constant (and corresponding to index offsets $0 - 63$). Also, many of the offsets at the boundary of this window can be computed with `MaskedAdd` operations instead of the full `PackedAdd`. In reasonably wide-banded datasets, this *offset shadow-grid* optimization nearly matches the throughput of dense storage and access, for all modern processor variants.

Performance We executed a series of benchmarks, comparing SPGrid with OpenVDB and static (lexicographically ordered) C++ arrays. As an example of a *streaming* (sequential-access) kernel, we used the OpenVDB `offset` filter, while the `Laplacian` tool was used as an example of a *stencil* kernel. The implementation using conventional C++ arrays allocated a full dense array, and employed a blocked traversal to optimize cache efficiency. To emu-

	Type		Timing (in seconds)	
			Dense	Narrow band
Serial	Streaming	SPGrid	.020452	.015052
		VDB	.075464	.056312
		C-style	.0143	.0267
	Stencil	SPGrid	.064024	.047056
		VDB	.600756	.44208
		C-style	.0488	.0944
4 cores	Streaming	SPGrid	.009584	.006828
		VDB	.021228	.015928
		C-style	.0065	.0119
	Stencil	SPGrid	.02428	.018232
		VDB	.169508	.126796
		C-style	.0148	.0227

Table 2: *Timing for streaming/stencil operations on dense and narrow band data sets for SPGrid, OpenVDB, and static C++ arrays.*

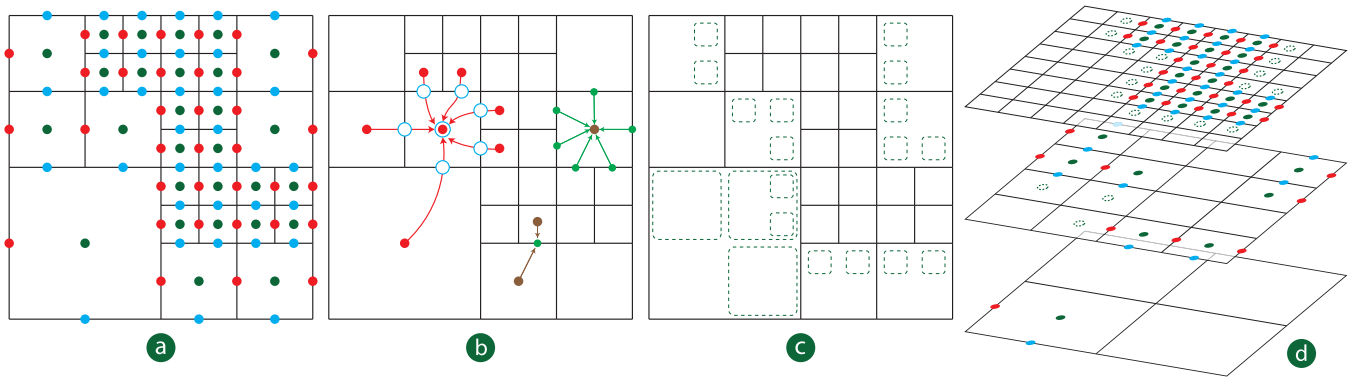


Figure 6: *(a) Pressure (green) and velocity (horizontal in red, vertical in blue) variables at their native locations in our octree discretization, (b) illustration of common stencils: a pressure gradient in brown, velocity divergence in green, and a pressure Laplacian in red - blue circles indicate locations where the stencil coefficients are stored (if necessary) on the lattice, (c) ghost cells introduced per level in our pyramid of sparsely populated grids, (d) distribution of simulation variables in our pyramid, ghost cell variables are plotted as dashed circles.*

late a more typical use scenario, the SPGrid version allocated eight $32-$bit data channels, resulting in a block size of $8\times4\times4$, although most channels remained unused in our test kernels. Two datasets were considered: (a) a *dense*, fully populated 256^3 uniform grid (16.8M voxels), and (b) a *sparse* sphere-shaped narrow band example embedded in a 1024^3 grid, containing 11.8M active voxels. The benchmarks were run on a 4-core 3.5Ghz Haswell system (Intel Xeon E3-1270 v3), both in serial and multithreaded execution. OpenVDB version 1.2.0 was used for our tests; our reported VDB performance numbers should not be construed as the performance ceiling of the library (performance could be higher in later versions), but merely as the results of our best effort in applying VDB to these workloads. Table 2 summarizes the results.

A few points to note: The conventional, lexicographically ordered C++ array was the best performer in the dense dataset; SPGrid was able, however to match about 70% of its throughput. For the narrow-band dataset the C++ array underperformed SPGrid, presumably due to suboptimal prefetching (and lower occupancy of cache lines with data). Both VDB and SPGrid do not appear to suffer a significant performance degradation when operating on narrow-band data, which would be justified due to the tighter clustering of geometrically proximate array entries in memory they both employ. SPGrid was shown to outperform VDB in these kernels, especially the Laplacian computation (which we largely attribute to the offset shadow-grid optimization). All of these kernels were memory-bound in their multi-threaded variant; when tested on an Ivy Bridge Xeon E5-2600 v2 system, the observed performance closely matched the memory bandwidth ratios of the two machines.

4 Adaptive Discretization

Adaptive discretizations such as octrees have been successfully used for CFD simulations [Losasso et al. 2004; Ferstl et al. 2014]. In certain cases, sequential operations (e.g., adding two channels) can be optimally efficient when operating on all leaf nodes if those have been explicitly flattened/linearized. However, the application of stencils is not a straightforward proposition as accessing neighbors might require the traversal of several levels in the tree-based representation. For non-graded octrees, this issue is exacerbated by the fact that the neighbor count is not known in advance; Thus, if stencil applications are performed one cell at a time, we end up underutilizing bandwidth because data cache lines are not used entirely, suffer high latency because of indirection, and do not exploit prefetching because the access patterns are not regular. Due to these reasons, conventional tree-based representations of octrees suffer a significant runtime penalty compared to dense uniform grids.

4.1 Pyramid of sparsely populated uniform grids

We propose a new scheme to ameliorate the above issues by orchestrating a sequence of actions which ensure the correct result of applying a stencil at all cells, and fully utilize bandwidth and the benefits of prefetching. Our key idea is to replace the tree-based octree representation with a pyramid of sparsely populated uniform grids, as shown in Figure 6(d). We use SPGrid to store the sparsely populated grid at every level of this pyramid. Our objective is to solve the incompressible inviscid Navier-Stokes equations

$$\mathbf{u}_t + (\mathbf{u}\cdot\nabla)\mathbf{u} + \frac{\nabla p}{\rho} = \mathbf{f} \quad (1)$$

$$\nabla\cdot\mathbf{u} = 0 \quad (2)$$

with a splitting scheme as in [Stam 1999]. Here, $\mathbf{u} = (u, v, w)$ is the vector velocity field, $\mathbf{f}$ encapsulates external forces, p is the scalar pressure field, and ρ is the fluid density. The fundamental operations that go into solving equations (1) and (2) are: scalar advection of cell-centered densities, advection of each face-centered velocity component, followed by a Poisson projection step of the velocity field to make it divergence free. The persistent simulation state variables for a smoke simulation are a scalar density field stored at cell-centered locations, and a vector velocity field stored component-wise on X, Y and Z-oriented faces. The scalar pressure field is stored at cell-centered locations. We use four float-valued SPGrid channels to store these variables. Finally, we dedicate an additional SPGrid channel (of type `uint32_t`) for storing Boolean flags associated with the geometry of the domain, for example, to indicate cells that are present in the octree and also faces that carry velocity degrees of freedom. When translating from the geometrical concept of an octree to the pyramid (see Figure 6(a),(d) respectively), we need to identify cells and faces that carry degrees of freedom. Thus, we define *active* cells and faces as follows:

- A cell at a given level of the pyramid is *active* if it is geometrically present and undivided in the octree.
- A face at a given level is *active* if that face is geometrically present and undivided in the octree. This implies that an active face has at least one cell neighbor that is active at its level.

As seen in Figure 6(d), every pyramid level collects all active variables belonging to it. We now describe how to emulate the main routines of an adaptive CFD smoke simulation with our two fundamental algorithmic kernels: stencil operations within a single level, and transfer operations between adjacent levels in the pyramid.

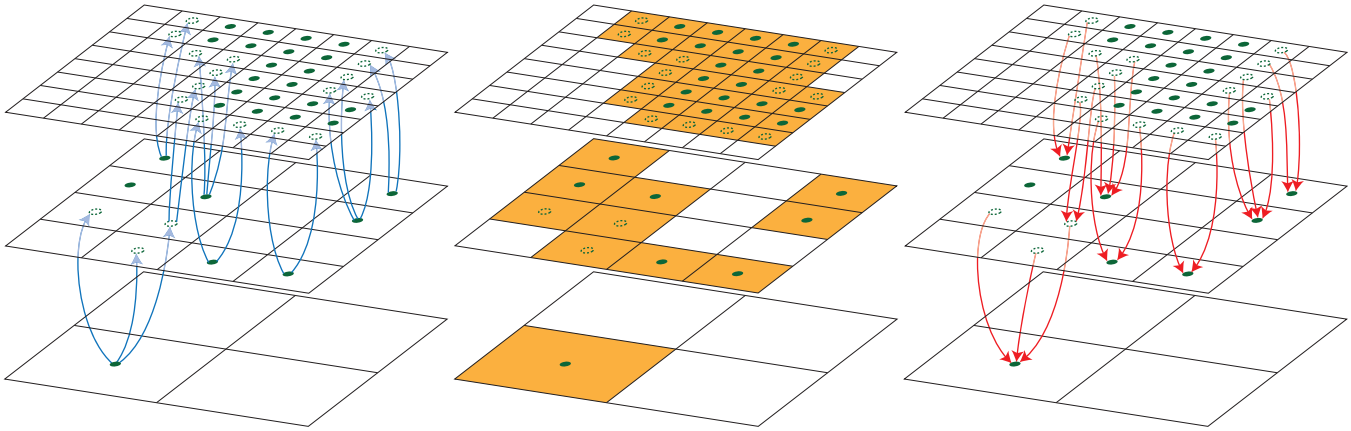


Figure 7: *Assembly of our Laplacian operator as a concatenation of uniform grid operations. (Left) Ghost values are progressively propagated from their parent cells from the coarsest to the finest grid, (Middle) a uniform Laplacian is evaluated independently on every level of the pyramid, and (Right) components of the per element Laplacian are successively accumulated from the finest grid to the coarsest grid.*

4.2 Discrete operators

For notational convenience, we index each level with $1 \leq l \leq L$, where lower indices denote finer grids. We denote a cell that natively lives at level l and has multidimensional index I as C_I^l. When remapping the octree to the pyramid, we introduce *ghost* cells - a cell C_I^l is ghost if the following three conditions are *all* jointly met:

1. C_I^l is not active at level l,
2. C_I^l neighbors a cell that is active at a level $s \leq l$,
3. There exists a *coarse* parent of C_I^l at level $l^* > l$ that is active.

See Figure 6(c) for an example. This description of ghost cells allows us to cleanly emulate even non-graded level transitions with no additional modifications to our algorithm. Our two kernels that operate on cell-centered data distributed across the pyramid are:

1. *Ghost Value Propagate* - an upsampling routine in which data from level l is *copied to* fine ghost children at level $l-1$ (see Figure 7, left). This operation is successively applied from coarser to finer grids at level transitions.
2. *Ghost Value Accumulate* - a downsampling routine in which data in level l *accumulates* contributions from any fine ghost children at level $l-1$ (see Figure 7, right). This operation is successively applied from finer to coarser grids.

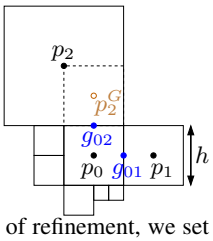
Implementation impact In a fashion similar to the native computation of stencil offsets in Section 3, the uniform grid nature of the pyramid levels allows us to compute pairs of child and parent indices natively using linearized offsets without converting back and forth into geometric coordinates. Given that our grids are sized to powers of 2, the parent index can be computed from the child index with simple shifts and inexpensive logical operations. We refer the reader to the code in our project page for implementation details.

Note that from an algebraic perspective, *Ghost Propagate* and *Ghost Accumulate* are adjoint operators. These very simple and parallelizable kernels allow us to emulate multi-resolution operations such as the discrete gradient, divergence, and Laplace operators.

Discrete Gradient Similar to [Losasso et al. 2004], we define pressure gradient values at each face bordering active cells. In the sample configuration shown below, the following gradient values are defined at the level of resolution of cell containing p_0

$$g_{01} = \frac{p_1 - p_0}{\Delta_{01}}, \quad g_{02} = \frac{p_2 - p_0}{\Delta_{02}}$$

Here, Δ_{0i} represents a scaling that depends on the sizes of the two cells bordering the respective face. As mentioned in [Losasso et al. 2004], for the case where the gradient lives between two cells of the same size, Δ should correspond to the distance between the two cell centers, for example, $\Delta_{01} = h$. They also mention that there is a certain amount of flexibility in the choice of Δ for the case where a face borders two cells of different resolutions. The flexibility arises because the divisor does not affect the ability of the resulting linear system to make the velocity field divergence free. However, the degree by which the linear system approximates the Poisson operator depends on the divisor. In our case, we set $\Delta = (h + h/2)/2 = 3h/4$ for a single level transition. If a face connects cells that differ by more than one level of refinement, we set $\Delta = 3h_{\min}/2$, where $h_{\min}$ is the size of the smaller of the two cells. This convention allows our propagation/accumulation kernels to be free of any external scaling factors, and simplifies the code. Now, consider the gradient g_{02}, which needs to access values p_0, p_2, which live on different levels of the SPGrid pyramid. Instead of performing on-the-fly traversal of the pyramid, we use the *Ghost Propagate* routine, in bulk, before starting the gradient computation, to propagate the value of p_2 into its ghost copy p_2^G which is the same level where p_0 resides. Ultimately, we simply compute $g_{02} = \frac{p_2^G - p_0}{\Delta_{02}}$. The very definition of a ghost cell guarantees that a gradient computation at the location of an active face will *always* have either *active* or *ghost* pressures at either side, and can thus be computed natively at the present level. In summary, for computing gradient values at all active faces, we first use our propagation kernel to copy pressure values from coarse to fine cells (see Figure 7, left). These values can then be directly used to compute the gradient at each level without additional considerations.



Divergence Similar to [Losasso et al. 2004], we compute the volume-weighted velocity divergences on cell centers as follows

$$\mathrm{V}_{\text{cell}} \nabla \cdot \mathbf{u} = \sum_{\text{faces}} (\mathbf{u}_{\text{face}} \cdot \mathbf{n}) \mathrm{A}_{\text{face}} \tag{3}$$

where $\mathbf{n}$ is the unit normal vector pointing out of a face, and A_{face} is the area of that face. Similar to our implementation of the gradient operator, instead of collecting the contributions of all faces (from different levels) to a given cell's divergence, we use the ghost cells to collect divergence components, starting at the level where each of the surrounding faces is active. In our insert example above, the contribution of the face marked g_{02} to the divergence stored at location p_2 will first be placed in the ghost value p_2^G, and accumulated

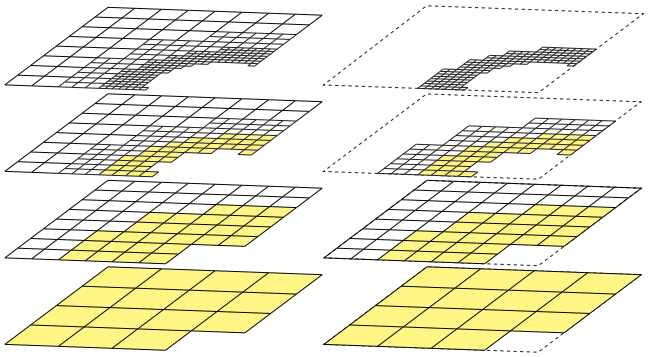


Figure 8: *Left: Conceptual hierarchy for our Multigrid V-cycle. Right: The actual pyramid used in our equivalent implementation. Note that the number of Multigrid levels is independent (here, higher) than the number of distinct levels of refinement in the initial octree. Cells introduced purely for Multigrid are shaded yellow.*

(at the end of all contributions) back to the parent (p_2) of the ghost cell. In summary: We first process each level of the pyramid independently; the contribution of every face to the divergence is first added to the ghost cell that immediately borders it. After all levels have been individually processed, we perform a sweep of *Ghost accumulate* calls, from finest to coarsest level (Figure 7, right).

Laplacian The Laplacian operator is composed by combining the gradient and divergence operator in sequence. Since the *Ghost propagate* routine appears at the beginning of this sequence, and the *Ghost accumulate* at the end, we combine the per-level gradient and divergence (in the middle of the sequence) to yield a per-level Laplacian. The complete sequence is illustrated in Figure 7. Composing the adjoint gradient and divergence operators together results in a symmetric Laplacian operator, as expected.

5 Adaptive Multigrid Preconditioner

A collateral benefit of our method for computing the Laplacian is that the operator is not explicitly computed, but applied in a matrix-free fashion. This benefit is in part negated if an Incomplete Cholesky factorization is used. First, the explicit matrix needs to be extracted and second, the factorization of the matrix does not align optimally with our SPGrid pyramid concept. However, it is noteworthy that even though the factorization is not a stencil-friendly operation, the forward and backward substitution operations can be SPGrid-optimized as shown in the supplemental document. Still, we view this as an obstacle to our SPGrid discretization attaining its full potential, as explicit preprocessing is required every time the underlying octree topology changes, and using Incomplete Cholesky factors inevitably makes preconditioning a bandwidth bound operation. Moreover, our methodology allows us to achieve levels of discretization that push the limits of the Incomplete Cholesky factorization as a preconditioner. We propose an alternative multigrid-based preconditioner for Conjugate Gradients, which is essentially an extension of a previously documented uniform multigrid scheme [McAdams et al. 2010] to an adaptive grid.

For our Multigrid preconditioner, we conceptually envision a hierarchy of *adaptive* grids, each one of which descends to a progressively coarser level of discretization, as shown in Figure 8(left). In practice, we generate this by successively coarsening the finest level of our octree discretization into their immediate parents for as many levels as desired. Note that the depth of our Multigrid hierarchy and the depth of our initial octree do not need to be tied to one another. For example, we could envision a shallow Multigrid cycle that descends to a level of resolution where our grid is still adaptive, or we could consider a deep cycle which descends to even coarser levels of resolution than initially present in the octree discretization. Figure 8 demonstrates the latter case. For each level of the hierarchy, all cells other than the most finely refined ones are guaranteed to also exist at the immediately coarser hierarchy level. We thus restrict such cells by simply copying their values to their counterparts on the next (coarser) octree of the hierarchy. The finest cells of each level additionally distribute their contents using the standard trilinear stencil as described in [McAdams et al. 2010]. Prolongation, as usual, is constructed as the transpose operator to restriction. Finally, as in [McAdams et al. 2010], our smoother of choice is the damped Jacobi iteration which is convergent for this problem and facilitates parallelism. The one aspect in which our approach is different is that our smoother only operates on the finest cells of every Multigrid level, as coarser levels will be smoothed in subsequent levels of the V-cycle. With these modifications, we can implement a complete V-cycle using only a single SPGrid pyramid instead of a hierarchy of SPGrid pyramids, as illustrated in Figure 8(right). Of course, this requires the augmentation of *active* cells (as defined in Section 4.1) with an additional collection of cells (shaded in yellow) which are the recipients of the restriction of finer levels within the hierarchy. Since those cells are only introduced in coarser levels of resolution and only in banded regions, their memory footprint is not significant or in any way prohibitive to allocate and maintain. Our Multigrid V-cycle is summarized in Algorithm 1.

All superscripts in Algorithm 1 refer to the *SPGrid* level of a variable, i.e., which level it belongs to in the original adaptive pyramid. We first copy the right hand side to active cells (line 2) at the finest level of resolution. All other cells, including fine ghost cells receive zero. At each level l, we begin with a zero initial guess (line 4) which is required in order to use Multigrid as a preconditioner. Next, a smoother is applied only to the the finest cells at this level of the Multigrid hierarchy, as mentioned above. For this reason, along with starting with a zero initial guess, the smoother does not require any transfer operators and is simply a uniform grid stencil. After computing the residual everywhere at level l, *including* on ghost cells, we restrict residual values to level $l+1$ and accumulate ghost residual values to level $l+1$. This ensures that all cells at level $l+1$ have the proper residual value. At the bottom of the Multigrid V-cycle, a non-Multigrid solver is used to exactly solve the coarse level problem. The upstroke of the V-cycle is the adjoint of the downstroke. Here, ghost values of $\boldsymbol{u}$ are propagated such that finer cells receive information from their coarse neighbors. Finally, we smooth $\boldsymbol{u}$ as in the downstroke. Figure 9 illustrates the convergence behaviors of our solver. For the example shown in

Algorithm 1 Multigrid algorithm

1: **procedure** MG_VCYCLE($\boldsymbol{f}$,$\mathcal{L}$,L)
2: $\quad\boldsymbol{b}^{(1)} \leftarrow \boldsymbol{f}^{(1)}$ ▷ $\boldsymbol{b}^{(1)} \equiv 0$ on ghost cells
3: $\quad$**for** $l = 1$ **to** $L-1$ **do** ▷ L multigrid levels
4: $\quad\quad\boldsymbol{u}^{(l)} \leftarrow \mathbf{0}$
5: $\quad\quad$Smooth($\boldsymbol{u}^{(l)}, \boldsymbol{b}^{(l)}$)
6: $\quad\quad\boldsymbol{r}^{(l)} \leftarrow \boldsymbol{b}^{(l)} - \mathcal{L}^{(l)}\boldsymbol{u}^{(l)}$ ▷ includes ghost cells
7: $\quad\quad\boldsymbol{b}^{(l+1)} \leftarrow \boldsymbol{f}^{(l+1)} +$ Restrict($\boldsymbol{r}^{(l)}$)
8: $\quad\quad$GhostValueAccumulate($\boldsymbol{r}_G^{(l)} \rightarrow \boldsymbol{b}^{(l+1)}$)
9: $\quad$**end for**
10: $\quad$Solve $\boldsymbol{u}^{(L)} \leftarrow (\mathcal{L}^{(L)})^{-1}\boldsymbol{b}^{(L)}$ ▷ Using ICPCG
11: $\quad$**for** $l = L-1$ **down to** 1 **do**
12: $\quad\quad$GhostValuePropagate($\boldsymbol{u}^{(l+1)} \rightarrow \boldsymbol{u}_G^{(l)}$)
13: $\quad\quad\boldsymbol{u}^{(l)} \leftarrow \boldsymbol{u}^{(l)} +$ Prolongate($\boldsymbol{u}^{(l+1)}$)
14: $\quad\quad$Smooth($\boldsymbol{u}^{(l)}, \boldsymbol{b}^{(l)}$)
15: $\quad$**end for**
16: **end procedure**

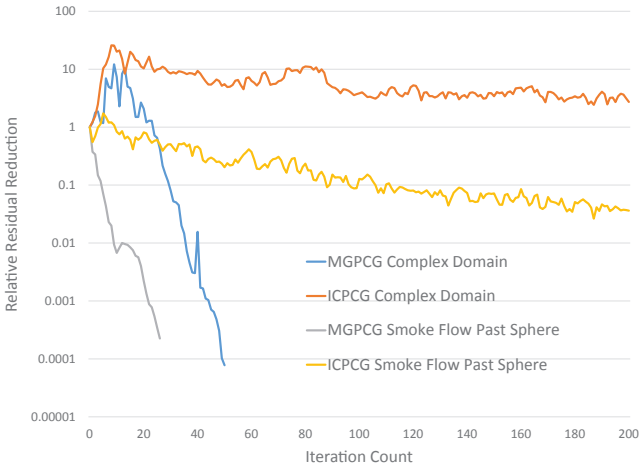


Figure 9: *Relative residual reduction factors for MGPCG and ICPCG for the examples shown in Figure 11 and 12. Note that MGPCG achieves asymptotic convergence rates in both examples.*

Figure 11, we benchmark three alternatives: a stock Incomplete Cholesky factorization using a compressed sparse row matrix storage requires around 170 sec per PCG iteration, a SPGrid-optimized Incomplete Cholesky version (as outlined in our supplemental material) requires around 1.5 sec per PCG iteration, and our proposed Multigrid preconditioned Conjugate Gradient algorithm requires 5 sec per PCG iteration (but converges in far fewer total iterations).

Preconditioner or stand-alone solver? A legitimate concern is whether we should pursue using multigrid as a preconditioner to a Krylov method, or as a stand-alone solver. Two scenarios were identified in [McAdams et al. 2010] where convergence of a "stock" standalone multigrid solver could become problematic: (a) If there is a geometric discrepancy between domains Ω^h and Ω^{2h} (for example, if we compute a voxelized approximation of the continuous domain). (b) If there are topological features at a grid spacing of h, such as thin slits or thin fingers of cells which are not resolved at a grid spacing of $2h$, further discrepancies would be observed in the discretized operators. In our adaptive setting, we are also confronted with a third challenge: (c) When using a first order approximation (as in [Losasso et al. 2004]) to a continuous discrete operator (in our case the Poisson operator), further discrepancies may arise between discretizations at different grids due to the presence (or not) of regions where the local truncation error is first order.

Multigrid-native remedies do exist for these challenges. For (a) typically the convergence behavior is normalized by smoothing more near boundaries. The methodology of [Ferstl et al. 2014] could be used to remedy (b), by introducing nonmanifold features in coarser levels of the hierarchy; unfortunately this would compromise the regularity and performance of SPGrid. More complex and esoteric Multigrid remedies exist for (b) and (c) such as recombined iterants or identification of the actual modes of discrepancy (see [Trottenberg et al. 2001]) and re-incorporation in a least-squares solve. Fortunately, it has been well-documented that scenarios (b) and (c) can be very successfully treated by wrapping a Krylov subspace solver around Multigrid. In addition, the intensity of extra boundary smoothing due to (a) can be kept to moderate levels, if only using the cycle as a preconditioner. This should not be construed as a defeatist move and an attempt to circumvent the problem, but as a calculated decision to trade a small number of additional PCG iterations for a much more complicated problem-specific Multigrid-centric solution. Typically, the only price we pay is a very slight increase in PCG iterations. This was certainly the case observed in our examples; for the complex flow of Figure 11 an extra few PCG

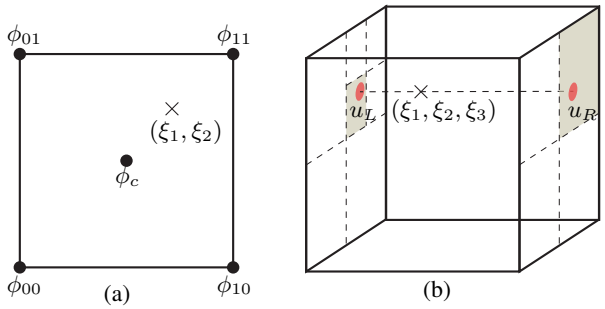


Figure 10: *(a) Interpolation from cell-centered values in two spatial dimensions, (b) Interpolation from face-centered values in three spatial dimensions.*

iterations were used, but no exotic modifications in our V-Cycle were necessary to ensure good preconditioning.

6 Advection Scheme

Lastly, we detail a minor contribution which complements the design choices offered by SPGrid and the pyramidal storage structure. In various parts of our simulation pipeline, most prominently in the course of the advection step or when performing modifications to the adaptive layout of the fluid domain, we need to perform unstructured interpolation, i.e., computation of the physical quantities at locations other than specific grid-aligned locations. In addressing this need, we have to balance convenience and accuracy. Of course, interpolation is easily handled in uniform regions of the fluid domain by standard bilinear interpolation in two spatial dimensions and trilinear interpolation in three spatial dimensions. However, it is not very clear how to interpolate values near areas of level transition. To handle such cases, [Losasso et al. 2004] averaged values to nodes which, albeit very convenient, is quite dissipative. In fact, their scheme suffers from dissipation even if the velocity field was zero everywhere - leading to increased dissipation with small time steps. This yields the awkward scenario of preventing the simulation from taking extremely small time steps to avoid excessive numerical dissipation. Ideally, an interpolation scheme should

- return the exact value if the lookup location coincides with a location where the physical quantity is natively stored,
- keep any advected quantity asymptotically unchanged, at the limit of the time step approaching zero,
- retain the convenience of the node-based averaging approach [Losasso et al. 2004] as much as possible.

Motivated by the above criteria, we propose the following interpolation scheme. As a first step, we perform an averaging of the simulation quantities from their native locations (face centers or cell centers) to nodes, qualitatively similar to [Losasso et al. 2004]. However, we use a different scheme for weighting contributions of quantities that originate from cells of different resolution. In particular, we perform a weighted contribution of every face-centered (or cell-centered) value to all of its surrounding nodal locations, where the weight is inversely proportional to the feature size of the geometric primitive where that quantity lives. Doing so guarantees that nodal values are interpolated without error for *linear* fields. Once we have a full contingent of both face-centered (or cell-centered) and node-centered values, then we interpolate as described next.

6.1 Per-cell interpolation

Consider a two dimensional example for interpolation from cell-centered quantities at a location with bilinear coordinates (ξ_1, ξ_2), as shown in Figure 10(a). First, we compute a bilinearly interpolated value ϕ_c^I at the cell center from the nodal values $\phi_{00}, \ldots, \phi_{11}$,

i.e.,

$$\phi_c^I = (\phi_{00} + \phi_{10} + \phi_{01} + \phi_{11})/4$$

Next, we compute a correction $\delta\phi_c = \phi_c - \phi_c^I$, similar in spirit to [Brochu et al. 2010; Ando et al. 2013]. Finally, the result of our interpolation scheme is computed as follows

$$\begin{aligned} \phi_{(\xi_1,\xi_2)} &= \phi_{00}(1-\xi_1)(1-\xi_2) + \phi_{10}\xi_1(1-\xi_2) \\ &+ \phi_{01}(1-\xi_1)\xi_2 + \phi_{11}\xi_1\xi_2 \\ &+ 2\delta\phi_c \cdot \min\{\xi_1, 1-\xi_1, \xi_2, 1-\xi_2\} \end{aligned} \quad (4)$$

Note that $\phi_{(\xi_1,\xi_2)} = \phi_c$ when (ξ_1, ξ_2) coincides with the location of the cell center. An analogous scheme is used for interpolating values from cell-centered data in three spatial dimensions.

6.2 Per-face interpolation

Consider a three dimensional example for interpolation from face-centered quantities at a location with barycentric coordinates (ξ_1, ξ_2, ξ_3). Without loss of generality, let us assume that the quantity being interpolated lives on faces normal to the x-axis, as shown in Figure 10(b). First, we obtain the values u_L, u_R at the projected locations on the left and right faces. This is done by identifying the undivided faces (shown shaded in Figure 10) that the locations x_L and x_R belong to and using equation (4) to compute the value at these locations from the nodal and face-centered values. The final interpolated value $u_{(\xi_1,\xi_2,\xi_3)} = u_L(1-\xi_1) + u_R\xi_1$.

7 Smoke Flow Examples

We demonstrate the effectiveness of our SPGrid framework through several examples. Figure 11 shows a complex domain where smoke flows in from the bottom left and exits from the top right. Our Multigrid preconditioned solver is able to accurately capture the incompressible behavior, while ICPCG fails to converge with 200 PCG iterations and produces a compressible flow field. Figure 12(top) shows smoke flow past a sphere with a source at the bottom. The adaptive placement of resolution produces large amounts of small scale detail that cannot be captured with a uniform grid using a comparable number of degrees of freedom (see the supplemental video). Figure 12(bottom) shows smoke emanating radially outwards from a sphere. Our method is able to capture a large number of vortices *without* explicit enhancements such as vorticity confinement [Fedkiw et al. 2001] or vortex particles [Selle et al. 2005]. In fact, none of our examples use such techniques to generate additional detail than that resulting from the underlying Eulerian discretization. Finally, Figure 13 shows an example where the underlying topology is changing dynamically. Table 3 shows the timing breakdown for all examples. **Implementation notes:** We implemented our smoke simulation pipeline within the SPGrid framework using 16 channels. We use four channels for storing the smoke density and velocity field, five channels for computing nodal velocities during advection and using as temporaries during interpolation. Our Multigrid preconditioner uses four channels, and PCG requires four channels as well. We dedicate a channel to storing Boolean flags associated with the geometry of the domain.

8 Limitations and Future Work

A number of limitations for SPGrid were listed in Table 1, corresponding to conscious choices in the interest of optimal throughput. Our current mechanism for dynamically adapting the topology initializes a new pyramid every time, however, we would like to move to the paradigm where we can instruct the virtual memory system to "forget" that a page had ever been touched (as a consequence, such page would be re-instanced and zero-filled on the next access attempt). Under Linux, this is provided via the `madvise` system call. Additionally, we have focused on the Linux programming interface for our current implementation; as future work we will produce a port of the SPGrid library that also runs under the Windows operating system. Finally, our smoke simulation scheme was very elementary, and we did not leverage opportunities for more accurate advection schemes, methods to counteract numerical dissipation [Olshanskii et al. 2013], and higher order accurate discretizations of the Poisson operator [Losasso et al. 2005]. We will address such simulation features, along with an investigation of applying SPGrid to free surface flows in future work.

	Fig. 11	Fig. 12 (top)	Fig. 12 (bottom)	Fig. 13
Time step	336	575	650	756
Advection	9	26	48	18
Projection	319	525	535	645
One PCG iteration	9	20	29	28
Write state to disk	2	5	16	3
Grid adaptation	N/A	N/A	N/A	111

Table 3: *Averaged timing breakdown (in seconds) for smoke flow examples. The first 3 columns were run on an Intel Xeon E5-2670; the last column was run on an Intel Xeon E5-2650.*

Acknowledgements The authors would like to thank Mark Hill, David Wood and Mike Swift for many insightful discussions at the early stages of this work. This work was supported in part by NSF grants IIS-1253598, CNS-1218432, IIS-1407282 & CCF-1423064.

References

ADAMS, B., PAULY, M., KEISER, R., AND GUIBAS, L. J. 2007. Adaptively sampled particle fluids. In *ACM SIGGRAPH 2007 Papers*, ACM, New York, NY, USA, SIGGRAPH '07.

AFTOSMIS, M. J., BERGER, M. J., AND MURMAN, S. M., 2004. Applications of space-filling curves to cartesian methods for CFD. 42nd Aerospace Sciences Meeting and Exhibit.

ANDO, R., THÜREY, N., AND WOJTAN, C. 2013. Highly adaptive liquid simulations on tetrahedral meshes. *ACM Trans. Graph. 32*, 4 (July), 103:1–103:10.

BATTY, C., XENOS, S., AND HOUSTON, B. 2010. Tetrahedral embedded boundary methods for accurate and flexible adaptive fluids. *Computer Graphics Forum 29*, 2, 695–704.

BERGER, M., AND COLELLA, P. 1989. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys. 82*, 64–84.

BROCHU, T., BATTY, C., AND BRIDSON, R. 2010. Matching fluid simulation elements to surface geometry and topology. In *SIGGRAPH '10*, 47:1–47:9.

BRUN, E., GUITTET, A., AND GIBOU, F. 2012. A local level-set method using a hash table data structure. *J. Comput. Phys. 231*, 6 (Mar.), 2528–2536.

CHENTANEZ, N., AND MÜLLER, M. 2011. Real-time eulerian water simulation using a restricted tall cell grid. In *SIGGRAPH '11*, 82:1–82:10.

CHENTANEZ, N., FELDMAN, B. E., LABELLE, F., O'BRIEN, J. F., AND SHEWCHUK, J. R. 2007. Liquid simulation on lattice-based tetrahedral meshes. SCA '07, 219–228.

CLAUSEN, P., WICKE, M., SHEWCHUK, J. R., AND O'BRIEN, J. 2013. Simulating liquids and solid-liquid interactions with lagrangian meshes. *ACM Trans. Graph. 32*, 2 (Apr.), 17:1–17:15.

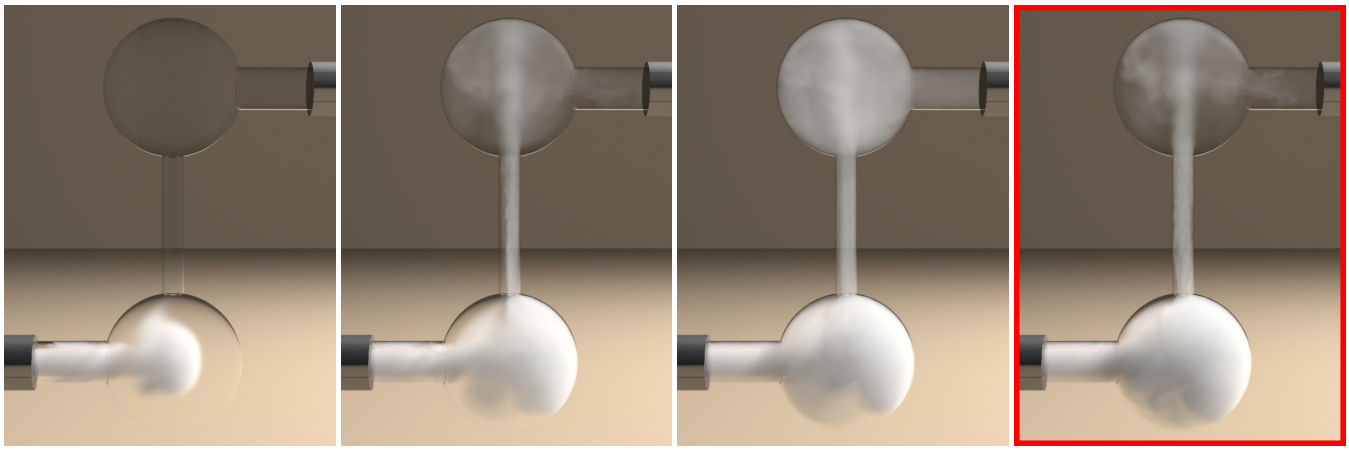


Figure 11: *Smoke flow through complex geometry with a thin nozzle simulated using* 36 *million voxels and MGPCG with* 6 *levels. Effective resolution is* $1K \times 1K \times 2K$ *with a physical memory footprint of* 25*GB. The page table size is* 56*MB. (Far right) Same as left but simulated using ICPCG; failure to converge makes the flow look significantly different from incompressible expectations.*

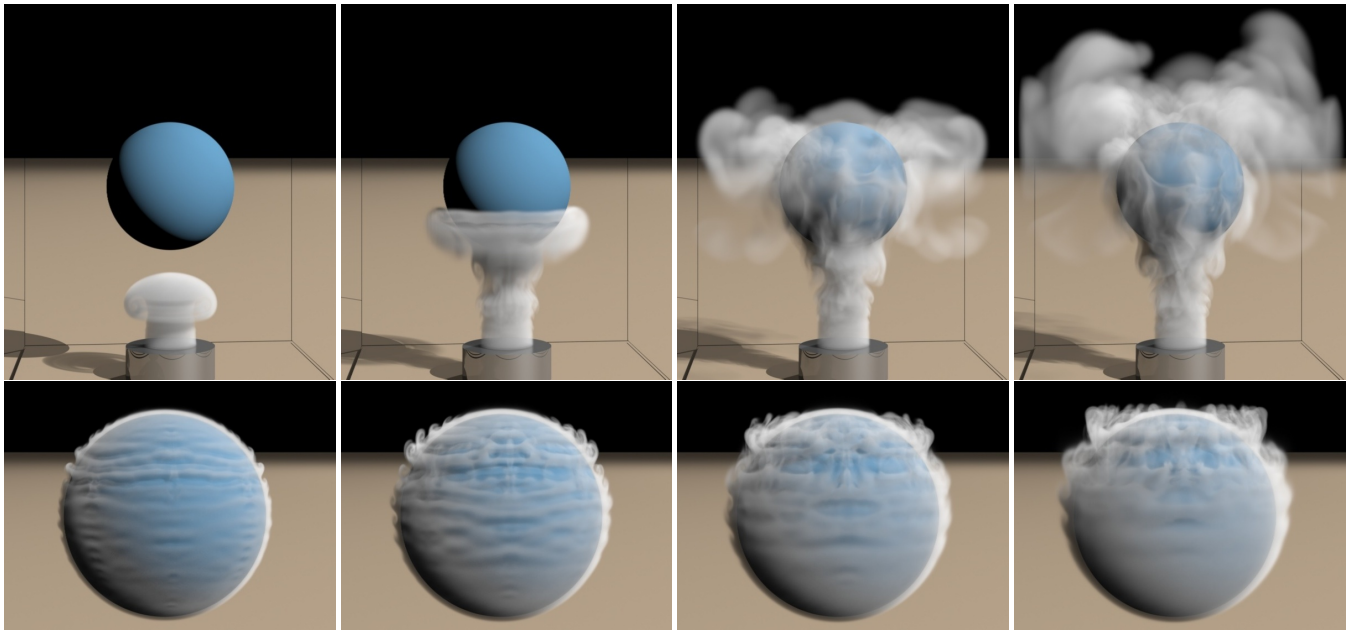


Figure 12: *(Top) Smoke flow past sphere using* 135 *million voxels and MGPCG with* 5 *levels. Effective resolution is* $1K \times 1K \times 2K$ *with a physical memory footprint of* 23*GB. Page table size is* 172*MB. (Bottom) Smoke emanating radially from a sphere simulated using* 409 *million voxels and* 6 *MGPCG levels. Effective resolution is* $2K \times 2K \times 4K$ *with a physical memory footprint of* 126*GB. Page table size is* 259*MB.*

COHEN, J., TARIQ, S., AND GREEN, S. 2010. Interactive fluid-particle simulation using translating Eulerian grids. In *ACM SIGGRAPH Symp. on Interactive 3D Graphics and Games*, 15–22.

CRASSIN, C., NEYRET, F., LEFEBVRE, S., AND EISEMANN, E. 2009. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. I3D '09, 15–22.

DESBRUN, M., AND CANI, M.-P. 1996. Smoothed particles: A new paradigm for animating highly deformable bodies. In *Comput. Anim. and Sim. '96*, Springer-Verlag, 61–76.

DOBASHI, Y., MATSUDA, Y., YAMAMOTO, T., AND NISHITA, T. 2008. A fast simulation method using overlapping grids for interactions between smoke and rigid objects. *Computer Graphics Forum 27*, 2, 477–486.

ENGLISH, R. E., QIU, L., YU, Y., AND FEDKIW, R. 2013. Chimera grids for water simulation. SCA '13, 85–94.

FEDKIW, R., STAM, J., AND JENSEN, H. W. 2001. Visual simulation of smoke. SIGGRAPH '01, 15–22.

FELDMAN, B., O'BRIEN, J., KLINGNER, B., AND GOKTEKIN, T. 2005. Fluids in deforming meshes. SCA '05, 255–259.

FERSTL, F., WESTERMANN, R., AND DICK, C. 2014. Large-scale liquid simulation on adaptive hexahedral grids. *IEEE Trans. Visualization & Computer Graphics 20*, 10 (Oct), 1405–1417.

FRISKEN, S., PERRY, R., ROCKWOOD, A., AND JONES, T. 2000. Adaptively sampled distance fields: A general representation of shape for computer graphics. SIGGRAPH '00, 249–254.

GOLAS, A., NARAIN, R., SEWALL, J., KRAJCEVSKI, P., DUBEY, P., AND LIN, M. 2012. Large-scale fluid simulation using velocity-vorticity domain decomposition. *ACM Trans. Graph. 31*, 6, 148:1–148:9.

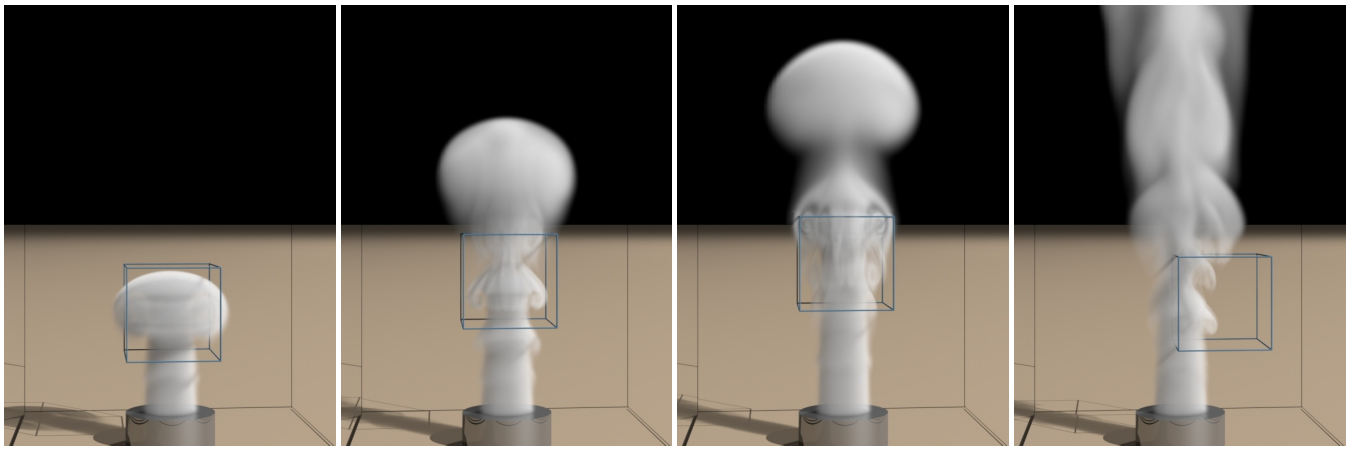


Figure 13: *Smoke flow with dynamic adaptation of topology simulated using* 59 *million voxels and MGPCG with* 5 *levels. Effective resolution is* $1K \times 1K \times 2K$ *with a physical memory footprint of* 16*GB. The page table size is* 109*MB. Higher resolution inside the box leads to the generation of interesting vortices which would otherwise not be produced in the ambient resolution.*

GOSWAMI, P., SCHLEGEL, P., SOLENTHALER, B., AND PAJAROLA, R. 2010. Interactive SPH simulation and rendering on the GPU. SCA '10, 55–64.

HARLOW, F. H., AND WELCH, J. E. 1965. Numerical calculation of timedependent viscous incompressible flow of fluid with free surface. *Physics of Fluids (1958-1988) 8*, 12.

HOUSTON, B., NIELSEN, M. B., BATTY, C., NILSSON, O., AND MUSETH, K. 2006. Hierarchical RLE level set: A compact and versatile deformable surface representation. *ACM Trans. Graph. 25*, 1, 151–175.

IRVING, G., GUENDELMAN, E., LOSASSO, F., AND FEDKIW, R. 2006. Efficient simulation of large bodies of water by coupling two and three dimensional techniques. SIGGRAPH, 805–811.

KLINGNER, B., FELDMAN, B., CHENTANEZ, N., AND O'BRIEN, J. 2006. Fluid animation with dynamic meshes. SIGGRAPH '06, 820–825.

LOSASSO, F., GIBOU, F., AND FEDKIW, R. 2004. Simulating water and smoke with an octree data structure. SIGGRAPH '04, 457–462.

LOSASSO, F., FEDKIW, R., AND OSHER, S. 2005. Spatially adaptive techniques for level set methods and incompressible flow. *Computers and Fluids 35*, 2006.

MCADAMS, A., SIFAKIS, E., AND TERAN, J. 2010. A parallel multigrid poisson solver for fluids simulation on large grids. SCA '10, 65–74.

MÜLLER, M., CHARYPAR, D., AND GROSS, M. 2003. Particle-based fluid simulation for interactive applications. SCA '03, 154–159.

MUSETH, K. 2011. DB+Grid: A novel dynamic blocked grid for sparse high-resolution volumes and level sets. SIGGRAPH '11.

MUSETH, K. 2013. VDB: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph. 32*, 3 (July), 27:1–27:22.

NIELSEN, M., AND BRIDSON, R. 2011. Guide shapes for high resolution naturalistic liquid simulation. SIGGRAPH '11, 1–8.

NIELSEN, M. B., AND MUSETH, K. 2006. Dynamic tubular grid: An efficient data structure and algorithms for high resolution level sets. *J. Sci. Comput. 26*, 3 (Mar.), 261–299.

NIELSEN, M. B., NILSSON, O., SÖDERSTRÖM, A., AND MUSETH, K. 2007. Out-of-core and compressed level set methods. *ACM Trans. Graph. 26*, 4 (Oct.).

OLSHANSKII, M. A., TEREKHOV, K. M., AND VASSILEVSKI, Y. V. 2013. An octree-based solver for the incompressible Navier-Stokes equations with enhanced stability and low dissipation. *Computers and Fluids 84*, 0, 231 – 246.

PATEL, S., CHU, A., COHEN, J., AND PIGHIN, F. 2005. Fluid simulation via disjoint translating grids. SIGGRAPH '05.

PREMOZE, S., TASDIZEN, T., BIGLER, J., LEFOHN, A., AND WHITAKER, R. 2003. Particle–based simulation of fluids. In *Comp. Graph. Forum (Eurographics Proc.)*, vol. 22, 401–410.

SELLE, A., RASMUSSEN, N., AND FEDKIW, R. 2005. A vortex particle method for smoke, water and explosions. SIGGRAPH '05, 910–914.

SIN, F., BARGTEIL, A., AND HODGINS, J. 2009. A point-based method for animating incompressible flow. SCA '09, 247–255.

SOLENTHALER, B., AND GROSS, M. 2011. Two-scale particle simulation. SIGGRAPH '11, 81:1–81:8.

STAM, J. 1999. Stable fluids. SIGGRAPH '99, 121–128.

SUSSMAN, M., ALMGREN, A. S., BELL, J. B., COLELLA, P., HOWELL, L. H., AND WELCOME, M. L. 1999. An adaptive level set approach for incompressible two-phase flows. *J. Comput. Phys. 148*, 1, 81–124.

TAN, J., YANG, X., ZHAO, X., AND YANG, Z. 2008. Fluid animation with multi-layer grids. In *SCA '08 Posters*.

TESCHNER, M., HEIDELBERGER, B., MÜLLER, M., POMERANTES, D., AND GROSS, M. 2003. Optimized spatial hashing for collision detection of deformable objects. In *VMV*, 47–54.

TROTTENBERG, U., OOSTERLEE, C. W., AND SCHULLER, A. 2001. *Multigrid*. Academic Press.

ZHU, Y., AND BRIDSON, R. 2005. Animating sand as a fluid. SIGGRAPH '05, 965–972.

ZHU, B., LU, W., CONG, M., KIM, B., AND FEDKIW, R. 2013. A new grid structure for domain extension. *ACM Trans. Graph. 32*, 4, 63:1–63:12.

Incomplete Cholesky Preconditioner

We propose novel accelerations both for computing and applying the Incomplete Cholesky preconditioner to the CG algorithm within our SPGrid framework. Although we do not keep the explicit system matrix for the entirety of the simulation, we temporarily compute it for performing an Incomplete Cholesky factorization.

Matrix Construction and Factorization

We compute the original system matrix by placing matrix coefficients in temporary channels since non-zero coefficients in a Laplace stencil correspond to active cells/faces. When we reach a face that lies at a level transition, i.e., between cells of different resolutions, we only place coefficients at the level that corresponds to the smaller cell. Since our system is symmetric, we can simply duplicate the off-diagonals.

Computing the Incomplete Cholesky factorization of this system requires a total ordering of the degrees of freedom, or in our case, active cell centers. As the choice of ordering is flexible, we first order cells by level, finer cells being ordered before coarser cells. Within each level, we order by the linearized SPGrid index. As we show below, this ordering allows for the efficient application of the preconditioner. Using this ordering, the in-place factorization can be computed by employing the algorithm in [Golub and Van Loan 1996]. The factorization computes $A = LU$, where A is the system matrix, L is a lower-triangular matrix with unit-diagonal, and $U = DL^T$ (D is a diagonal matrix). L displays the same sparsity pattern as the original matrix, i.e., an element in the factorized matrix is non-zero if and only if it is non-zero in the original matrix. This means that the factorized matrix can be stored as a diagonal and off-diagonal terms that represent active faces. Once L has been computed, we recycle the data channels used for storing the original system matrix. Ultimately, storing the factorization requires only four persistent data channels.

Forward and Backward Substituion

Given a factorization of the matrix $A = LU$, the preconditioner can be applied by solving the system $LUx = y$. This is achieved by forward and backward substitutions, first solving $Lz = y$ for a temporary variable z and then solving $Ux = z$. Consider an in-place forward substitution algorithm

Algorithm 1 In Place Forward Substitution(x)

```
1: for j = 1 to n do
2:     for i > j do
3:         x_i ← x_i − L_ij x_j
4:     end for
5: end for
```

where i and j are indices given by the specified ordering of the degrees of freedom, and x is the right hand side. This seemingly simple algorithm can cause issues in the case of adaptive discretizations as finding degrees of freedom with greater index could potentially involve a traversal of the pyramid. Given an ordering, higher indexed cell centers correspond to either cells that live at the same level of resolution or coarser cells. Furthermore, all non-zero off-diagonal elements correspond to active faces. So, for fixed j, L_{ij} is non-zero if and only if cell i shares an active face with cell j. Also the order in which the inner loop is processed is irrelevant, as long as all indices $i > j$ are covered. This essentially reduces to a stencil application, where we subtract a value from a subset of the neighbors of cell j. As mentioned, cell i could potentially live at a coarser level and we account for such cells using our transfer operators. If cell j neighbors a cell i that lives at a coarser resolution then the following conditions hold: a) the face corresponding to L_{ij} is active at this resolution because faces are always attributed to smaller cells, and b) there is a ghost cell in place of the coarser cell i. By simply treating the ghost cell as an active cell, and later using our accumulation kernel to add that value to the coarser parent, forward substitution can be efficiently carried out. Note that we have slightly modified the conventional algorithm by switching the order of iteration. Again, this allows for the utilization of the *same* grid transfer operations and stencil optimization techniques as before.

Our implementation of in-place backward substitution resembles the conventional algorithm more closely.

Algorithm 2 In Place Backward Substitution(x)

```
1: for i = n to 1 do
2:     y_i ← y_i / D_ii
3:     for j > i do
4:         y_i ← y_i − L_ji y_j
5:     end for
6: end for
```

Here, y is the right hand side. Algorithm 2 is equivalent to solving $Ux = y$ where $U = DL^T$. The inner loop iterates over cells j with index greater than i, implying that cell j either lives at the resolution of cell i or at a coarser resolution. Again, because L_{ji} corresponds to active faces that border cell i, we can use a stencil very similar to that of the Laplace operator to update y_i. Let cell j be a neighbor of cell i; if cell j is active at a finer resolution, then it can be ignored as $j < i$. If cell j is active at the same resolution as cell i, then its value can simply be used. In the case where cell j is coarser then cell i, we use our propagation kernel. The iteration order of backward substitution guarantees that when cell i is reached, all cells with index greater than i have already been solved for. Thus, at any given point of the algorithm, values at all cells coarser then cell i have already been solved for and can be freely used. After executing this algorithm at a single level, we use our propagation kernel to copy all y-values to finer ghost children. In this fashion, whenever a finer cell i needs a value from a coarse neighbor, it appears in a ghost cell.

References

GOLUB, G. H., AND VAN LOAN, C. F. 1996. *Matrix Computations*. Johns Hopkins University Press.