A scalable Schur-complement fluids solver for heterogeneous compute platforms

Haixiang Liu Nathan Mitchell Mridul Aanjaneya Eftychios Sifakis
University of Wisconsin-Madison

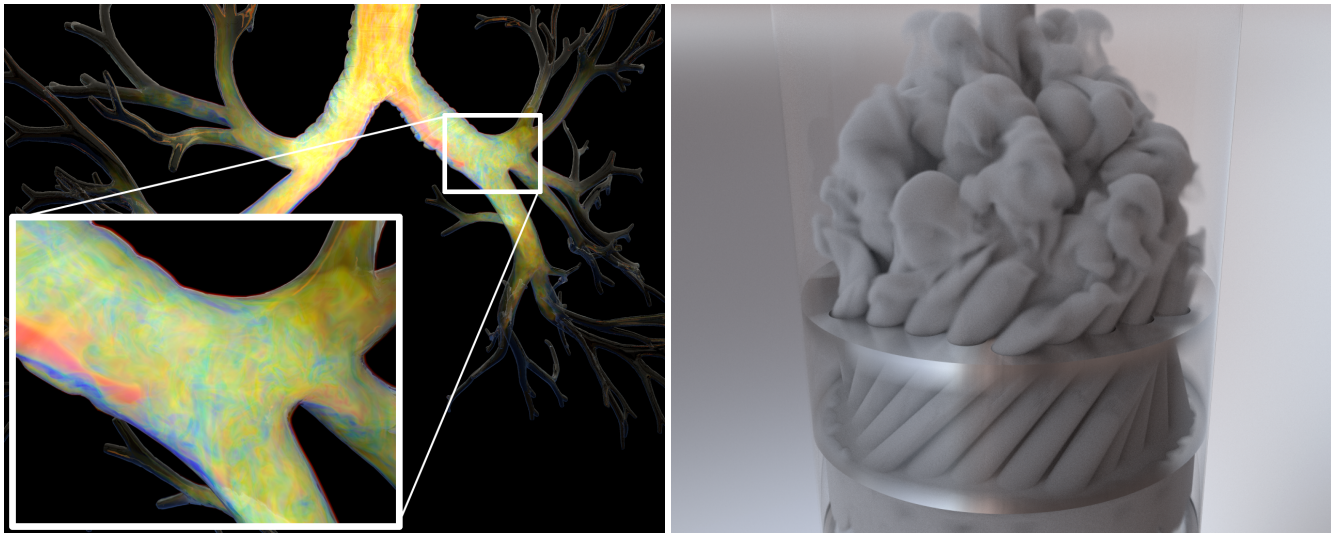


Figure 1: *Left: Smoke injected into a model of the bronchi. Color illustrates vorticity magnitude. Simulation contains 1.8 billion active cells, sparsely occupying a* $8192^2 \times 4096$ *background grid. Right: Smoke injected from the bottom of a cylinder, and forced through a metal gasket (rendered semi-transparent) with a twisted bundle of cylindrical holes. Total of 1.2 billion active cells, in a* $1024^2 \times 2048$ *background grid.*

Abstract

We present a scalable parallel solver for the pressure Poisson equation in fluids simulation which can accommodate complex irregular domains in the order of a billion degrees of freedom, using a single server or workstation fitted with GPU or Many-Core accelerators. The design of our numerical technique is attuned to the subtleties of heterogeneous computing, and allows us to benefit from the high memory and compute bandwidth of GPU accelerators even for problems that are too large to fit entirely on GPU memory. This is achieved via algebraic formulations that adequately increase the density of the GPU-hosted computation as to hide the overhead of offloading from the CPU, in exchange for accelerated convergence. Our solver follows the principles of Domain Decomposition techniques, and is based on the Schur complement method for elliptic partial differential equations. A large uniform grid is partitioned in non-overlapping subdomains, and bandwidth-optimized (GPU or Many-Core) accelerator cards are used to efficiently and concurrently solve independent Poisson problems on each resulting subdomain. Our novel contributions are centered on the careful steps necessary to assemble an accurate global solver from these constituent blocks, while avoiding excessive communication or dense linear algebra. We ultimately produce a highly effective Conjugate Gradients preconditioner, and demonstrate scalable and accurate performance on high-resolution simulations of water and smoke flow.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. © 2016 ACM.
SA '16 Technical Papers, December 05-08, 2016, Macao
ISBN: 978-1-4503-4514-9/16/12
DOI: http://dx.doi.org/10.1145/2980179.2982430

ACM Reference Format
Liu, H., Mitchell, N., Aanjaneya, M., Sifakis, E. 2016. A scalable Schur-complement fluids solver for heterogeneous compute platforms. ACM Trans. Graph. 35, 6, Article 201 (November 2016), 12 pages.
DOI = 10.1145/2980179.2982430 http://doi.acm.org/10.1145/2980179.2982430.

Keywords: heterogeneous computing, fluid simulation, pressure projection, domain decomposition, GPU, multigrid, preconditioner

Concepts: •**Computing methodologies** → **Physical simulation;**

1 Introduction

For many years, simulation-based animation of fluids has been an indispensable component of visual effects pipelines [Geiger et al. 2006; Froemling et al. 2007; Van Opstal et al. 2014]. Although the ever growing demands of production environments for higher detail and resolution have prompted some practitioners to use computer clusters for parallel solvers [Irving et al. 2006; Bailey et al. 2015], a number of innovations both in hardware as well as algorithmic foundations have made it possible to scale up the quality and resolution of fluid simulations that can be hosted on standalone servers and workstations. This has encouraged researchers to explore novel data structures [Museth 2013], careful modeling simplifications [Rasmussen et al. 2003], intricate use of GPUs [Horvath and Geiger 2009; Macklin et al. 2014], accelerated numerical techniques [McAdams et al. 2010; Zhang and Bridson 2014], and strategic use of adaptivity [Losasso et al. 2004; Ando et al. 2013].

In this paper, we seek to boost the scale and complexity of fluid simulations that can be effectively accommodated on single-computer platforms. In particular, we target the pressure Poisson equation which is almost invariably the computational bottleneck at large scales, and develop a solver that can support intricately shaped domains (see Fig.1) with more than one billion active degrees of freedom on a well-equipped (albeit, not exotic) graphics workstation. We will aggressively use any number of GPU or Many-Core accelerator cards, installed in our target platform, to facilitate this goal. However, the scale of the problems we target will present a unique challenge: many of them will be *too large* to fit in GPU memory, and although the same problem would fit in the host system memory, it would be unattractively slow to execute purely on the CPU.

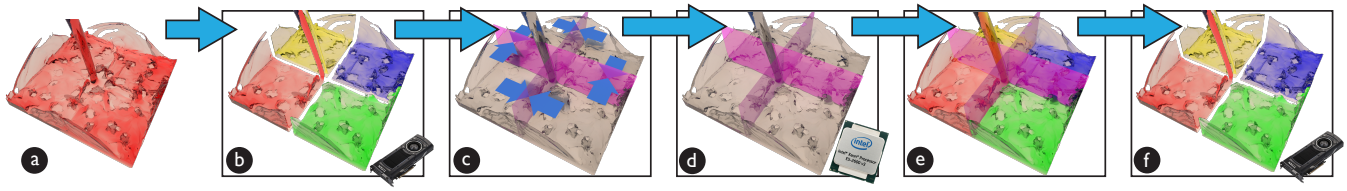


Figure 2: *Illustration of the core concept of our method: (b) We split the computational grid into subdomains, and independently solve them on the GPU(s), using zero Dirichlet conditions on subdomain boundaries. (c) Fluxes of the subdomain solutions are computed and sent to the CPU. (d) A specially formulated system is solved on the interface, using the CPU. This produces the exact value of the interface variables. (e) Those values are sent to the subdomains, and set as Dirichlet conditions. (e) A final subdomain solve on the GPU yields the global solution.*

Our technique leverages Domain Decomposition theory and, more specifically, is rooted in the principles of Schur Complement Methods [Quarteroni and Valli 1999] for the solution of elliptic Partial Differential Equations. Intuitively, our solver is designed according to a divide-and-conquer paradigm. Using *independent* solvers for small, *artificially decoupled* partitions of our domain as algorithmic building blocks, we seek to produce an accurate solver for the aggregate, fully coupled domain. The fundamental theory, which we review in Section 4, explains that this composition is perfectly possible from a purely algebraic standpoint. Furthermore, if the independent partitions of our domain are adequately sized, each decoupled Poisson problem can be dispatched to a separate GPU accelerator, and fully solved with an efficient numerical technique that optimally exploits its bandwidth and compute capability. However, direct application of this basic principle is utterly impractical, as the algebraic operations necessary to fuse together the independent partition solvers into an accurate global scheme (the *"and-conquer"* part of our paradigm) have an absolutely prohibitive cost. Our core contribution is a novel synergy of CPU-hosted numerical algorithms that drastically reduces the complexity of re-combining the independent solvers into a global one, facilitating the design of a highly efficient heterogeneous global solver. The price we pay for this dramatic complexity improvement, is that our overall scheme is no longer an exact solver; nevertheless, we show that it can be used as an excellent preconditioner for Conjugate Gradients, achieving rapid convergence in very few iterations even for very large, billion degree-of-freedom domains. Figure 2 provides a visual summary of our pipeline, which we elaborate on in later sections.

Our technical contributions are summarized as follows:

- We show a preconditioner design based on Schur Complement methods. Although the algebraic principles have precedent in scientific computing literature, we believe this is the first direct application of this theory to a graphics application.
- We present an original numerical technique for the combination of independent solvers into a coherent global scheme. This new methodology is hosted on the CPU, has very attractive complexity, and relies on the synergy of multigrid, adaptivity, and assembly-free relaxation methods.
- We analyze the efficacy of our technique in the context of large-scale smoke and water simulations, providing convergence and runtime comparisons with alternative solvers. Our tests are conducted on heterogeneous systems accelerated by multiple GPUs or Many-Core accelerator cards.

2 Background, challenges and scope

Before we venture into the discussion of prior work that inspired our approach, and the details of our scheme, it is important to provide some insight into the technological factors that motivated this work. We further offer some comments on the challenges that our technique must address, and delineate the scope of our method.

Why target heterogeneity? Heterogeneous platforms, i.e. servers or workstations equipped with one or more bandwidth-optimized accelerator cards, are presently the most price-efficient way to pack aggregate computational power into a single computer system. Today, it is very realistic for a simulation workstation (aided by a main CPU and multiple GPU cards) to combine an *aggregate* memory bandwidth of 1.1 TB/sec, peak compute throughput of 23 TFLOPS and memory capacity of 512 GB (plus any GPU memory) at a price point below $10K[1]. However, these aggregate capabilities are markedly *not uniformly available* across the platform. Each GPU card enjoys high bandwidth for internal computation, but needs to communicate through the PCIe bus to reach data stored in a different accelerator's internal memory, or in the memory of the host CPU. The CPU itself can host large workloads in its memory, and leverage task-level parallelism even in combinatorial workloads that lack the regularity that GPUs thrive on, but is limited in memory bandwidth and peak computational throughput. We note that this non-uniformity of bandwidth and compute capability is not a temporary aberration, but rather a persistent trait across generations of hardware designs. This has manifested in the design of NUMA (Non-Uniform Memory Access) multi-socket systems, is evident in GPU-accelerated platforms, and remains fully relevant in the emerging generation of accelerators that leverage high-bandwidth stacked memory (in addition to slower, but larger DDR memory). We believe that investigating numerical techniques that are deeply cognizant of this heterogeneity is timely and worthwhile, even if they are more complex than heterogeneity-agnostic alternatives.

When does offloading become a liability? For memory-bound numerical workloads, the core benefit of a GPU port primarily stems from the additional bandwidth available, internally, to each GPU card, which is often 6x-8x the bandwidth available to each CPU socket. This capability is optimally exploited when the entire problem fits in GPU memory, in which case a GPU-homogeneous solver can be used for it. However, problems that are too large to fit in GPU memory have to be split into smaller partitions, each of which needs to dispatch ("offload") its data to the GPU, carry out an operation, and then transfer the results back to the CPU to make room for the next partition. This round-trip data transfer takes place over the PCIe bus, which offers bandwidth two orders of magnitude less than what is internally available on the GPU. Thus, in order to justify this offloading overhead, it is crucial that the data remains on the GPU while carrying out useful computations for long enough to hide the PCIe transfer cost. Unfortunately, some of the best performing solvers (in terms of convergence efficiency) such as multigrid do not offer this opportunity. The building blocks of a multigrid solver, namely the smoothing routine and transfer operators, require global synchronization after their execution while carrying out only a very modest amount of useful computation in between synchronization points. In particular, a Jacobi or Gauss-Seidel style smoother requires no more than two passes over the

[1]Sample platform: SuperMicro 7048A-T, Dual Intel Xeon E5-2650 v4 processors, 512GB DDR4 RAM, 2×NVidia Titan X cards.

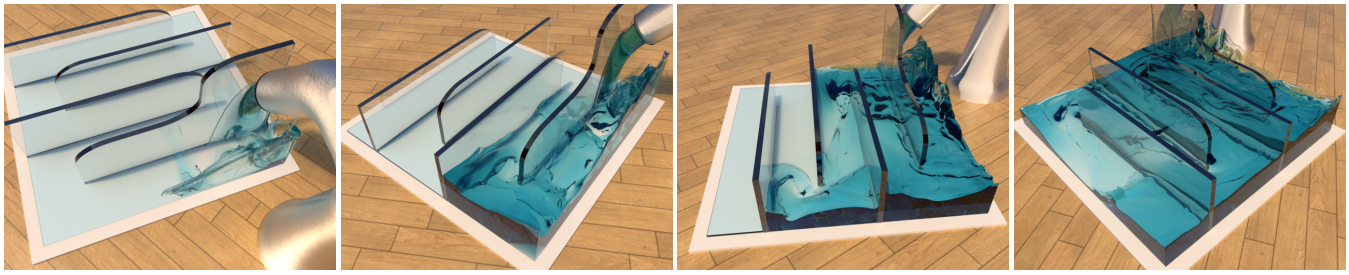


Figure 3: *Free-surface simulation of water poured in a container with multiple interior walls causing the flow to meander around them. The frame shown in the right-most illustration corresponds to* 114 *million active cells, in a* $1024\times1024\times1024$ *background grid.*

data in memory, which completes at a fraction of the time needed to move its data from/to the GPU. Of course, one could take the opportunity to carry out several smoothing iterations per offload operation; however, without synchronization at partition boundaries this extra effort will hardly translate to worthwhile gains in convergence. As a result, the benefit of the GPU offload is negated, and such large problems are better off being solved homogeneously on the CPU. Although there might be room for implementation refinements and adaptation of multigrid paradigms to curb this overhead, we are not aware of prior work that has demonstrated viability of a GPU-offload paradigm for multigrid solvers, when the problem size exceeds the memory capacity of the GPU card(s), compare to a well-optimized CPU implementation. Our proposed approach directly addresses this challenge: instead of executing just a few iterations of a smoother routine, we run an entire solver routine on the GPU for each independent subdomain we offload to it. In our case, that extra effort does translate to accelerated convergence, and the GPU computation is long enough to absorb the cost of offloading.

Divide-and-conquer with "nearly" accurate pieces The classical divide-and-conquer paradigm encountered in combinatorial algorithms often presumes building blocks that accurately solve subsets of the overall problem. In our numerical context, one of the key opportunities we will exploit is the option to design what is not an exact solver, but an excellent approximation of one, and subsequently use it as a preconditioner. In this context, we will adopt a slightly different standard which we will design our "inaccurate" divide-and-conquer scheme to satisfy. Our building block will be an inexact solver for the Poisson equation on independent partitions of our domain, which is however of good enough quality to be used as an excellent Conjugate Gradients preconditioner (i.e. it would lead to convergence in a small number of iterations, that does not significantly increase as the subdomain size grows larger). Subsequently, our objective would be to combine such "nearly accurate" building blocks into a global approximate solver that meets the same benchmark, i.e. it can be an effective preconditioner that allows CG to converge in a comparably small number of iterations as its individual constituents. Short of this standard, using divide-and-conquer with inexact components can easily result in degraded performance.

Scope Our primary objective is to demonstrate an effective, and hopefully inspiring adaptation to fluids simulation of a class of numerical techniques that has received much more exposure in scientific computing than graphics research. We note, however, that Schur Complement methods provide a general framework, and not just a singular algorithm; in fact, similar to multigrid methods, careful variations from the general algebraic theme make all the difference between a given scheme being highly effective or underwhelming for a specific application. In this vein, we consciously restricted the scope of our investigation to just uniform discretizations of fluids, specifically targeted the Poisson equation (although our formulations should readily extend to elasticity, or other elliptic problems), and did not emphasize the implications of heterogeneous computing to other parts of the fluid simulation pipeline.

3 Related Work

Fluid simulation has been an active area of research within computer graphics since the early work of [Stam 1999; Foster and Fedkiw 2001]. Since the memory overhead associated with uniform grids quickly escalates in three spatial dimensions, several adaptive techniques have been proposed, including adaptive Cartesian grids [Losasso et al. 2004; Zhu et al. 2013; Ferstl et al. 2014; Setaluri et al. 2014], adaptive tetrahedral meshes [Klingner et al. 2006; Chentanez et al. 2007; Ando et al. 2013], RLE-based schemes [Houston et al. 2006; Irving et al. 2006; Chentanez and Müller 2011], adaptive mesh refinement (AMR) and chimera grid schemes [Dobashi et al. 2008; Tan et al. 2008; Cohen et al. 2010; English et al. 2013]. Lagrangian methods present an interesting alternative because they avoid many of the numerical dissipation issues characteristic of Eulerian methods. Several methods have been proposed, including smoothed particle hydrodynamics (SPH) [Solenthaler and Gross 2011; Ihmsen et al. 2014; Bender and Koschier 2015], particle-based schemes [Adams et al. 2007; de Goes et al. 2015], position-based fluids [Macklin and Müller 2013], triangle meshes [Wojtan et al. 2010; Thürey et al. 2010; Da et al. 2015] and simplicial complexes [Zhu et al. 2014]. However, due to their unstructured nature, these methods are unable to leverage the regularity and parallelism potential of uniform grids. To circumvent this issue to some extent, hybrid methods have also been proposed [Foster and Metaxas 1996; Zhu and Bridson 2005; Losasso et al. 2008; Zhu et al. 2010; Raveendran et al. 2011; Jiang et al. 2015; Chen et al. 2015]. Authors have also investigated the use of Fast Fourier transforms [Stam 2002], which was later extended to handle slip boundary conditions [Long and Reinhard 2009], model reduction [Liu et al. 2015], and regression forests [Ladický et al. 2015].

The pressure projection step is widely accepted to be the computationally dominating step in fluid simulations, and researchers have investigated the design of fast solvers using coarse grids [Lentine et al. 2010], multigrid methods on either uniform grids [McAdams et al. 2010; Dick et al. 2016] or adaptive grids [Chentanez and Müller 2011; Ferstl et al. 2014; Setaluri et al. 2014], iterated orthogonal projection [Molemaker et al. 2008], dimension reduction [Ando et al. 2015b], fast summation methods [Zhang and Bridson 2014], and stream functions [Ando et al. 2015a]. Fast matrix factorization updates were explored [Hecht et al. 2012] by exploiting sparsity patterns in the Cholesky algorithm in ways analogous to our method, albeit in the context of elasticity simulations. An increasing number of researchers have also adopted the use of GPUs for better computational performance since efficient solvers such as multigrid tend to be memory-bound [Ament et al. 2010; Dick et al. 2011; Zhang and Bridson 2014; Chen et al. 2015; Wu et al. 2016].

Unlike previous approaches where the goal was to increase performance on homogeneous platforms, we use domain decomposition techniques to develop an efficient Krylov preconditioner whose design is tailored towards maximizing performance on heterogeneous computing platforms. One earlier investigation that does address

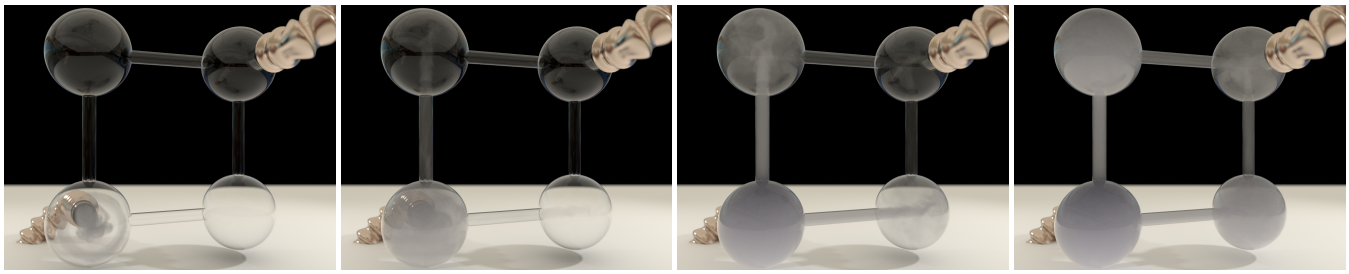


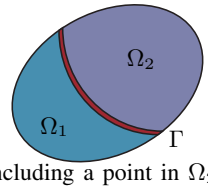
Figure 4: *Smoke flow in a network of interconnected vessels simulated using a* $1024^2 \times 512$ *background grid and* 42 *million active cells. The computational domain was divided into four subdomains. The proper flux is observed both in the inlet and the outlet of the flow.*

heterogeneity is [Jung et al. 2013], which proposed a wavelet-based method that used GPUs for increasing the performance of a multigrid solver hosted on the CPU. While researchers have proposed methods classified as domain decomposition [Golas et al. 2012; Edwards and Bridson 2015], these are quite different from ours because we work specifically in the context of Schur complement methods [Smith et al. 1996]. Methods based on Schur complements have been used for virtual surgery simulations [Bro-nielsen and Cotin 1996], skinning [Gao et al. 2014], subspace deformable body simulations [Teng et al. 2015; Wu et al. 2015], or fluid control [Raveendran et al. 2012].

4 The classic Schur complement method

We introduce the basic principles of the Schur complement method [Quarteroni and Valli 1999] by explaining how an aggregate solver for the pressure Poisson equation can be assembled using as subroutines two independent solvers for two non-overlapping partitions of the entire computational domain. After covering the basic theory we will detail how this construction extends to multiple partitions, and derive a preconditioner based on this concept in later sections.

4.1 The two-subdomain case



Consider a domain Ω that has been partitioned into two subdomains Ω_1 and Ω_2 through an interface region Γ. Let us assume we have a finite-difference discretization of the pressure Poisson equation on Ω, and that the interfacial region Γ is thick enough to shield any stencil in Ω_1 from including a point in Ω_2 (and vice-versa). In practice, when using the standard 7-point stencil in a Cartesian discretization, the interface layer Γ can simply be one-node thick as long as it cleanly decouples Ω into two distinct subdomains (although Γ could also be made wider, if desired). For simplicity of notation we will write the Poisson equation as $Ax = b$, with the understanding that the vector x contains the unknown pressure values and b contains the respective divergence values of the velocity field. We then reorder degrees of freedom as:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_\Gamma \end{bmatrix}, \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ b_\Gamma \end{bmatrix} \tag{1}$$

where x_i, b_i correspond to values in Ω_i, for $i \in \{1, 2\}$ (and similarly x_Γ, b_Γ correspond to degrees of freedom in Γ). Under this reordering, the matrix A assumes the following block form:

$$A = \begin{bmatrix} A_{11} & & A_{1\Gamma} \\ & A_{22} & A_{2\Gamma} \\ A_{\Gamma 1} & A_{\Gamma 2} & A_{\Gamma\Gamma} \end{bmatrix} \tag{2}$$

Note that due to symmetry of A, we have $A_{\Gamma 1}^T = A_{1\Gamma}$ and $A_{\Gamma 2}^T = A_{2\Gamma}$ for the off-diagonal blocks. Using this block form of A it is possible to write the following factorization of the *inverse* matrix A^{-1}:

$$A^{-1} = \underbrace{\begin{bmatrix} I & & -A_{11}^{-1}A_{1\Gamma} \\ & I & -A_{22}^{-1}A_{2\Gamma} \\ & & I \end{bmatrix}}_{U} \underbrace{\begin{bmatrix} A_{11}^{-1} & & \\ & A_{22}^{-1} & \\ & & \Sigma^{-1} \end{bmatrix}}_{D} \underbrace{\begin{bmatrix} I & & \\ & I & \\ -A_{\Gamma 1}A_{11}^{-1} & -A_{\Gamma 2}A_{22}^{-1} & I \end{bmatrix}}_{U^T}$$

where

$$\Sigma = A_{\Gamma\Gamma} - A_{\Gamma 1}A_{11}^{-1}A_{1\Gamma} - A_{\Gamma 2}A_{22}^{-1}A_{2\Gamma}$$

is the *Schur complement* of the block $A_{\Gamma\Gamma}$ in equation (2). The validity of this factorization can be verified via a direct substitution into the identity $A \cdot A^{-1} = I$. Finally, since A (and its inverse) is a symmetric positive definite (SPD) matrix, this factorization implies that the Schur complement is also SPD (the matrix D is equal to the symmetric conjugation of the SPD matrix A^{-1} with the matrix U^{-1}; hence its diagonal sub-block Σ^{-1} is symmetric definite, too).

4.2 The multiple subdomain solver

This formulation extends naturally to an arbitrary number of k subdomain partitions $\Omega_1, \ldots, \Omega_k$ separated by an interface set Γ (figure 2 depicts such a partitioning into four subdomains, with the interface Γ highlighted as the magenta-colored separator surface). The corresponding factorization of A^{-1} in this case is given in equation (3). In order to translate this algebraic expression into a solver algorithm, we first re-factor this into the five-matrix product of equation (4), which has every subdomain inverse A_{ii}^{-1} appear only twice (as opposed to three inversions per subdomain, in equation 3). The last algebraic manipulation, as given in equation (5) further avoids the appearance of the subdomain Laplacian A_{ii}, requiring only the inverses of such matrices. In this expression we have also substituted the symbol $M^\dagger \approx M^{-1}$ for *approximate* inverses of A_{ii} and Σ. If the *exact* inverse of these matrices was used, equation (5) becomes identically equal to the five-factor expression of equation (4). We will later engage in such approximations; for now, we may assume that all these inverses are exact.

The Schur complement method effectively solves the equation $Ax = b$ by multiplying the right hand side b with the factorized equivalent of A^{-1} from equation (5). The key observation is that we can apply this multiplication indirectly, without explicitly constructing the matrix in this factorization. We do this as follows:

1. Solve k subproblems: $A_{11}\hat{x}_1 = b_1, \ldots, A_{kk}\hat{x}_k = b_k$.
2. Solve $\Sigma x_\Gamma = b_\Gamma - A_{\Gamma 1}\hat{x}_1 - A_{\Gamma 2}\hat{x}_2 - \ldots - A_{\Gamma k}\hat{x}_k$.
3. Solve the k new subproblems $A_{11}\delta x_1 = -A_{1\Gamma}x_\Gamma, \ldots, A_{kk}\delta x_k = -A_{k\Gamma}x_\Gamma$.
4. Update $x_1 \leftarrow \hat{x}_1 + \delta x_1, \ldots, x_k \leftarrow \hat{x}_k + \delta x_k$.

Observe that steps (1) and (3) require the solution of fully decoupled systems for each subdomain Ω_i, and this can easily be performed in parallel, without any communication. Step (2) requires the solution of a symmetric and positive definite system (with the Schur complement Σ as the coefficient matrix). Traditionally,

$$A^{-1} = \begin{bmatrix} I & & -A_{11}^{-1}A_{1\Gamma} \\ & \ddots & \vdots \\ & I & -A_{kk}^{-1}A_{k\Gamma} \\ & & I \end{bmatrix} \begin{bmatrix} A_{11}^{-1} & & \\ & \ddots & \\ & & A_{kk}^{-1} \\ & & & \Sigma^{-1} \end{bmatrix} \begin{bmatrix} I & & & \\ & \ddots & & \\ & & I & \\ -A_{\Gamma 1}A_{11}^{-1} & \dots & -A_{\Gamma k}A_{kk}^{-1} & I \end{bmatrix} \tag{3}$$

$$= \begin{bmatrix} A_{11}^{-1} & & & \\ & \ddots & & \\ & & A_{kk}^{-1} & \\ & & & I \end{bmatrix} \begin{bmatrix} I & & & -A_{1\Gamma} \\ & \ddots & & \vdots \\ & & I & -A_{k\Gamma} \\ & & & I \end{bmatrix} \begin{bmatrix} A_{11} & & & \\ & \ddots & & \\ & & A_{kk} & \\ & & & \Sigma^{-1} \end{bmatrix} \begin{bmatrix} I & & & \\ & \ddots & & \\ & & I & \\ -A_{\Gamma 1} & \dots & -A_{\Gamma k} & I \end{bmatrix} \begin{bmatrix} A_{11}^{-1} & & & \\ & \ddots & & \\ & & A_{kk}^{-1} & \\ & & & I \end{bmatrix} \tag{4}$$

$$\approx \begin{bmatrix} A_{11}^{\dagger} & & & \\ & \ddots & & \\ & & A_{kk}^{\dagger} & \\ & & & I \end{bmatrix} \begin{bmatrix} I & & & -A_{1\Gamma} \\ & \ddots & & \vdots \\ & & I & -A_{k\Gamma} \\ & & & I \end{bmatrix} \begin{bmatrix} I & & & \\ & \ddots & & \\ & & I & \\ & & & \Sigma^{\dagger} \end{bmatrix} \left\{ I + \begin{bmatrix} & & & \\ & & & \\ & & & \\ -A_{\Gamma 1} & \dots & -A_{\Gamma k} & I \end{bmatrix} \begin{bmatrix} A_{11}^{\dagger} & & & \\ & \ddots & & \\ & & A_{kk}^{\dagger} & \\ & & & I \end{bmatrix} \right\} \tag{5}$$

solvers based on this method attempt to solve this interface system using a preconditioned Krylov subspace method such as Conjugate Gradients. We will deviate from this practice, and use equation (5) instead, to design a preconditioner for the *global* (coupled) system.

Let us examine the structure of Σ and assess the computational cost of solving the system in step (2) directly. For a volumetric domain Ω with N total degrees of freedom, the dimensionality of Γ would be $O(\sqrt{N})$ in 2D, and $O(N^{2/3})$ in 3D. Note, however, that in contrast to the sparse Laplace matrix A, the Schur complement is a dense matrix, thus having $O(N^{4/3})$ entries and requiring at least as much computation to solve. Asymptotically, this would make step (2) above by far the bottleneck of the solver, if Σ was to be explicitly constructed. Furthermore, the construction of the matrix alone would likely require even more computation, as it would need to account for computing the subdomain inverses A_{ii}^{-1}. Using Conjugate Gradients as the solver in step (2) opens up an interesting possibility: the CG algorithm does not need an explicitly constructed matrix Σ, as long as we have a way to compute matrix-vector products Σx_Γ. In turn, this would require computing products of the form $A_{\Gamma i}A_{ii}^{-1}A_{i\Gamma}x_\Gamma$ as efficiently as possible. Although the factors $A_{\Gamma i}$, $A_{i\Gamma}$ are sparse enough to allow efficient multiplication, multiplying with A_{ii}^{-1} (i.e. solving a subdomain Poisson problem) requires at least linear cost relative to the size of the subdomain (assuming a linear-complexity solver, like an extremely well built multigrid scheme, iterated to full convergence). There would be opportunity for parallelization across subdomains, but we would be still confronted with a linear complexity cost for *each* CG iteration, and we would have to rely on constructing an extremely efficient preconditioner to ensure that only a small finite number of iterations would suffice, independent of resolution. This is, in fact, what is done by many variants of the Schur complement method (often referred to as *iterative substructuring*; see Quarteroni and Valli [1999]).

5 A Schur-complement preconditioner

In light of these challenges we propose certain strategic simplifications that would make the Schur complement method yield an approximate solver of the Poisson equation, rather than a strictly accurate one. Our intent would be to use this approximation as a preconditioner for the Conjugate Gradients method, applied to the full-scale Poisson problem. Our last transformation of the factorized form for A^{-1}, captured in equation (5) was precisely intended to facilitate this process. We can easily show that any *nonsingular* approximation $A_{ii}^{\dagger} \approx A_{ii}^{-1}$ of the subdomain inverses, combined with a *symmetric positive definite (SPD)* approximation $\Sigma^{\dagger} \approx \Sigma^{-1}$ will produce, after substitution in equation (5), an SPD matrix approximation to A^{-1}. Thus multiplication with this expression can be used as a preconditioner for the Conjugate Gradients method.

From an implementation standpoint, we map the application of this preconditioner to a heterogeneous platform by assigning the interior degrees of freedom of each subdomain Ω_i to a single GPU or Many-Core accelerator card, while the interface degrees of freedom (Γ) will be the maintained on the CPU. We design the approximate subdomain inverses $A_{ii}^{\dagger}$ so that they can be multiplied with respective vectors exclusively on the GPU, local to the accelerator that owns the subdomain Ω_i. Multiplication with the matrix blocks $A_{\Gamma i}$ will coincide with data transfer from the card that owns Ω_i to the CPU, while multiplication with the transpose $A_{i\Gamma}$ will relay data from the CPU to the respective accelerator in the opposite direction. Multiplication with the approximate inverse of the Schur complement, i.e. $\Sigma^{\dagger}$, will be handled fully on the CPU. The application of this preconditioner is formalized in pseudocode in Algorithm 1.

Algorithm 1 Preconditioner application $\boldsymbol{z} = A^{\dagger}\boldsymbol{r}$, from eqn. (5)

parallel for $i = 1 \dots k$ **do** ▷ on GPU
  Get $r_i \leftarrow$ CPU
  Solve $q_i \leftarrow A_{ii}^{\dagger} r_i$
  Compute $s_\Gamma^{(i)} \leftarrow -A_{\Gamma i} q_i$
  Send $s_\Gamma^{(i)} \rightarrow$ CPU
  Send $q_i \rightarrow$ CPU
end parallel for
Compute $f_\Gamma = r_\Gamma + s_\Gamma^{(1)} + \dots + s_\Gamma^{(k)}$ ▷ on CPU
Solve $z_\Gamma \leftarrow \Sigma^{\dagger} f_\Gamma$
parallel for $i = 1 \dots k$ **do** ▷ on GPU
  Get $q_i \leftarrow$ CPU
  Get $z_\Gamma \leftarrow$ CPU
  Compute $f_i \leftarrow -A_{i\Gamma} z_\Gamma$
  Solve $z_i \leftarrow A_{ii}^{\dagger} f_i$
  Add $z_i += q_i$
  Send $z_i \rightarrow$ CPU
end parallel for

5.1 Multigrid subdomain solver

For approximating $A_{ii}^{\dagger}$ in the formulation described above, we use a simple, voxel-accurate multigrid solver in the spirit of prior works [McAdams et al. 2010; Molemaker et al. 2008], with some embellishments to support sparse domains as discussed in section 7. The multigrid hierarchy is constructed by classifying every grid cell as "interior", "exterior" (to the active domain) or "Dirichlet", and coarsening this classification to voxels of lower resolution grids. Trilinear transfer operators and a damped Jacobi smoother are employed, with an additional smoothing effort in a narrow band around the boundary (3-7 iterations) for each interior smoothing pass.

Figure 5: *Smoke injected from the bottom of a cylinder, and forced through a twisted bundle of cylindrical holes. Color corresponds to vorticity magnitude. Total 1.2B active cells, in a* $1024^2 \times 2048$ *grid.*

6 The interface Schur-complement system

The last remaining piece for generating our preconditioner is the design of the approximation $\Sigma^\dagger$ and the application of its effective numerical solution which is hosted exclusively on the CPU. As previously stated, our objective is to arrive at an algorithm that is sublinear in complexity relative to the size of the overall boundary and yields a good approximation to the exact matrix $\Sigma = A_{\Gamma\Gamma} - \sum_{i=1}^{k} A_{\Gamma i} A_{ii}^{-1} A_{i\Gamma}$. The approximations incurred in $\Sigma^\dagger$ are twofold: (a) instead of solving $\Sigma x_\Gamma = b_\Gamma$ exactly, we will substitute a number of V-cycles of an appropriately designed multigrid scheme, and (b) we modestly modify the matrix used in this multigrid scheme, using adaptivity, to reduce the cost of direct algebra.

6.1 Multigrid solver for the interface

The Schur complement matrix Σ is not only symmetric and positive definite, but it can further be shown that it is an *elliptic* operator, as it is a discretization of the continuous and elliptic *Steklov-Poincaré operator* for the Poisson equation [Smith et al. 1996]. This suggests that a multigrid solver (on the interface variables; separate from the multigrid cycles used to approximate the subdomain inverses) could be applicable. The simplest technique for building the multigrid hierarchy is to use Galerkin coarsening to construct the operator at each resolution level. However, this would require explicitly computing the matrix Σ at the finest level, which is a computationally expensive proposition. We propose a different hierarchy construction, and design a smoother that avoids explicit matrix assembly.

We construct the coarser level operators in the following fashion, $\Sigma^{2h} = A_{\Gamma\Gamma}^{2h} - \sum_{i=1}^{k} A_{\Gamma i}^{2h} (A_{ii}^{2h})^{-1} A_{i\Gamma}^{2h}$, where the entire matrix A^h has first been coarsened down to A^{2h} using trilinear interpolation, and then the individual building blocks are harvested and reassembled for computing Σ^{2h}. While this may appear a plausible choice for the multigrid hierarchy, and indeed our experiments show that this gives good convergence, the intuition behind it comes from the following observation. Suppose we constructed a multigrid hierarchy for the full problem $Ax = b$, where the right hand side b has non-zero entries *only* on the interface Γ, i.e., we are solving the following equation via multigrid:

$$\begin{bmatrix} A_{11} & & A_{1\Gamma} \\ & A_{22} & A_{2\Gamma} \\ A_{\Gamma 1} & A_{\Gamma 2} & A_{\Gamma\Gamma} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_\Gamma \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ b_\Gamma \end{bmatrix} \tag{6}$$

then the solution can be shown to satisfy $x_\Gamma = \Sigma^{-1} b_\Gamma$. Let us further assume that our smoother routine was designed such that it completely eliminated any residual of equations interior to the subdomains, leaving nonzero residuals only on interface degrees of freedom. We can then interpret our proposed multigrid procedure, which operates solely on the interface, as algebraically equivalent to a full-domain multigrid scheme used with a right hand side that is zero anywhere outside the boundary, as in equation (6), combined with a smoother that annihilates residuals on subdomain interiors.

The terms $\sum_{i=1}^{k} A_{\Gamma i} A_{ii}^{-1} A_{i\Gamma}$ in the formula of Σ correspond directly to this concept of subdomain-interior equations being solved exactly, and used to eliminate those degrees of freedom from the dimensionality of Σ. A smoother that can operate on Σ directly, implicitly ensures that all equations in the interior of each subdomain are satisfied at all times. Finally, the restriction and prolongation operators for the interface-based multigrid solver can be inferred from the transfer operators of the global problem. Note that, for the purposes of this section we depart from the cell-centered perspective of grid values, as employed for example in the hierarchy construction by McAdams et al [2010], and switch to viewing unknowns as stored in the nodes of the *dual* of the typical MAC grid used for the Navier-Stokes discretization (i.e. pressure values stored on *nodes* of this new grid). We then coarsen the cells in the typical 8-to-1 fashion, using trilinear interpolation. This ensures that interface degrees of freedom remain fully aligned across levels of the multigrid hierarchy, and that trilinear prolongation of a finer level's interface variables will only need coarse interface variables as input. Although restriction *into* coarse interface values technically touches interior values as well, the residuals of all such interior equations will be zero (since the Schur complement operator assumes those equations fully satisfied). Thus the transfer operators we ultimately use in our cycle are *bilinear interpolation* along the aligned 2D interface surfaces at each level of the hierarchy.

6.2 Smoothing the Schur-complement system

As previously shown, Σ is a symmetric and positive definite matrix. Thus, in principle, damped Jacobi or Gauss-Seidel would have been convergent smoothers. However, since we do not have access to the explicit matrix form of Σ, operations that would be required (for example, the diagonal elements of Σ) are not readily available. Thus, we take a different approach of designing a smoother that can be iterated without an explicit construction of the matrix. Using the definition of Σ, the system $\Sigma x_\Gamma = b_\Gamma$ can be rewritten as:

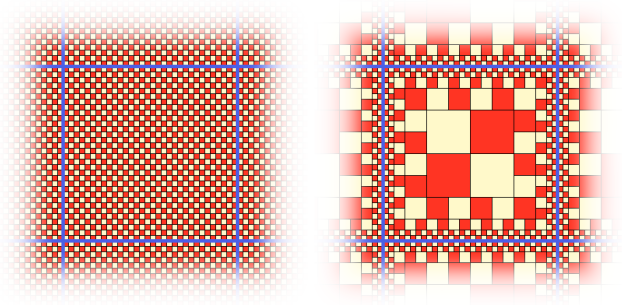
$$A_{\Gamma\Gamma} x_\Gamma = b_\Gamma + \sum_{i=1}^{k} A_{\Gamma i} A_{ii}^{-1} A_{i\Gamma} x_\Gamma \tag{7}$$

We can use equation (7) to design a fixed-point iteration as follows:

$$A_{\Gamma\Gamma} x_\Gamma^{(n+1)} = b_\Gamma + \sum_{i=1}^{k} A_{\Gamma i} A_{ii}^{-1} A_{i\Gamma} x_\Gamma^{(n)} \tag{8}$$

One may recognize the similarity of equation (8) with the analogous matrix form $D x_\Gamma^{(n+1)} = b_\Gamma + (L+U) x_\Gamma^{(n)}$ of the Jacobi iteration based on the decomposition $\Sigma = D - L - U$, should that have been explicitly available. Instead of isolating just the diagonal part of Σ, our decomposition employs the entire $A_{\Gamma\Gamma}$ term. In Appendix A we provide a proof that this iterative scheme will always converge.

The iterative scheme in equation (8) requires two basic blocks. First, given an already computed right hand side, solving for x_Γ requires solving a sparse symmetric and positive definite system. Since the matrix $A_{\Gamma\Gamma}$ is very sparse and structured, we use a sparse Cholesky factorization using the Intel MKL PARDISO library, which we have found to be very well-performing especially



(a) (b)

Figure 6: *Comparison of (a) a uniform discretization of a subdomain interior and (b) our adaptive approximation in Section 6.3.*

due to the fact that the interface is highly structured and admits a very effective nested bisection for reordering its degrees of freedom to maximize sparsity. Second, the right hand side requires the inverse operator A_{ii}^{-1} for each subdomain. Again, our approach would be to use a Cholesky factorization of A_{ii} (with appropriate reordering) to solve the inversion problem using forward/backward substitution. Pseudocode for the smoother routine is given below:

Algorithm 2 Application of smoother routine. Input: $b_\Gamma, x_\Gamma^{(n)}$

1: **for** $i = 1 \dots k$ **do**
2: Compute $y_i \leftarrow A_{i\Gamma} x_\Gamma^{(t)}$ ▷ sparse; fast
3: Solve $z_i \leftarrow A_{ii}^{-1} y_i$ ▷ PARDISO
4: Compute $w_\Gamma^{(i)} \leftarrow A_{\Gamma i} z_i$ ▷ sparse; fast
5: **end for**
6: Compute $f_\Gamma = b_\Gamma + w_\Gamma^{(1)} + \dots + w_\Gamma^{(k)}$
7: Solve $x_\Gamma^{(n+1)} \leftarrow A_{\Gamma\Gamma}^{-1} f_\Gamma$ ▷ PARDISO

The matrix $A_{\Gamma\Gamma}$ in step 7 has $O(N^{2/3})$ nonzero entries, and we observed that with appropriate reordering (using PARDISO) the nonzero entries in the Cholesky factors remain asymptotically well below $O(N)$. The subdomain matrices A_{ii}, however (step 3) contain on the aggregate $O(N)$ nonzero entries, which will yield a strictly superlinear number of nonzero entries in their Cholesky factors, even with excellent reordering. We thus proceed to make one last approximation, in the interest of reducing the dimensionality of these factors, as explained in the following section.

6.3 Adaptive approximation of subdomains

Another opportunity for a dimensionality-saving approximation can be exposed by analyzing the action of the operator $A_{\Gamma i} A_{ii}^{-1} A_{i\Gamma}$ on a vector x_Γ (as used in equation 8). This matrix-vector multiplication can be equivalently interpreted as the following process:

1. The value x_Γ is used as a Dirichlet boundary condition in a *Laplace* problem $A_{ii}\hat{x}_{ii} = A_{i\Gamma}x_\Gamma$, that computes a harmonic interpolant $\hat{x}_{ii}$ of x_Γ in the interior of Ω_i (moving Dirichlet conditions to the right-hand side yields the product $A_{i\Gamma}x_\Gamma$).
2. A global scalar field $\hat{x}$ is assembled by combining the values x_Γ on the interface, with the harmonic interpolants $\hat{x}_{ii}$ from each subdomain.
3. The Laplacian $y = A\hat{x}$ of this interpolated result is computed. Naturally y will be zero in the interior of each subdomain, as $\hat{x}$ was built as a harmonic interpolant in those locations. Nonzero values will result along the interface, however. It can be shown that the restriction y_Γ of y on the interface degrees of freedom is exactly what the Schur complement operator $y_\Gamma = \Sigma x_\Gamma$ computes. The contribution of each subdomain Ω_i to this result is exactly equal to $-A_{\Gamma i}\hat{x}_{ii}$.

Based on this interpretation, we observe that the harmonic interpolant $\hat{x}_{ii}$ in this process could be very well approximated by an *adaptive* tessellation of the subdomain interior, as shown in figure 6. Starting from the uniform grid spanning each subdomain (figure 6a), we aggressively coarsen as we transition to regions farther towards the subdomain interior (figure 6b). All our experiments have indicated that the quality of this approximation is excellent; remember that even if small errors might be observed in the actual interpolants, deep inside the subdomains, only the Laplacian of the resulting interpolant *on the interface* is ultimately relevant.

The performance implications of this approximation are substantial. When adaptively approximated using our aggressive coarsening in figure 6, the actual degrees of freedom of the (octree-type) adaptive subdomain discretization enumerate in the same order of magnitude as the interface variables in $\Gamma \cap \Omega_i$. In practical terms, this adaptive subdomain approximation translates to matrices $A_{i\Gamma}$, A_{ii}, $A_{\Gamma i}$ in Algorithm 2 being replaced by lower dimensionality, adaptive variants $A_{i\Gamma}^*$, A_{ii}^*, and $A_{\Gamma i}^*$. Matrix $A_{i\Gamma}^*$ will is simply constructed from $A_{i\Gamma}$ by removing rows that correspond to interior nodes that have been coarsened away (or have become T-junctions); all such rows would have been full of zeros in $A_{i\Gamma}$, since our coarsening scheme preserves the layer of nodes immediately adjacent to the interface at full resolution (and those are the only interior nodes touched by the stencils of interface equations). Likewise for the transposes $A_{\Gamma i}$, $A_{\Gamma i}^*$ of those matrices. Combined, all adapted interior matrices A_{ii}^* have $O(N^{2/3})$ nonzero entries, and we observed that with proper reordering their Cholesky factors remain clearly sublinear in their aggregate size, allowing us to run step 3 of Algorithm 2 (and the entire smoother) with asymptotic cost safely below the $O(N)$ mark.

7 Implementation details

Construction of adaptive operators We construct the adaptively coarsened discretization of section 6.3 based on a Galerkin process. Let us consider the example of the finest level of the multigrid hierarchy. We define an interpolation operator P_h^* that "prolongates" the adaptive degrees of freedom x^* into their trilinearly interpolated uniform counterparts $x = P_h^* x^*$ (this interpolation is conscious of any T-junctions). Thus, the adaptive discretization of the Laplacian is simply computed as $A_h^* = (P_h^*)^T A_h P_h^*$. In fact, we never explicitly build the uniform matrix A_h, but rewrite this equation as

$$A_h^* = \sum_{a_{ij} \neq 0} a_{ij} \mathbf{p}_i \mathbf{p}_j^T$$

where $a_{ij} = [A_h]_{ij}$, and $\mathbf{p}_k^T$ denotes the k-th row of P_h^*. This formulation allows us to construct the adaptive discretization directly (by iterating over the uniform grid, and processing every spoke a_{ij} of any stencil we encounter), without ever building the uniform matrix. Since our smoother (Algorithm 2) never needs to use the *uniform* subdomain discretization, we construct the adaptive discretizations of coarser levels of the hierarchy A_h^*, A_{2h}^*, A_{4h}^*, etc. by selective (Galerkin) coarsening of the immediately finer *adaptive* discretization, rather than coarsening the corresponding uniform discretization at that level into an octree.

Avoiding nullspace issues Global nullspace components (pockets of fluid with purely Neumann boundary conditions) are handled at the top-level PCG algorithm via projection, as usual. In our smoother subroutine, we generally have the guarantee that the left-hand-side matrix $A_{\Gamma\Gamma}$ will be positive definite (it is always symmetric), if the global matrix A is definite too. However, it is possible for nullspace components to appear in the coarsened version of this matrix $A_{\Gamma\Gamma}^{2h}$, $A_{\Gamma\Gamma}^{4h}$, as a result of the Galerkin procedure, in the vicinity of Neumann domain boundaries. To avoid this, we slightly shift the eigenvalues of every coarsened discretization, say A_{2h}^* by adding a

minute multiple of the identity. The shifted matrix $A^*_{2h}+\epsilon I$ is practically spectrally equivalent to the original, and fully appropriate as a substitute in a multigrid hierarchy. Since we use direct solvers to invert $A_{\Gamma\Gamma}$ in the smoother, conditioning is not an issue. Effectively, this eigenvalue shift will penalize solution components that lie in the nullspace to be effectively equal to zero.

Boosting accuracy We use a high-order defect correction technique [Trottenberg et al. 2001] to allow our approximate inverse of a first-order discretization to be used as a CG preconditioner for a higher order scheme. We structure our top-level PCG solver to implement matrix-vector multiply operations in accordance with a second order accurate discretization of the Laplace operator [Enright et al. 2003]. Upon invocation of the preconditioner, however, we perform the following steps: (i) We execute a few iterations of a Jacobi smoother, using the 2nd order operator, (ii) we then compute the residual, and multiply this with our first-order preconditioner, and finally (iii) we again compute the residual r, write the error equation $Ae = -r$ using the second order operator, which we solve using the same number of Jacobi iterations. We finally add the correction back to the result returned by the preconditioner. This operation, as described, preserves the symmetry and definiteness of the preconditioner, and allows the first order method to be used as an effective preconditioner for the second-order problem (at the comparably minimal expense of some additional smoothing effort near the high-order interface).

Sparse grid storage We use the SPGrid data structure [Setaluri et al. 2014] to store grid data, for all our examples which utilized highly irregular, sparsely populated grids. SPGrid partitions a virtual background grid into rectangular blocks (8^3 voxels each, in our examples), and performs streaming operations and stencil applications by iterating over the sparse collection of active blocks. We used SPGrid directly on CPU and Xeon Phi while on the GPU (that does not offer a virtual memory subsystem as utilized by SPGrid) we translated this representation into a linearized array with explicit pointers to the 26 neighbors of each rectangular block.

8 Incompressible free surface flow

We solve the incompressible Euler equations

$$\vec{u}_t + (\vec{u}\cdot\nabla)\vec{u} + \frac{\nabla p}{\rho} = \vec{f}, \qquad \nabla\cdot\vec{u} = 0$$

using the splitting scheme as described in [Stam 1999]. Here, $\vec{u} = (u, v, w)$ is the velocity field vector, ρ is the fluid density, p is the scalar pressure field, and $\vec{f}$ denotes external forces (such as gravity). We discretize these equations on a MAC grid, where we first explicitly update the advection terms

$$\frac{\vec{u}^\star - \vec{u}^n}{\Delta t} + (\vec{u}\cdot\nabla)\vec{u} = \vec{f}$$

using a semi-Lagrangian scheme [Selle et al. 2008] and then solve for the pressure via a Poisson equation

$$\nabla\cdot\frac{\nabla p}{\rho} = \frac{\nabla\cdot\vec{u}^\star}{\Delta t}$$

in order to update the intermediate velocity as follows

$$\frac{\vec{u}^{n+1} - \vec{u}^\star}{\Delta t} + \frac{\nabla p}{\rho} = 0$$

For tracking the free surface, we generally follow [Enright et al. 2002] using the level set advection of [Enright et al. 2005], the reinitialization scheme of [Losasso et al. 2005], velocity extrapolation method of [Adalsteinsson and Sethian 1999], and a second order accurate pressure discretization of [Enright et al. 2003].

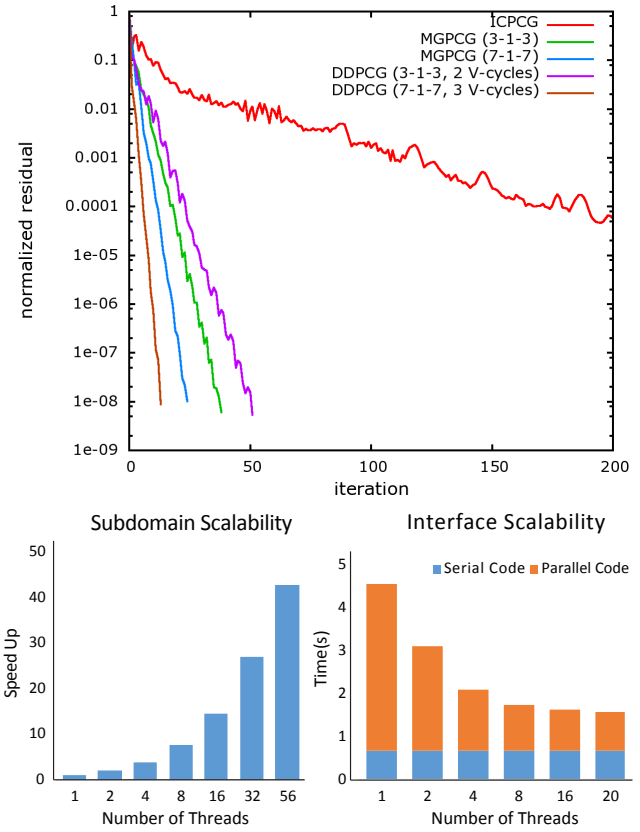


Figure 7: *(Top) convergence profiles for ICPCG, MCPCG and DDPCG (our method) from the water simulation of Figure 3. The numbers (B-I-B) indicate multigrid smoothers that perform B boundary sweeps, then I interior sweeps, and B more boundary sweeps at the end. V-Cycle counts in DDPCG reflect the number of cycles used in the approximate solution of each subdomain. (Bottom) Scaling of subdomain solvers (left; on Xeon Phi) and interface solver (right; on CPU) relative to the active cores/threads utilized.*

9 Examples and performance benchmarks

We demonstrate the effectiveness of our preconditioner through several examples. Figure 1 illustrates two smoke simulations with more than one billion of active degrees of freedom, each. Figure 4 shows a network of interconnected vessels where smoke enters from the lower left corner and exists from the upper right corner. Our solver is able to capture the correct incompressible behavior in relatively few iterations with four subdomains. Figure 8 shows an example where water is poured in a pool with multiple immersed objects, creating complex Neumann interfaces. Figure 3 shows an example where water flows in a channel with multiple interior walls, which cause the flow to meander around them. Figure 9 provides a breakdown of individual kernels of our Schur Complement solver for all these examples, along with timings for alternative solvers, detailed in the following section. We note that no vorticity confinement was used in our smoke examples. Finally, in the interest of efficiency we used as high of a CFL number as our examples could tolerate – sometimes leading to minor loss of detail.

10 Discussion

Evaluation of convergence and scaling In our benchmarks, we compared the convergence behavior of our Schur-Complement Domain Decomposition preconditioned CG *("DDPCG")* with a stan-

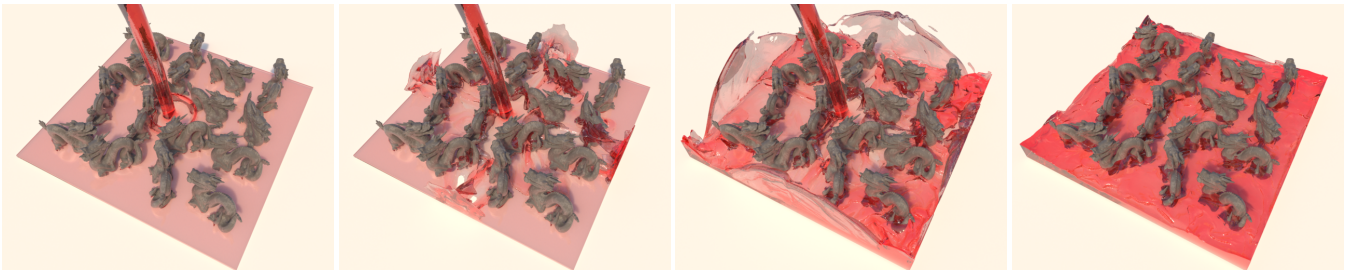


Figure 8: *Water poured in a pool with multiple immersed objects. Second figure from the right shows* 70*M active cells, in a* $1024^2 \times 512$ *grid.*

dard Incomplete Cholesky preconditioner *("ICPCG")* [Foster and Fedkiw 2001], and a standard Multigrid-Preconditioned CG algorithm *("MGPCG")* [McAdams et al. 2010]. For the multigrid option, specifically, we note that although we did not experiment with improved CPU-based versions of MGPCG that take extra steps to better capture the topology of the domain on coarser levels of the multigrid hierarchy [Ferstl et al. 2014], we invested a significant effort to optimize the stock MGPCG to the absolute best of our capacity, both on the CPU as well as on the accelerator cards (using standard domain partitioning practices and CUDA, on the GPU). We produced two, heavily optimized MGPCG implementations: One designed to run exclusively on the CPU, and one designed to run *homogeneously* on just a single GPU, for problems that are small enough to fit entirely in GPU memory. There is only one algorithmic difference between the two implementations: The pure-CPU MGPCG was set up to solve the coarsest level of the multigrid hierarchy using ICPCG – this was done to improve the convergence behavior at the bottom of the multigrid cycle, which was crucial in obtaining acceptable performance at our examples with more than a billion degrees of freedom (without requiring an extremely deep, and occasionally inaccurate V-cycle). The GPU-native implementation of MGPCG used a large number of smoother applications at the bottom of the V-cycle (which was effective for its smaller problem size), to avoid using Incomplete Cholesky on the GPU. We benchmarked the pure-CPU MGPCG solver on the faster (dual socket) of our two test platforms.

In all our examples, the ICPCG solver exhibited dramatically slower convergence performance than both MGPCG, and our proposed DDPCG method, often needing more than an order of magnitude of iterations higher than DDPCG to reach comparable performance. We were unable to use ICPCG for our largest of examples with billions of cells, as the footprint of the explicitly constructed matrices would cause it to run out of memory. For our smallest examples, even each iteration of our heterogeneous DDPCG actually required less time than one CPU-based ICPCG iteration. As a consequence, we did not find ICPCG to be a competitive alternative.

On the other hand, the convergence behavior of MGPCG remained competitive in several of our smaller-size examples. We should point out that the behavior of DDPCG is tunable; investing more V-cycles in the independent subdomain solves, or additional multigrid iterations in the interface solve can boost its convergence efficiency. We found MGPCG to be most competitive with our DDPCG technique in the context of our smaller examples, especially the free-surface water simulations. This is attributed to the prominence of Dirichlet boundary conditions in those scenarios, which dramatically improves the efficacy of smoothing boundary regions, which is essential for multigrid to behave favorably as a preconditioner [McAdams et al. 2010]. On average, across the various frames of the water simulations (Figures 3,8) MGPCG would converge in no more than 1.5x-3x the number of iterations required by our tuned DDPCG, while in the smoke simulation of Figure 4, MGPCG required approximately 2x-2.5x more iterations than DDPCG. In terms of run time, however, the findings paint a quite different picture. The smaller two of our examples (Figures 4, 8) were compact enough to fit on just a single GPU card, where a single iteration of MGPCG was between 5x-8x times faster than a DDPCG iteration. Thus, in spite of the moderately slower convergence of MGPCG, its faster per-iteration cost on the homogeneous, single-GPU implementation makes it preferable to DDPCG by a factor of 3x-5x. Incidentally, the geometry of the smoke example in Figure 4 led to another interesting observation: Although the narrow cylindrical connectors between the glass spheres allowed for a small interface between subdomains used in our solver (and a reduction in CPU computation cost), the same geometry traits increased the approximation error induced by our adaptive coarsening of the subdomain interiors, increasing the required iterations for PCG convergence.

The situation is dramatically different for our larger simulation examples, which cannot be solved with MGPCG on a single accelerator card. For those examples, the only practical alternative was to run MGPCG homogeneously on the CPU. In this context, we observed that each iteration of our DDPCG method was within 20% of the cost of a CPU-only MGPCG iteration. However, for the large-scale examples, dominated by Neumann boundary conditions, we observed MGPCG requiring up to 5x more iterations for convergence, leading to a 3.5x-4.5x performance benefit of DDPCG versus the CPU-only MGPCG. We conclude that for small problem sizes, in the order of 100M degrees of freedom or less, a *homogeneous GPU* implementation of MGPCG is the preferred solver, provided that the problem can fully fit in GPU memory. For problem sizes that do not fit completely in GPU memory, DDPCG appears to be consistently superior to CPU-based MGPCG, with the performance gap becoming larger as the resolution increases.

Limitations and future work The most fundamental limitation of our proposed method is that, in order for its scaling benefits to take effect, it needs to be applied to a problem of adequately large size. In designing our (GPU-hosted) multigrid cycle for the solution of the subdomain problems, we made a conscious choice to keep the design of this solver as simple as possible, approximating the domain at voxel-accuracy at every level, and not enacting any remedies for topological inconsistencies, such as regions merging or small Neumann gaps disappearing after coarsening. This was done in the interest of simplicity, to facilitate low-level optimizations of the solver components. It is quite likely that topology-conscious coarsening schemes [Ferstl et al. 2014] could further improve the convergence properties of this component, and the balance between enacting such improvements and retaining opportunities for aggressive optimization certainly merits investigation. Finally, one should not discount the software engineering challenges that are still associated with developing numerical software that is as inherently heterogeneous as our solver. The established programming paradigms that are available for *homogeneous* thread-based parallel development (e.g. OpenMP) are arguably much more accessible to the non-expert developer. Given the precedent of CUDA, and the growing presence of heterogeneity in modern systems, we hope that programming abstractions for these platforms will continue to evolve.

The scope of our work was specifically restricted to the design and optimization of the pressure Poisson solver on a heterogeneous computer. We also specifically targeted fluid simulation on *uniform* grids in the development of our solver. Although extending the concepts of our solver to an adaptive discretization is certainly possible from an algebraic perspective, we feel that a careful investigation is warranted to make sure that the complexity of work that needs to happen on the interface region remains comparatively lower. This aspect, as well as practices for efficient dynamic partitioning of temporally changing adaptive grids would be an exciting topic for continued investigation.

A very interesting venue for future work might focus on extending our technique to deeper hierarchies of heterogeneous platforms, using for example clusters of network-interconnected GPU-accelerated nodes. The same way that we used a multigrid cycle to approximate a subdomain solver, one could envision using our entire preconditioner as the approximate solver for a subdomain assigned to each cluster node, which is internally subdivided to use GPU accelerations as we currently do. Although bandwidths of network interconnects would be even slower than that of PCIe, due to economy of scale the relevant asymptotics (relative size of interfaces vs. entire grid) could remain favorable. Finally, emerging GPU architectures and technologies (stacked memory, integration of CPU and GPU) might facilitate programming in a homogeneous model (using capabilities such as unified memory spaces), but non-homogeneity in memory bandwidth is almost certain to persist in some form (cores having significantly higher bandwidth to their "local" region of memory). We feel that the adaptation of solver concepts to such architectural traits is an exciting research thread.

Acknowledgments

The authors are grateful to Mark Hill, David Wood and Michael Swift for many insightful early discussions, and Tim Czerwonka for systems setup and support. This work was supported in part by NSF grants IIS-1253598, CCF-1533885, CCF-1423064, IIS-1407282. The bronchi model was sourced via sketchfab.com (published by user thuntu and distributed under the Creative Commons license).

References

ADALSTEINSSON, D., AND SETHIAN, J. 1999. The fast construction of extension velocities in level set methods. *Journal of Computational Physics 148*, 1, 2 – 22.

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.

AMENT, M., KNITTEL, G., WEISKOPF, D., AND STRASSER, W. 2010. A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-gpu platform. In *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, 583–592.

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.

ANDO, R., THUEREY, N., AND WOJTAN, C. 2015. A stream function solver for liquid simulations. *ACM Trans. Graph. 34*, 4, 53:1–53:9.

ANDO, R., THÜREY, N., AND WOJTAN, C. 2015. A dimension-reduced pressure solver for liquid simulations. *EUROGRAPHICS 2015*.

BAILEY, D., BIDDLE, H., AVRAMOUSSIS, N., AND WARNER, M. 2015. Distributing liquids using OpenVDB. In *ACM SIGGRAPH 2015 Talks*, SIGGRAPH '15.

BENDER, J., AND KOSCHIER, D. 2015. Divergence-free smoothed particle hydrodynamics. SCA '15, 147–155.

BRO-NIELSEN, M., AND COTIN, S. 1996. Real-time volumetric deformable models for surgery simulation using finite elements and condensation. In *Computer Graphics Forum*, 57–66.

CHEN, Z., KIM, B., ITO, D., AND WANG, H. 2015. Wetbrush: Gpu-based 3d painting simulation at the bristle level. *ACM Trans. Graph. 34*, 6, 200:1–200:11.

CHENTANEZ, N., AND MÜLLER, M. 2011. Real-time Eulerian water simulation using a restricted tall cell grid. 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.

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.

DA, F., BATTY, C., WOJTAN, C., AND GRINSPUN, E. 2015. Double bubbles sans toil and trouble: Discrete circulation-preserving vortex sheets for soap films and foams. *ACM Trans. Graph. 34*, 4, 149:1–149:9.

DE GOES, F., WALLEZ, C., HUANG, J., PAVLOV, D., AND DESBRUN, M. 2015. Power particles: An incompressible fluid solver based on power diagrams. *ACM Trans. Graph. 34*, 4, 50:1–50:11.

DICK, C., GEORGII, J., AND WESTERMANN, R. 2011. A real-time multigrid finite hexahedra method for elasticity simulation using CUDA. *Simulation Modelling Practice and Theory 19*, 2, 801 – 816.

DICK, C., ROGOWSKY, M., AND WESTERMANN, R. 2016. Solving the fluid pressure poisson equation using multigrid – evaluation and improvements. *IEEE Trans. Visualization & Computer Graphics* (to appear).

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.

EDWARDS, E., AND BRIDSON, R. 2015. The discretely-discontinuous Galerkin coarse grid for domain decomposition. *CoRR abs/1504.00907*.

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

ENRIGHT, D., MARSCHNER, S., AND FEDKIW, R. 2002. Animation and rendering of complex water surfaces. *ACM Trans. Graph. 21*, 3, 736–744.

ENRIGHT, D., NGUYEN, D., GIBOU, F., AND FEDKIW, R. 2003. Using the particle level set method and a second order accurate pressure boundary condition for free surface flows. In *Proc. 4th ASME-JSME Joint Fluids Eng. Conf.*

ENRIGHT, D., LOSASSO, F., AND FEDKIW, R. 2005. A fast and accurate semi-Lagrangian particle level set method. *Comput. Struct. 83*, 6-7, 479–490.

	Water (Fig. 3)		Water (Fig. 9)		Smoke (Fig. 4)		Smoke (Fig. 1; right)	Smoke (Fig. 1; left)
	Active DOFs: 114M		Active DOFs: 70M		Active DOFs: 42M		1.2G DOFs	1.8G DOFs
	GPU	Phi	GPU	Phi	GPU	Phi	GPU	Phi
Number of subdomains	16		16		4		16	40
Subdomain resolution	128 x 128 x 768		128 x 128 x 768		512 x 512 x 512		512 x 512 x 512	1536 x 1536 x 1920
DDPCG total solve time	51.894	44.532	61.446	42.882	38.472	25.704	1790.11	4247.98
DDPCG iteration cost	5.766	4.948	4.389	3.063	1.603	1.071	100.49	193.09
Preconditioner Application	4.291	4.314	3.657	2.769	0.999	0.843	88.9341	183.9
Subdomain Solve (3 V-cycles)	1.362	1.249	1.046	0.865	0.457	0.351	26.4284	152.79
Each MG V-cycle	0.378	0.311	0.311	0.218	0.115	0.079	2.68	21.83
Data transfer from/to CPU	0.158	0.309	0.084	0.169	0.103	0.105	7.6252	16.002
Interface Solve (1 V-cycle)	1.392	1.539	1.214	1.096	0.029	0.032	33.042	14.3918
Smoothing (Top-Level)	0.413	0.349	0.314	0.215	0.006	0.003	7.864	3.741
Restriction/Prolongation	0.008	0.012	0.008	0.007	0.00013	0.0002	0.145	0.094
ICPCG iteration cost (CPU only)		4.16		2.82		1.32	N/A	N/A
MGPCG iteration cost (GPU only)	0.424		0.316		0.1736		N/A	N/A
MGPCG solve time (GPU only)	10.1838		17.679		7.983		N/A	N/A
MGPCG iteration cost (CPU only)		5.55		3.135		4.489	67.28	175.4
MGPCG solve time (CPU only)		127.65		153.5933		218.64	3902.8062	16742

Figure 9: *Timing information for four examples.* ***All run times cited are in seconds****. Our "GPU" platform is an Intel Xeon E5-1650v3 CPU equipped with two NVidia GTX Titan X GPUs and 128GB RAM, while our "Phi" platform is an Intel Xeon E5-2650v3 CPU equipped with six Intel Xeon Phi 31S1P cards. The row labeled* Preconditioner Application *reflects the preconditioning cost for a single PCG iteration.*

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.

FOSTER, N., AND FEDKIW, R. 2001. Practical animation of liquids. In *Proc. of ACM SIGGRAPH 2001*, 23–30.

FOSTER, N., AND METAXAS, D. 1996. Realistic animation of liquids. *Graph. Models Image Process. 58*, 5, 471–483.

FROEMLING, E., GOKTEKIN, T., AND PEACHEY, D. 2007. Simulating whitewater rapids in Ratatouille. In *ACM SIGGRAPH 2007 Sketches*, SIGGRAPH '07.

GAO, M., MITCHELL, N., AND SIFAKIS, E. 2014. Steklov-poincaré skinning. SCA '14, 139–148.

GEIGER, W., LEO, M., RASMUSSEN, N., LOSASSO, F., AND FEDKIW, R. 2006. So real it'll make you wet. In *ACM SIGGRAPH 2006 Sketches*, SIGGRAPH '06.

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.

HECHT, F., LEE, Y. J., SHEWCHUK, J. R., AND O'BRIEN, J. F. 2012. Updated sparse cholesky factors for corotational elastodynamics. *ACM Trans. Graph. 31*, 5, 123:1–123:13.

HORVATH, C., AND GEIGER, W. 2009. Directable, high-resolution simulation of fire on the gpu. In *ACM SIGGRAPH 2009 Papers*, ACM, New York, NY, USA, SIGGRAPH '09, 41:1–41:8.

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.

IHMSEN, M., CORNELIS, J., SOLENTHALER, B., HORVATH, C., AND TESCHNER, M. 2014. Implicit incompressible SPH. *IEEE Transactions on Visualization and Computer Graphics 20*, 3, 426–435.

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 '06, 805–811.

JIANG, C., SCHROEDER, C., SELLE, A., TERAN, J., AND STOMAKHIN, A. 2015. The affine particle-in-cell method. *ACM Trans. Graph. 34*, 4, 51:1–51:10.

JUNG, H.-R., KIM, S.-T., NOH, J., AND HONG, J.-M. 2013. A heterogeneous CPU-GPU parallel approach to a multigrid poisson solver for incompressible fluid simulation. *Computer Animation and Virtual Worlds 24*, 3-4, 185–193.

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

LADICKÝ, L., JEONG, S., SOLENTHALER, B., POLLEFEYS, M., AND GROSS, M. 2015. Data-driven fluid simulations using regression forests. *ACM Trans. Graph. 34*, 6, 199:1–199:9.

LENTINE, M., ZHENG, W., AND FEDKIW, R. 2010. A novel algorithm for incompressible flow using only a coarse grid projection. *ACM Trans. Graph. 29*, 4, 114:1–114:9.

LIU, B., MASON, G., HODGSON, J., TONG, Y., AND DESBRUN, M. 2015. Model-reduced variational fluid simulation. *ACM Trans. Graph. 34*, 6, 244:1–244:12.

LONG, B., AND REINHARD, E. 2009. Real-time fluid simulation using discrete sine/cosine transforms. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, 99–106.

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.

LOSASSO, F., TALTON, J., KWATRA, N., AND FEDKIW, R. 2008. Two-way coupled SPH and particle level set fluid simulation. *IEEE Transactions on Visualization and Computer Graphics 14*, 4, 797–804.

MACKLIN, M., AND MÜLLER, M. 2013. Position based fluids. *ACM Trans. Graph. 32*, 4, 104:1–104:12.

MACKLIN, M., MÜLLER, M., CHENTANEZ, N., AND KIM, T.-Y. 2014. Unified particle physics for real-time applications. *ACM Trans. Graph. 33*, 4, 153:1–153:12.

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

MOLEMAKER, J., COHEN, J. M., PATEL, S., AND NOH, J. 2008. Low viscosity flow simulations for animation. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, Eurographics Association, Aire-la-Ville, Switzerland, SCA '08, 9–18.

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

QUARTERONI, A., AND VALLI, A. 1999. *Domain decomposition methods for partial differential equations*, vol. 10. Clarendon Press.

RASMUSSEN, N., NGUYEN, D. Q., GEIGER, W., AND FEDKIW, R. 2003. Smoke simulation for large scale phenomena. *ACM Trans. Graph. 22*, 3, 703–707.

RAVEENDRAN, K., WOJTAN, C., AND TURK, G. 2011. Hybrid smoothed particle hydrodynamics. SCA '11, 33–42.

RAVEENDRAN, K., THUEREY, N., WOJTAN, C., AND TURK, G. 2012. Controlling liquids using meshes. SCA '12, 255–264.

SELLE, A., FEDKIW, R., KIM, B., LIU, Y., AND ROSSIGNAC, J. 2008. An unconditionally stable MacCormack method. *J. Sci. Comput. 35*, 2-3 (June), 350–371.

SETALURI, R., AANJANEYA, M., BAUER, S., AND SIFAKIS, E. 2014. SPGrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Trans. Graph. 33*, 6, 205:1–205:12.

SMITH, B. F., BJØRSTAD, P. E., AND GROPP, W. D. 1996. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press.

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.

STAM, J. 2002. A simple fluid solver based on the FFT. *J. Graph. Tools 6*, 2, 43–52.

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

TENG, Y., MEYER, M., DEROSE, T., AND KIM, T. 2015. Subspace condensation: Full space adaptivity for subspace deformations. *ACM Trans. Graph. 34*, 4, 76:1–76:9.

THÜREY, N., WOJTAN, C., GROSS, M., AND TURK, G. 2010. A multiscale approach to mesh-based surface tension flows. *ACM Trans. Graph. 29*, 4, 48:1–48:10.

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

VAN OPSTAL, B., JANIN, L., MUSETH, K., AND ALDÉN, M. 2014. Large scale simulation and surfacing of water and ice effects in Dragons 2. In *ACM SIGGRAPH 2014 Talks*, SIGGRAPH '14.

WOJTAN, C., THÜREY, N., GROSS, M., AND TURK, G. 2010. Physics-inspired topology changes for thin fluid features. *ACM Trans. Graph. 29*, 4, 1–8.

WU, X., MUKHERJEE, R., AND WANG, H. 2015. A unified approach for subspace simulation of deformable bodies in multiple domains. *ACM Trans. Graph. 34*, 6, 241:1–241:9.

WU, J., DICK, C., AND WESTERMANN, R. 2016. A system for high-resolution topology optimization. *IEEE Transactions on Visualization and Computer Graphics 22*, 3, 1195–1208.

ZHANG, X., AND BRIDSON, R. 2014. A PPPM fast summation method for fluids and beyond. *ACM Trans. Graph. 33*, 6, 206:1–206:11.

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

ZHU, B., YANG, X., AND FAN, Y. 2010. Creating and Preserving Vortical Details in SPH Fluid. *Computer Graphics Forum*.

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.

ZHU, B., QUIGLEY, E., CONG, M., SOLOMON, J., AND FEDKIW, R. 2014. Codimensional surface tension flow on simplicial complexes. *ACM Trans. Graph. 33*, 4, 111:1–111:11.

A Appendix: Proof of smoother convergence

Consider the fixed-point iteration

$$\begin{aligned} A_{\Gamma\Gamma}\boldsymbol{x}_{\boldsymbol{\Gamma}}^{(n+1)} &= \boldsymbol{b}_{\boldsymbol{\Gamma}} + \textstyle\sum_{i=1}^{k} A_{\Gamma i}A_{ii}^{-1}A_{i\Gamma}\boldsymbol{x}_{\boldsymbol{\Gamma}}^{(n)} \\ \Rightarrow S_1\boldsymbol{x}_{\boldsymbol{\Gamma}}^{(n+1)} &= \boldsymbol{b}_{\boldsymbol{\Gamma}} + S_2\boldsymbol{x}_{\boldsymbol{\Gamma}}^{(n)} \end{aligned} \tag{9}$$

where $S_1 = A_{\Gamma\Gamma}$ and $S_2 = \sum_{i=1}^{k} A_{\Gamma i}A_{ii}^{-1}A_{i\Gamma}$. Let $\boldsymbol{x}_\star$ be the exact solution of this iterative scheme, i.e., $\boldsymbol{x}_\star$ satisfies the equation

$$S_1\boldsymbol{x}_\star = \boldsymbol{b}_{\boldsymbol{\Gamma}} + S_2\boldsymbol{x}_\star \tag{10}$$

Subtracting equation (10) from equation (9) gives

$$S_1\boldsymbol{e}^{(n+1)} = S_2\boldsymbol{e}^{(n)} \Rightarrow \boldsymbol{e}^{(n+1)} = S_1^{-1}S_2\boldsymbol{e}^{(n)}$$

where $\boldsymbol{e}^{(n)} = \boldsymbol{x}_{\boldsymbol{\Gamma}}^{(n)} - \boldsymbol{x}_\star$ is the error in the n-th iteration. In order to show convergence of the iterative scheme in equation (9), we need to show that the spectral radius of $S_1^{-1}S_2$ is less than 1. Now,

$$\Sigma = S_1 - S_2 \Rightarrow S_1 = \Sigma + S_2$$

Since S_2 is symmetric and positive definite (SPD), $S_2^{1/2}$ is well-defined. Noting that the two matrices $S_1^{-1}S_2$ and $S_2^{1/2}S_1^{-1}S_2^{1/2}$ are related by a similarity transform (via $S_2^{1/2}$), it follows that

$$\begin{aligned} \rho(S_1^{-1}S_2) &= \rho(S_2^{1/2}S_1^{-1}S_2^{1/2}) = \rho\left[(S_2^{-1/2}S_1S_2^{-1/2})^{-1}\right] \\ &= \rho\left[(S_2^{-1/2}(\Sigma + S_2)S_2^{-1/2})^{-1}\right] = \rho\left[(I + S_2^{-1/2}\Sigma S_2^{-1/2})^{-1}\right] \\ &= \frac{1}{\lambda_{\min}\left(I + S_2^{-1/2}\Sigma S_2^{-1/2}\right)} = \frac{1}{1 + \lambda_{\min}\left(S_2^{-1/2}\Sigma S_2^{-1/2}\right)} \end{aligned}$$

Since Σ is SPD and $S_2^{-1/2}$ is symmetric, the matrix $S_2^{-1/2}\Sigma S_2^{-1/2}$ is SPD, so $\lambda_{\min}\left(S_2^{-1/2}\Sigma S_2^{-1/2}\right) > 0$. Thus, $\rho(S_1^{-1}S_2) < 1$.