The Role of Matrix Structure on PDE Solver Performance

Joseph McLaughlin

December 14, 2025

Abstract

Sparse linear systems arise from physical simulations and have structure that, in principle makes them efficient to solve, but in practice this structure is difficult to exploit on modern parallel hardware. Properties, such as sparsity, that make these systems mathematically tractable do not inherently make them computationally efficient. Modern hardware depends on predictable memory access and balanced parallel work, while sparse systems often involve irregular data movement and uneven workloads. Several methods confront the same underlying challenge of structuring irregular sparsity in a manner that can be efficiently computed. Such methods reorder, eliminate, or approximate elements of sparse systems to better align with the behavior of hardware. In this review, we detail how these methods expose data locality, reuse, and parallelism. We demonstrate why combining methods that exploit different sources of information is necessary for efficient solution at scale and provide guidance for selecting and composing approaches based on available structure and computational constraints. This work is intended to provide a clear introduction to the constraints of sparse linear systems at scale.

1 Introduction

Solving sparse linear systems consumes 70-80% of runtime in production scientific applications running on contemporary high-performance computing systems [41]. On the Frontier exascale supercomputer at Oak Ridge National Laboratory, in applications including plasma accelerator design [32] and climate modeling for atmospheric simulation [63], the linear solver accounts for similarly large portions of computational time. In simulations of Earth's subsurface, this can vary from 40-95% [74]. This bottleneck persists despite exponential growth in computational throughput because sparse operations are typically constrained by memory performance rather than arithmetic throughput.

Data movement now costs orders of magnitude more energy than arithmetic on contemporary devices [39]. These memory performance constraints mean that sparse operations achieve only 1-5% of peak arithmetic throughput on GPUs, largely because irregular memory access patterns prevent effective bandwidth utilization [6, 73]. For example, an Nvidia A100 GPU delivers 19.5 TFLOPS FP64 peak performance but achieves only 150-250 GFLOPS on sparse matrix operations [5]. As problem sizes grow from millions to billions of unknowns and require distribution across thousands of compute nodes, performance depends on minimizing data movement through a hierarchy of progressively slower memory.

Sparse operations are typically *memory bandwidth-bound*, achieving performance that is limited by the slowest level of the memory hierarchy that the algorithm must access. Cache provides TB/s bandwidth but holds only MB; main memory provides hundreds of GB/s but is limited to tens or hundreds of GB; memory on remote nodes on provides tens of GB/s to essentially unlimited capacity. To effectively utilize this structure we consider two algorithmic properties:

- **Memory footprint.** Faster memory often has less capacity, thus reducing memory requirements improves performance. The amount of data an algorithm must keep accessible during execution (*i.e.*, its *memory footprint*) determines which level of the hierarchy it uses. A memory footprint that fits in cache can be quickly accessed, one that spills to main memory less so, and one that exceeds the capacity of the device cannot be processed locally at all. For sparse problems, the size of one's memory footprint is influenced by the problem formulation, preprocessing, and different choices can change memory requirements by orders of magnitude.

- **Communication pattern.** When data access becomes expensive and slow, access it less often by reducing synchronization frequency. Distributed solvers must periodically exchange data across the network. For example, global operations like inner products require coordination across all devices. Similarly, boundary exchanges transfer data between neighboring subdomains. Reducing how often the algorithm must communicate is the primary lever for improving distributed performance.

PDE discretizations create matrices with structural properties absent in arbitrary sparse systems. Each row contains only a few nonzeros corresponding to geometric neighbors on the mesh; a second-order finite difference method in 3D results in 7 nonzeros per row, and finite element methods produce 7-27 depending on element characteristics. This sparsity reflects the locality of differential operators and low-dimensional spatial embeddings.

Algorithms can exploit this structure to close the gap between sparse and dense performance. Dense linear algebra routinely achieves 60-80% of peak hardware throughput, while sparse operations on arbitrary matrices achieve 1-5%. The difference is not inevitable for PDE-derived systems: their structure enables complexity improvements impossible for arbitrary sparse matrices.

Building faster hardware does not automatically enable larger simulations. Exascale machines draw 20+ megawatts, and sparse solvers consume most of that power. Hardware capability does not translate to compute capability without algorithms that exploit problem structure. The question is not whether faster hardware can be built, but whether algorithms can be designed to use it.

Production applications at exascale face both memory and communication constraints simultaneously. Reducing memory footprint enables distribution but leaves iteration count unchanged; reducing iteration count accelerates convergence but does not shrink memory requirements. Because no single method addresses both, production codes compose multiple approaches: domain decomposition distributes memory, reordering reduces per-subdomain footprint, and multigrid accelerates convergence [9].

This work examines how structural properties of PDE-derived systems can be exploited to address memory and communication constraints at billion-unknown scales. We emphasize demonstrated performance on contemporary hardware and the compositional strategies used in production codes.

2 The Memory Hierarchy

Contemporary computing systems organize memory into a hierarchy of progressively larger but slower storage, as shown in Table 1. At one extreme, L1 cache provides nanosecond-scale access to kilobytes of data at TB/s bandwidth. At the other, network-attached memory on remote nodes provides effectively unlimited capacity but at microsecond latencies and tens of GB/s bandwidth per link. Each level of this hierarchy differs from its neighbors by roughly an order of magnitude in both capacity and access speed. The consequence is that algorithm performance depends critically on which level of the hierarchy dominates data access. An algorithm data fits entirely in L3 cache operates at TB/s bandwidth; one that spills to DRAM operates at hundreds of GB/s; one that requires network communication operates at tens of GB/s with microsecond latencies between accesses.

2.1 The Roofline Model

Sparse matrix operations from PDE discretizations are more often *memory-bound* rather than *compute-bound* because sparse operations perform very few floating-point operations per byte accessed from memory. A task is memory-bound when its performance is limited by how fast data can be moved from memory to the processor, not by how fast the processor can perform arithmetic. For example sparse matrix-vector multiplication (SpMV) is memory-bound because it reads approximately 12-16 bytes per nonzero (8 bytes for the matrix value plus 4-8 bytes for indices) while performing only 2 floating-point operations (one multiply, one add), yielding an *arithmetic intensity* of approximately 0.125-0.17 flops/byte.

Low arithmetic intensity imposes a performance ceiling determined by memory bandwidth rather than compute capability. For any processor with memory bandwidth β and peak arithmetic throughput P_{peak}, achievable performance on operations with arithmetic intensity I satisfies

$$P \leq \min(P_{\text{peak}}, \beta \cdot I) \tag{1}$$

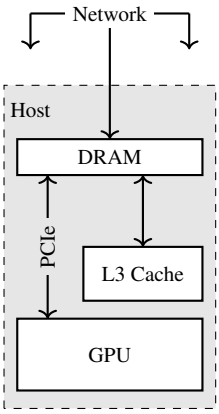
[73]. When I is low, the bandwidth term $\beta \cdot I$ determines the ceiling regardless of arithmetic capability, this is commonly called the *Roofline model*. Sparse matrix-vector multiplication on the Nvidia A100 GPU with 1.6 TB/s HBM bandwidth and $I \approx 0.125$ flops/byte achieves at most 200 GFLOPS, approximately 1% of the 19.5 TFLOPS arithmetic peak. In contrast, dense matrix multiplication achieves arithmetic intensity of 10-100 flops/byte through data reuse, routinely reaching 60-80% of peak throughput.

2.2 Memory Footprint

Each level of the memory hierarchy imposes a capacity limit, and an algorithm that requires more data than fits at one level must access the next level down. An algorithm whose data fits entirely in cache operates at TB/s bandwidth; one that spills to DRAM operates at hundreds of GB/s, a 3-10$\times$ slowdown; one that exceeds device memory cannot execute locally at all. The memory footprint of a sparse solver is the amount of data that must be simultaneously accessible during execution, including the matrix, solution and right-hand side vectors, and any auxiliary data structures the algorithm requires. Storing a sparse matrix requires approximately 12-16 bytes per nonzero (value, indices), so a billion-unknown system with 10

Table 1: Memory and communication hierarchy for contemporary high-performance computing systems with architectural diagram showing node organization. Latencies in nanoseconds enable direct comparison across five orders of magnitude. Bandwidth specifications bidirectional. CPU specifications per dual-socket node; GPU specifications per device. Network specifications from [47]; GPU specifications from [50, 64].

	Read Latency (ns)	Write Latency (ns)	Bandwidth	Capacity
Network (InfiniBand HDR)				
Point-to-point	1,000–2,000	1,000–2,000	25 GB/s	—
Collective (100K cores)	100,000–200,000	100,000–200,000	—	—
Host Memory (Dual-Socket)				
DDR5 DRAM	80–100	15–30*	200–300 GB/s	512 GB
L3 cache (per socket)	10–20	10–20	>1 TB/s	384–768 MB
Host-Device (PCIe x16)				
PCIe 4.0	10,000–20,000	5,000–10,000†	64 GB/s	—
PCIe 5.0	10,000–20,000	5,000–10,000†	128 GB/s	—
GPU Memory				
HBM2e (A100, MI250X)	5–10	5–10	1.6–3.2 TB/s	80–128 GB
HBM3 (H100, MI300X)	5–10	5–10	3.4–5.3 TB/s	80–192 GB



*Posted writes; actual DRAM cycle time 80–100 ns. †Posted writes; round-trip reads.

nonzeros per row requires roughly 120-160 GB for the matrix alone, near the capacity GPU memory.

Algorithms that construct additional data structures during execution increase footprint beyond the input size. LU factorization creates new nonzeros (*fill-in*) at positions that were zero in the original matrix, and for 3D problems these factors can require 100-1000× more storage than the input. A matrix that fits comfortably in device memory may produce factors that exceed cluster memory. The rate of a matrix's fill-in growth depends on *ordering*, but even optimal orderings produce superlinear growth for 3D problems.

2.3 Communication Costs

Network calls sits at the bottom of the memory hierarchy with effectively unlimited capacity but 50-100× lower bandwidth than local memory and 1000× higher latency, so when a problem must be distributed across nodes, every access to non-local data pays these costs.

Network latency creates a floor on communication time that is independent of message size, because sending any message, even a single byte, requires 1-10 microseconds on current interconnects. An algorithm that requires k sequential message exchanges takes at least k microseconds regardless of bandwidth or message content. Global operations involving all p processors require $O(\log p)$ sequential rounds, so at 100,000 processors this means at least 17 round trips totaling 100-200 microseconds minimum. If an algorithm performs such operations repeatedly, latency accumulates and 1000 global operations cost 100-200 milliseconds in latency alone.

2.3.1 Limits of Distributed Communication

Network bandwidth limits the volume of data that can be transferred per unit time, with current interconnects providing 10-25 GB/s per link compared to 200-400 GB/s for DRAM and 1.6-3.2 TB/s for GPU HBM. Moving 1 GB across the network takes 40-100 milliseconds while moving the same data from local HBM takes under 1 millisecond. Large data transfers are bandwidth-limited and small frequent transfers are latency-limited, and most sparse algorithms fall in the latency-limited regime because the data exchanged per operation is small but operations are frequent.

These physical limits bound what any distributed algorithm can achieve. An algorithm requiring k synchronization points across p processors takes at least $O(k \log p)$ microseconds, and an algorithm exchanging V bytes with neighbors takes at least V/β seconds where β is link bandwidth.

The ratio of local computation time to communication time determines whether distributing the problem can improve performance. When local work between network calls is large, communication costs are amortized, but when local work is small, communication dominates. Increasing processor count while holding problem size fixed (*i.e.*, *strong scaling*) eventually reaches a regime where communication time exceeds computation time, and adding processors provides no benefit.

2.4 KEY INSIGHT

Sparse matrix performance is determined by memory throughput, not arithmetic throughput.

Reducing footprint keeps data in faster memory; reducing distributed communication avoids expensive network calls. Both strategies improve performance despite being constrained by hardware.

3 PDEs and Matrix Structure

Discretizing partial differential equations on computational meshes produces sparse linear systems $Ax = b$ where $A \in \mathbb{R}^{n \times n}$ is the system matrix, $x \in \mathbb{R}^n$ is the unknown solution vector, and $b \in \mathbb{R}^n$ is the right-hand side. The matrix A encodes the discrete differential operator and mesh connectivity. Solving for x given A and b is the computational task that dominates runtime in PDE-based applications.

For PDE-derived systems, A is sparse with $\text{nnz}(A) = O(n)$ to $O(n^{4/3})$ nonzero entries depending on spatial dimension and discretization order. This contrasts with dense matrices where $\text{nnz}(A) = n^2$. The sparsity reflects locality in the differential operator, where each unknown couples primarily to geometric neighbors on the mesh.

3.1 Numerical Discretizations

PDE problems are solved numerically by replacing the continuous unknown function with a finite set of values. For example, solving the heat equation on a metal plate might model a continuous field, but the discretization of this problem utilizes an approximation with a discrete number of grid points. The now discretized problem is typically represented as a linear system $Ax = b$ where x contains the unknown temperatures, b encodes boundary conditions and sources, and A encodes how neighboring temperatures influence each other according to the differential operator.

Different discretization methods produce matrices with different structural properties. *Finite difference methods* approximate derivatives using differences of values at neighboring grid points, producing matrices with 7 nonzeros per row for 3D problems and regular structure imposed by a grid. *Finite element methods* approximate solutions as linear combinations of basis functions defined on mesh elements, producing nonzeros with sparsity patterns encoding mesh topology. *Discontinuous Galerkin methods* use basis functions entirely within single elements, producing matrices with dense element-local blocks and sparse inter-element coupling.

3.2 Structural Properties of Sparse Matrices

Sparse matrices from PDE discretizations possess structural properties that allow a variety of algorithms to achieve better complexity than methods for general sparse matrices. Social network graphs and web hyperlink matrices are also sparse, but they lack the properties that make PDE-derived systems tractable [27]. The key distinction is that PDE unknowns correspond to locations in low-dimensional physical space, yielding matrix graphs with geometric embeddings and bounded-degree structure. Figure 1 illustrates three categories of structural information that appear reliably in PDE-derived matrices, each enabling specific algorithmic improvements.

3.2.1 Patterned Sparsity

PDE often describe low-dimensional physical space, and this spatial embedding constrains the matrix sparsity pattern in ways that arbitrary sparse matrices do not share. The graph

$$G(A) = (V, E) \tag{2}$$

has vertices representing unknowns and edges connecting coupled unknowns, and for PDE-derived matrices this graph inherits its structure from the computational mesh. Edges connect nearby spatial locations because *differential operators* couple each point to nearby geometric neighbors. This geometric embedding guarantees the existence of small *vertex separators*, which are vertex subsets whose removal disconnects the graph into independent components. The *planar separator theorem* establishes that graphs from 2D discretizations admit separators of size $O(\sqrt{n})$, while 3D problems have separators of size $O(n^{2/3})$ [43]. These bounds are consequences of the low-dimensional embedding, since a surface of dimension $d-1$ can separate a d-dimensional region and PDE meshes inherit this property.

3.2.2 Block Structure

Finite element discretizations assemble matrices from element-by-element contributions, and this assembly process creates block structure reflecting the connectivity of the mesh of elements. Each element contributes a dense local matrix coupling all degrees of freedom within that element, and the global matrix is the sum of these contributions. Certain discretizations produce particularly clean block structures. For example, discontinuous Galerkin methods have interior degrees of freedom with no direct connection to unknowns in other elements [23]. These interior unknowns appear as dense diagonal blocks in the global matrix, coupled to other elements only through the sparse inter-element terms arising from *numerical fluxes*

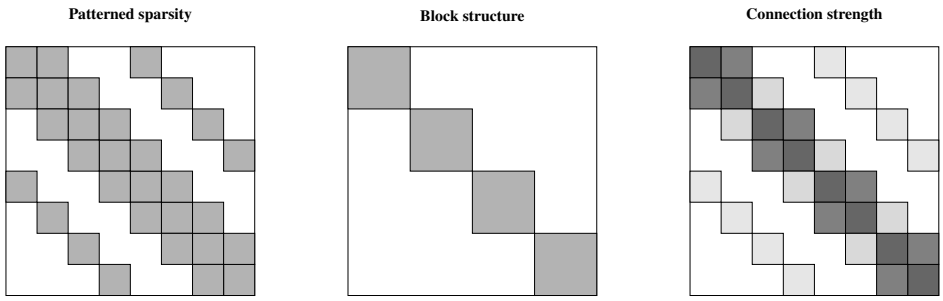


Figure 1: Illustrations of three forms of structure in PDE-derived sparse matrices. **Left:** patterned sparsity emerging from spatial locality. **Center:** block structure shows dense diagonal blocks. **Right:** Connection strength shows varying entry magnitudes, with darker shading indicating great magnitude.

(*i.e.,* coupling interfaces) on element boundaries. The block structure is a consequence of the discretization, not an algorithmic choice.

3.2.3 Connection Strength

The matrix entries themselves encode coupling intensities that reflect physical properties of the underlying PDE, not just the sparsity pattern. The magnitude of an entry a_{ij} indicates how strongly unknown i influences unknown j, and this strength varies across the matrix depending on PDE coefficients and mesh geometry. Anisotropic problems produce matrices with directionally varying connection strengths. Variable-coefficient problems produce spatially varying strengths, with regions of high diffusivity having stronger local coupling than regions of low diffusivity. Stretched meshes produce similar effects, with smaller mesh spacing yielding stronger coupling in the refined direction. *Diagonal dominance*, where $|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|$, indicates that self-coupling exceeds interaction coupling and arises naturally in many elliptic discretizations. This pattern of strong and weak connections is determined by the physical properties of the problem.

3.3 KEY INSIGHT

PDE-derived matrices inherit structure from discretization, such as patterned sparsity, block structure, and predictable connection strengths. These properties enable algorithms to achieve better complexity than methods for unstructure sparsity.

4 Reducing Memory Footprint

Factorizing a *naturally ordered* matrix produces $O(n^2)$ new nonzeros at positions that were zero in the original matrix; the introduction of new matrix entries during factorization is called *fill-in*. The growth of such fill-in will easily exceed the capacity of modern GPU accelerators. For problems with billions of unknowns, this rate of growth yields petabytes of fill-in data.

Problems can be reordered to reduce fill-in, but minimizing this problem is known to be NP-complete for arbitrary sparse matrices [76]; no such polynomial-time algorithm guarantees even a near-optimal solution. PDE-derived sparse matrices circumvent this constraint through a variety of patterned structure including, minimal geometric separators, discontinuous block structure, and algebraic sparsity patterns. These three structural properties have provable spatial complexity bounds under $\mathcal{O}(n^2)$ [36, 43], improving upon storage constraints and easing bandwidth demands. The methods that exploit each type of structure address specific constraints while also creating new bottlenecks, but no method addresses every constraint [33].

4.1 Reordering Methods

Factorization with natural ordering produces $O(n^2)$ fill-in, new nonzeros created at positions that were zero in the original matrix. This scaling makes direct factorization infeasible at modest problem sizes, with factor storage exceeding GPU memory capacity beyond roughly one million unknowns and requiring petabytes for billion-unknown problems.

4.1.1 Fill-in

Eliminating a variable during factorization modifies the sparsity pattern of the remaining system. This process can be understood through the matrix graph, where vertices represent variables and edges connect variables that share a nonzero entry. Eliminating variable k removes it from the graph and adds edges between all pairs of its neighbors; these new edges are fill-in. If the graph has two disconnected components, eliminating variables in one com-

Table 2: Fill-reducing ordering algorithms. Bold entries have production implementations. Results show proven bounds where available, otherwise empirical findings.

Method	Year	Approach	Contribution	Note	Parallel
Minimum Degree [65]	1967	Local greedy	Eliminate lowest-degree vertex first	Superseded by AMD	No
RCM [26]	1969	Local greedy	BFS from peripheral node, reversed	Minimizes bandwidth, not fill	No
Nested Dissection [35]	1973	Separator	Recursive bisection via small separators	$O(n^{1.5})$ factorization (2D)	MPI
AMD [2]	1996	Local greedy	Cheap bounds avoid exact degree computation	5–10× faster than MD, similar fill	No
METIS [40]	1998	Multilevel	Coarsen-partition-refine	Fill within 1.1–1.3× optimal	Both
Hypergraph [19]	2011	Multilevel	Models separators as hyperedge cuts	Avoids explicit A^TA formation	Threaded
spaND [18]	2020	Separator	Compresses dense separator interactions	$O(n)$ factorization (elliptic)	MPI
Reduction + ND [51]	2021	Multilevel	Graph simplification before nested dissection	6× faster than METIS, less fill	Threaded
Fast MD [25]	2021	Local greedy	Improved complexity via amortized analysis	$O(nm)$ ordering time	No
RL-GNN [34]	2021	Learned	Neural network predicts elimination ordering	Fill within 5% of METIS	No
ParAMD [20]	2025	Local greedy	Parallel via independent vertex sets	4–8× speedup over sequential AMD	Threaded
Parth [78]	2025	Reuse	Reuses ordering when sparsity changes locally	Up to 14× ordering speedup	No

ponent cannot create fill-in in the other because there are no paths between them.

4.1.2 Separators & Nested Dissection

A separator is a set of vertices whose removal disconnects the graph into independent components. Figure 2 illustrates this with the center panel shows a graph partitioned into components V_0 and V_1 separated by S, and the right panel shows the reordered matrix with V_0 and V_1 forming independent diagonal blocks while S appears last. If we order variables so that both components are eliminated before the separator, fill-in is confined within each component and cannot propagate between them. Nested dissection [35] applies this idea recursively by find a small separator that bisects the graph, recursively order each component, then order the separator last. For a 2D mesh, each level of recursion bisects the domain along a curve containing $O(\sqrt{n})$ mesh points, and the total fill-in across all levels sums to $O(n \log n)$ rather than $O(n^2)$.

Finding the ordering that minimizes fill-in is NP-complete for general sparse matrices [76]. PDE-derived matrices avoid this hardness because their graphs arise from low-dimensional geometric embeddings that guarantee small separators. The planar separator theorem [43] establishes that 2D meshes admit separators of size $O(\sqrt{n})$; 3D meshes admit separators of size $O(n^{2/3})$. Bisecting an $n^{1/3} \times n^{1/3} \times n^{1/3}$ cube requires cutting through $O(n^{2/3})$ mesh points regardless of cut orientation and eliminating a separator of size $|S|$ creates at least $|S|^2$ fill-in. This occurs because elimination forms a dense clique among all separator variables that were previously connected only through the eliminated subgraph. No additional reordering can improve on the scaling imposed by separator geometry. METIS [40] computes near-optimal orderings through multilevel graph coarsening and refinement, achieving fill-in within 30% of theoretical optimum in $O(n)$ time. Table 2 compares METIS with other ordering algorithms by approach. Recent work extends separator-based ordering to new settings. Graph reduction techniques [51] accelerate separator discovery by shrinking graphs 50-90% through preprocessing rules that provably preserve ordering quality. Parth [78] maintains separator trees incrementally as mesh topology evolves, recomputing only affected subtrees. Machine learning approaches [34] learn separator selection heuristics from families of related meshes, amortizing training cost across repeated solves.

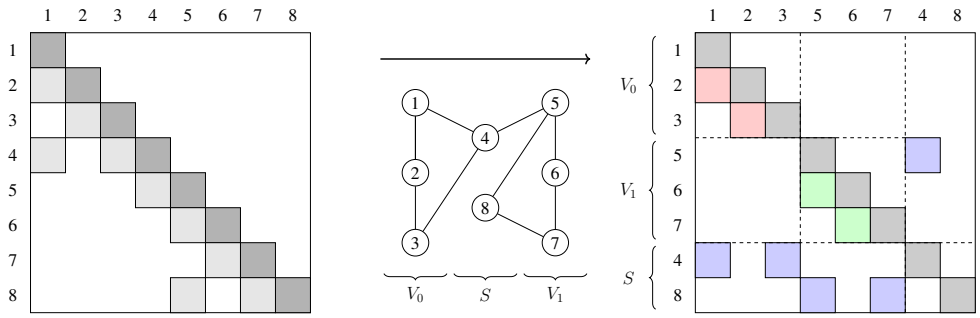


Figure 2: Nested dissection reordering of a sparse matrix. Left: original matrix with tridiagonal-like sparsity pattern. Center: corresponding graph partitioned into subsets. Right: reordered matrix with rows and columns permuted, revealing block structure.

4.1.3 Practical Memory Savings

The improvement from $O(n^2)$ to $O(n^{4/3})$ fill-in typically yields 10-20$\times$ storage reduction for 3D problems. Mixed precision factorization in FP32 with iterative refinement provides an additional $2\times$ reduction [3]. Together, reordering and mixed precision extend the feasible problem size for single-device direct solvers to roughly 50 million unknowns.

Beyond this threshold, factor storage exceeds single-device memory regardless of ordering. Reordering exploits the sparsity pattern to minimize fill-in but cannot change the pattern itself. The $O(n^{4/3})$ bound for 3D problems is imposed by separator geometry, not algorithm quality, and represents a fundamental limit of this approach.

4.2 Static Condensation

Several discretizations produce matrices where interior degrees of freedom couple only within single elements [23, 52]. This structure conforms with the form

$$\begin{bmatrix} A_{II} & A_{I\Gamma} \\ A_{\Gamma I} & A_{\Gamma\Gamma} \end{bmatrix} \begin{bmatrix} x_I \\ x_\Gamma \end{bmatrix} = \begin{bmatrix} b_I \\ b_\Gamma \end{bmatrix}. \tag{3}$$

The resulting local coupling matrix A_{II} is block-diagonal with one block per element; inverting any block requires no information from other blocks. Block Gaussian elimination exploits this independence to eliminate interior variables cheaply, reducing the global system to boundary unknowns [37]. This technique, *static condensation* in the finite element literature (Guyan reduction in structural dynamics) has been applied since the 1960s to reduce problem size before solution [24].

4.2.1 Forming the Schur Complement

To factorize we proceed by partitioning unknowns into interior x_I and boundary x_Γ. Computing

$$x_I = A_{II}^{-1}(b_I - A_{I\Gamma}x_\Gamma) \tag{4}$$

from the first block row and substituting into the second yields the Schur complement system $Sx_\Gamma = g$ where

$$S = A_{\Gamma\Gamma} - A_{\Gamma I}A_{II}^{-1}A_{I\Gamma}. \tag{5}$$

Because A_{II} is block-diagonal (Figure 3), forming A_{II}^{-1} requires only independent dense factorizations of element-sized blocks, costing $O(n_e^3)$ per element for element dimension n_e. These local operations involve dense linear algebra on small matrices, achieving arithmetic intensity far exceeding sparse operations; batched dense kernels achieve 5-8 TFLOPS on Nvidia A100 GPUs [1], so formation cost is small relative to solving the condensed system. After solving for x_Γ, back-substitution recovers interior values through independent element-local solves.

4.2.2 Memory and Performance Implications

The Schur complement system is significantly smaller than the original because boundaries between elements are *lower-dimensional* than the element interiors. In 3D, boundaries are surfaces and interiors are volumes; consequently, boundary unknowns grow as $O(n^{2/3})$ while total unknowns grow as $O(n)$. The original system requires $O(n)$ storage, but the Schur complement requires $O(n^{2/3})$ storage. Higher polynomial degree improve this reduction because interior unknowns per element scale as p^3 while boundary unknowns per face scale as p^2; in practice, reductions range from $8\times$ at $p = 3$ to $15\times$ at $p = 5$ [49]. For problems approaching single-device memory limits, this reduction determines whether the problem fits locally or requires distribution.

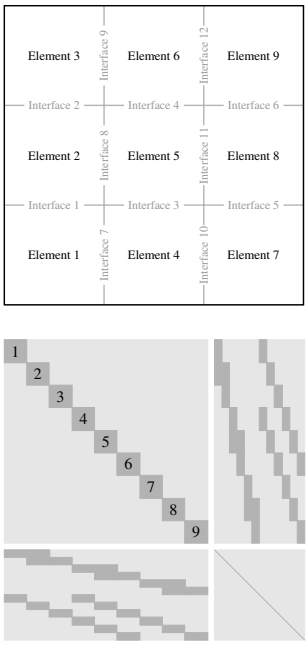


Figure 3: Static condensation using element-wise block structure. **Top:** nine elements coupled through twelve interfaces. **Bottom:** the resulting matrix has block-diagonal interior-interior coupling A_{II} (top-left quadrant), with off-diagonal blocks $A_{I\Gamma}$ and $A_{\Gamma I}$ coupling interiors to interfaces.

Recent work has demonstrated that grouping elements into macro-elements enables tuning local problem size to hardware capacity [8]. Unlike standard HDG where element matrices are small enough for dense factorization, macro-element interiors require sparse solves, but the global interface system shrinks proportionally. This trade-off proves favorable: as increasing local problem size through mesh refinement within macro-elements simultaneously reduces global degrees of freedom and iteration counts. The global solve operates matrix-free, and local solves are embarrassingly parallel.

4.3 Incomplete Factorization

Structural methods reduce memory requirements but do not eliminate the superlinear storage growth of direct factorization. For problems where even reordered factors exceed available memory, iterative methods become necessary. Iterative methods require $O(\text{nnz}(A))$ storage, but without preconditioning they converge in $O(h^{-1})$ iterations for elliptic problems, yielding thousands of iterations on fine meshes [56]. Preconditioning reduces iteration counts to tens or hundreds.

Incomplete LU factorization computes approximate factors $\tilde{L}$ and $\tilde{U}$ by discarding entries during Gaussian elimination [46]. The simplest variant restricts factors to the original sparsity pattern, adding storage equal to $\text{nnz}(A)$. Threshold-based variants allow controlled fill, with typical parameters yielding factors with 5–20$\times$ original matrix nonzeros [54]. Table **??** summarizes variants from classical sequential methods to recent parallel algorithms designed for GPU construction.

4.3.1 In Elliptic Systems

For elliptic problems, incomplete factorization presents a favorable memory-iteration tradeoff. Minimal additional storage often reduces iteration counts from thousands to hundreds; allowing modest fill further reduces iterations to tens [56]. This tradeoff arises because elliptic discretizations produce diagonally dominant matrices where discarded fill-in contributes little to preconditioner effectiveness [46]. The memory cost of 5–20$\times$ original matrix storage compares favorably to exact factorization, which requires 100–1000$\times$ for 3D problems.

This tradeoff relies on diagonal dominance that not all PDE discretizations possess. Constrained systems have zero diagonal entries in constraint rows, and wave operators produce indefinite matrices where approximation error is amplified rather than damped [10, 30]. For these problem classes, incomplete factors provide poor preconditioners regardless of fill level, and the memory invested in denser factors yields no corresponding reduction in iteration count.

4.4 KEY INSIGHT

These methods reduce the memory footprint by exploiting structural properties of PDE-derived matrices, and the bounds they reach are imposed by that structure. Reordering methods limit fill-in propagation during factorization; static exploits block structure to reduce the size of the system; incomplete factorization trades exactness for bounded storage. Such algorithms yield smaller memory footprints, keeping data closer to compute units, in cache rather than DRAM, and on-device rather than distributed across a network.

5 Reducing Distributed Communication

Distributing a problem across multiple processors requires coordination. Processors must exchange data to handle unknowns that couple across partition boundaries, and they must synchronize to compute global quantities like inner products and norms. Iterative solvers pay these coordination costs every iteration, so methods requiring

Table 3: Incomplete factorization methods. Storage column reports factored nonzeros relative to nnz(A).

Method	Year	Storage	Contribution	Parallel
ILU(0) [46]	1977	$1\times$	Original pattern only; no tuning	No
ILU(k) [56]	1980s	2–5$\times$	Level-of-fill controls density	No
ILUT [54]	1994	5–20$\times$	Dual threshold (τ, p)	No
ILUM [55]	1996	5–15$\times$	Multilevel via independent sets	No
ARMS [57]	2002	10–30$\times$	Recursive Schur complements	Distributed (MPI)
ParILU [22]	2015	$1\times$	Fixed-point iteration for factors	Async. (GPU)
ParILUT [4]	2018	5–20$\times$	Parallel pattern adaptation	Async. (GPU)
BILU [13]	2020	5–20$\times$	Block detection; BLAS kernels	Dense blocks (threaded)

1000 iterations pay 1000 times. Reducing communication volume and frequency is essential for distributed performance.

PDE-derived systems possess two structural properties that enable such reduction. Spatial locality confines coupling to geometric neighbors, so partitioning by region yields thin interfaces rather than dense inter-processor coupling. Smooth error structure allows coarse grids to represent fine-grid errors accurately, so hierarchical correction reduces iteration counts from thousands to tens. Domain decomposition exploits spatial locality to reduce bytes transferred; multigrid exploits smooth error to reduce synchronization frequency.

5.1 Communication Patterns

Distributed iterative methods exhibit three characteristic communication patterns, each with different scaling behavior and optimization strategies.

- **Neighbor exchange** moves data between processors sharing partition boundaries. Each processor sends and receives from a fixed number of geometric neighbors, typically 6-26 in three dimensions depending on whether face, edge, and corner neighbors participate. Message size scales with interface area for 3D partitioning of n/p unknowns. Total network traffic scales as $O(p \cdot (n/p)^{2/3}) = O(n^{2/3}p^{1/3})$, growing sublinearly with processor count. For this reason, this pattern ends up bandwidth limited.

- **Global reductions** compute scalar quantities requiring contributions from all processors. Inner products, norms, and convergence checks require collective operations that touch every processor regardless of data size. Latency dominates: on 10^5 processors connected by InfiniBand HDR or Slingshot-11 interconnects, each reduction requires 100-200 μs even for 8-byte payloads [29]. At 1000 iterations with two reductions per iteration, synchronization latency alone accumulates to 200-400 ms.

- **Collective broadcasts** distribute shared data from one or few processors to many. Coarse-grid solutions, global parameters, and convergence information follow this pattern. Cost scales as $O(\log p)$ in well-implemented trees but grows with the size of data being broadcast.

The distinction matters for optimization. Neighbor exchange benefits from reducing interface size and message count; global reductions benefit from reducing iteration count; collective broadcasts benefit from reducing the data being broadcast. Domain decomposition primarily addresses neighbor exchange, multigrid primarily addresses global reductions, and their composition addresses both.

5.2 Domain Decomposition

Partitioning into non-overlapping subdomains

$$\Omega_1, \ldots, \Omega_p \tag{6}$$

that share only interface boundaries transforms the communication pattern from global vector exchanges to local interface exchanges [66]. Ordering unknowns by subdomain interiors followed by interfaces yields block structure where diagonal blocks A_{ii} represent independent subdomain interiors. Eliminating interior variables produces the Schur complement coupling only interface unknowns. The Schur complement is never formed explicitly; applying S to a vector requires independent subdomain solves followed by interface assembly.

5.2.1 Communication Behavior

Subdomain solves of dimension n/p execute with no communication, and interface assembly exchanges only $(n/p)^{2/3}$ values with geometric neighbors. For subdomains of 10^6 unknowns, interfaces contain approximately

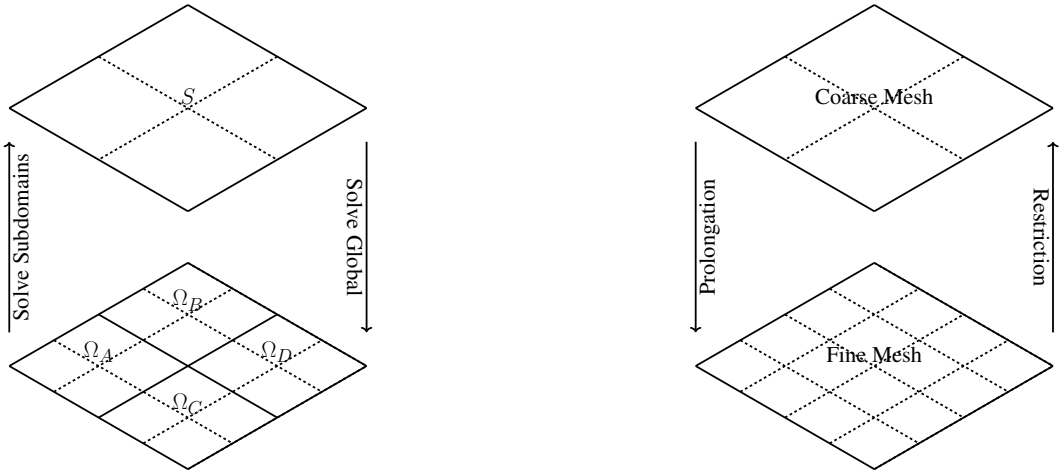


Figure 4: **Left:** Illustration of non-overlapping domain decomposition, partitioning the domain into subdomains $\Omega_A, \Omega_B, \Omega_C, \Omega_D$, each solved independently on separate processors. Eliminating interior unknowns produces the Schur complement S coupling only interface unknowns (dashed lines). **Right:** Illustration of multigrid, with a hierarchy of progressively coarser grids. Restriction transfers residuals from fine to coarse grids; prolongation interpolates corrections from coarse to fine.

10^4 degrees of freedom, requiring 80 KB of data transfer per neighbor in double precision. At 25 GB/s interconnect bandwidth, this transfer completes in 3 μs. A standard iteration on the global system would exchange full vectors of 10^6 values (8 MB) for each sparse matrix-vector product, requiring 320 μs at the same bandwidth. The surface-to-volume ratio of three-dimensional domains, where boundaries scale as $(n/p)^{2/3}$ while interiors scale as n/p, reduces per-iteration communication time by two orders of magnitude.

Larger subdomains yield better ratios of local computation to interface communication. Subdomain factorization with nested dissection ordering requires $O((n/p)^{4/3})$ floating-point operations while interface exchange moves $8(n/p)^{2/3}$ bytes. The resulting $(n/p)^{2/3}$ flops per byte grows with subdomain size, eventually shifting the performance bottleneck from network bandwidth to local arithmetic throughput. Differential operators couple unknowns primarily to spatial neighbors, confining cross-subdomain interaction to thin interface layers; this geometric locality is the structural property that domain decomposition exploits.

5.2.2 Two Level Methods

Without coarse correction, iteration counts grow as $O(p^{1/3})$ in three dimensions because interface information propagates one subdomain diameter per iteration [59]. Two-level methods add a coarse problem of dimension $O(p)$ that propagates global information in a single step. BDDC constructs coarse basis functions from subdomain corners, edges, and faces, achieving low condition number, independent of processor count [28]. FETI-DP enforces continuity at selected interface points through a dual formulation with similar bounds [31]. Both methods yield 10-20 iterations regardless of whether $p = 10^2$ or $p = 10^5$.

At extreme processor counts, the coarse problem of dimension $O(p)$ becomes a bottleneck. With $p = 10^5$ subdomains and 30 coarse degrees of freedom per subdomain, the coarse system has dimension 3×10^6, requiring either redundant solution across processor subgroups or parallel sparse factorization. Three-level methods apply domain decomposition recursively to the coarse problem, reducing coarse dimension from $O(p)$ to $O(p^{2/3})$ at the cost of additional communication levels [68]. Parallel implementations of multilevel BDDC have demonstrated scaling to over 200,000 cores by overlapping coarse and fine computations in time [7].

Adaptive coarse spaces address problems where fixed coarse spaces produce poor conditioning. For heterogeneous coefficients with jumps exceeding 10^6, standard BDDC coarse spaces based on geometric constraints fail to capture modes that propagate slowly across coefficient discontinuities. GenEO (Generalized Eigenvalues in the Overlaps) constructs coarse basis functions by solving local generalized eigenvalue problems in each subdomain, selecting eigenvectors corresponding to eigenvalues below a user-specified threshold [60]. The resulting coarse space adapts to coefficient heterogeneity, maintaining bounded iterations where fixed coarse spaces require hundreds. The cost is additional setup time for local

solves, typically 10-30% of total solution time for problems requiring many solves with the same operator.

5.3 Multigrid

Reducing communication volume does not reduce synchronization frequency; each iteration still requires global reductions for inner products, paying 200 μs latency on 10^5 processors regardless of bytes transferred. Unpreconditioned methods for elliptic problems require $O(\sqrt{\kappa})$ iterations where condition number $\kappa = O(h^{-2})$ grows with mesh refinement [56]. For mesh spacing $h = 10^{-3}$, condition numbers reach 10^6, yielding approximately 10^3 iterations and 200 ms of accumulated synchronization latency. Reducing iterations from 10^3 to 10 would cut synchronization time from 200 ms to 2 ms, and multigrid achieves exactly this reduction by exploiting a property of iterative smoothers on elliptic problems.

Iterative methods (*e.g.*, Jacobi) update each unknown using a weighted average of its neighbors. This local averaging rapidly damps oscillatory error, where neighboring values have opposite signs, because averaging values of opposite sign produces smaller magnitudes. However, the same averaging barely affects smooth error, where neighboring values are similar, because averaging similar values changes little. After a few smoothing iterations on an elliptic problem, oscillatory components have been reduced by factors of 10-100 while smooth components remain nearly unchanged [16]. The remaining error varies gradually across the domain, changing appreciably only over distances of many mesh spacings.

This smooth residual error can be represented accurately on a coarser grid with spacing $2h$ using fewer unknowns, because smooth functions are well-approximated by their values at widely-spaced sample points. Correcting error on the coarse grid and interpolating the correction back to the fine grid eliminates smooth modes that fine-grid smoothing cannot efficiently reduce. Recursive application produces a hierarchy of grids where work decreases geometrically with level.

5.3.1 V-cycles

A single pass through this hierarchy is called a V-cycle. The V-cycle applies smoothing on the fine grid, restricts the residual to the next coarser grid, recurses until reaching the coarsest level, solves the coarsest problem directly, then interpolates corrections back through successively finer levels with additional smoothing at each. The name reflects the shape traced by visiting grids from fine to coarse and back. For elliptic problems, each V-cycle reduces error by a factor $\rho = 0.1$-0.2 independent of problem size, yielding 10^{-10} relative residual in 10-15 V-cycles [16].

5.3.2 Geometric vs Algebraic

Geometric multigrid (GMG) constructs this hierarchy directly from mesh geometry, defining restriction and interpolation operators through known spatial relationships between grid levels. For elliptic problems on regular grids, GMG achieves the convergence factors above, reducing iterations from 1000 to 15 and cutting synchronization time from 200 ms to 3 ms on 10^5 processors. GMG requires explicit mesh hierarchies that adaptive mesh refinement codes and unstructured discretizations often do not maintain.

Algebraic multigrid (AMG) constructs the hierarchy from matrix entries alone, enabling multigrid convergence rates without geometric information [53]. Strong connections, defined by $|a_{ij}| \geq \theta \max_{k \neq i} |a_{ik}|$ for threshold $\theta = 0.25$-0.5, identify variable pairs whose values are tightly coupled. Coarsening algorithms select coarse variables ensuring every fine variable has at least one strongly connected coarse neighbor for accurate interpolation. Coarsening by factor 4-8 per level produces hierarchies of 4-5 levels with total storage 1.3-1.5 times the fine-grid matrix [61]. AMG recovers the $O(1)$ iteration counts of geometric multigrid by inferring from matrix entries the smooth error structure that GMG exploits through explicit geometry.

5.3.3 Communication Behavior

The V-cycle communication pattern favors fine levels where data is distributed and local, with progressively less data moved on coarser levels as grids shrink by factors of 4-8. Fine-grid smoothing and residual computation require only neighbor communication for halo exchange, similar to domain decomposition interface exchange. The coarsest level requires global communication but operates on fewer than 10^6 unknowns after 4-5 levels of coarsening from 10^9 fine-grid unknowns. Since $O(1)$ V-cycles suffice for convergence, total communication volume is dominated by fine-level operations executed 10-20 times rather than 1000 times.

Parallel scalability of AMG is limited by coarse-level communication, where the Galerkin triple product $P^T A P$ creates increasingly dense matrices as the hierarchy is formed. Stencil sizes that start at 7-27 nonzeros per row on the fine grid can grow to thousands on coarse levels, generating high communication volume for coarse-level sparse matrix-vector products. Table 4 summarizes developments since 2014 addressing this bottleneck through three complementary directions: algorithmic restructuring to reduce message count and size, hardware-aware implementations to reduce the cost of each message, and machine learning to automate parameter selection and operator construction. AMG has attracted substantially more research activity than domain decomposition in re-

Table 4: Algebraic multigrid variants and developments. Bold entries have production implementations.

Method	Year	Category	Contribution	Result
Classical AMG [53]	1987	Coarsening	Identifies strongly connected variables for coarsening	Foundation for all AMG variants
Smoothed Aggregation [69]	1996	Interpolation	Groups variables into aggregates, smooths interpolation	Handles elasticity and higher-order PDEs
AMGe [14]	2000	Interpolation	Uses element stiffness matrices for interpolation	Better accuracy for finite element problems
Energy Minimization [71]	2000	Interpolation	Computes interpolation by minimizing energy functional	Provably optimal interpolation weights
α**SA** [15]	2004	Adaptive	Automatically discovers smooth error modes	Self-tuning without user input
Additive AMG [70]	2014	Cycle	Applies all multigrid levels simultaneously	More parallelism, fewer synchronization points
AmgX [48]	2015	GPU	Complete AMG implementation for GPUs	Production solver in NVIDIA ecosystem
Sparsification [11]	2016	Communication	Drops small entries from coarse-grid operators	Up to 50% communication reduction
Node-aware [12]	2020	Communication	Routes messages through faster intra-node paths first	2–4$\times$ speedup at scale
Neural Smoothers [21]	2020	ML	Neural network learns effective smoothing operator	1.5–4$\times$ faster convergence
Mixed-precision [45]	2021	Precision	Uses lower precision on coarse levels	1.5–2$\times$ speedup
RL Coarsening [62]	2021	ML	Reinforcement learning selects coarse variables	Generalizes to larger problems, linear cost
Three-precision [67]	2023	Precision	Assigns precision (16/32/64-bit) per multigrid level	Works across AMD, Intel, NVIDIA GPUs
AmgT [44]	2024	GPU	Exploits GPU tensor cores for sparse operations	1.3–2$\times$ over cuSPARSE
ML Coarse Operators [38]	2024	ML	Learns sparser coarse-grid operators	Lower memory and compute cost
CRAMG [77]	2025	Communication	Combines grid transfer and residual computation	35% fewer messages, 45% less data

cent years; the modularity of its components (coarsening, interpolation, smoothing, cycling) creates natural interfaces for algorithmic innovation, while its purely algebraic formulation makes it amenable to machine learning approaches that would require geometric information in domain decomposition contexts.

5.3.4 Modern AMG Methods

Algorithmic restructuring attacks communication at the protocol level. Sparsification systematically removes small entries from coarse-grid matrices after hierarchy construction, dropping entries that contribute little to convergence and reducing communication volume by up to 50% while maintaining convergence rates [11]. Additive variants replace the sequential V-cycle with parallel application of all levels simultaneously, eliminating the fine-to-coarse-to-fine synchronization pattern [70]. Node-aware communication restructures message routing to aggregate intra-node transfers before inter-node sends, exploiting the order-of-magnitude difference between on-node bandwidth (hundreds of GB/s) and network bandwidth (tens of GB/s) to achieve 2-4$\times$ speedup at scale [12]. CRAMG fuses interpolation and residual computations into combined operations, reducing the number of halo exchanges per V-cycle level from four to two and achieving 35% fewer messages with 45% less data transferred [77]. These techniques compose: node-aware routing reduces inter-node messages, sparsification reduces bytes per message, and fused operations reduce message count.

Hardware-aware implementations reduce the cost of each byte transferred. Mixed-precision AMG stores coarse-level matrices and vectors in single or half precision, halving both memory footprint and bytes communicated; convergence analysis establishes that coarse-grid corrections tolerate lower precision because they need not be computed exactly [45]. Three-precision implementations assign 64-bit, 32-bit, and 16-bit formats to successive hierarchy levels, with production implementations available across AMD, Intel, and NVIDIA GPUs [67]. AmgX provides a complete GPU-accelerated AMG implementation integrated into NVIDIA's solver ecosystem [48], while AmgT reformulates sparse operations as blocked matrix-matrix products suitable for Tensor Cores, achieving 1.3-2$\times$ speedup over standard sparse kernels [44]. The combination of reduced precision and accelerator-optimized kernels addresses both bandwidth (fewer bytes) and latency (faster processing).

Machine learning approaches automate decisions that

traditionally required expert tuning. Reinforcement learning can select coarse variables that yield sparser hierarchies than classical strength-based heuristics, with inference cost scaling linearly and generalizing to problems larger than training examples [62]. Neural networks can learn smoothing operators that converge faster than hand-designed relaxation schemes [21], or approximate interpolation operators with controlled sparsity that reduce coarse-grid communication [38]. Automated parameter selection removes the need for problem-specific tuning [17]. These methods remain active research topics with emerging production deployment, but they demonstrate that data-driven optimization can reduce communication costs beyond what manual algorithm design achieves.

Multigrid convergence degrades for problems lacking smooth error structure. Helmholtz problems at high wavenumber produce oscillatory solutions where error varies at scales comparable to mesh spacing, preventing accurate representation on coarser grids [30]. Convection-dominated flows produce error aligned with flow characteristics rather than varying smoothly in all spatial directions [72]. For these problem classes, coarse-grid correction provides limited benefit regardless of hierarchy construction, and iteration counts grow with problem size.

5.4 Coordinating Both Methods

Spatial locality and smooth error structure address different hardware constraints through different mechanisms. Domain decomposition exploits the fact that differential operators couple unknowns to geometric neighbors, so partitioning by spatial region yields interfaces of dimension $(n/p)^{2/3}$ rather than n/p. For 10^6-unknown subdomains at 25 GB/s bandwidth, interface exchange requires 3 μs compared to 320 μs for full-vector communication, reducing bandwidth demand by two orders of magnitude. Multigrid exploits the fact that elliptic operators produce smooth error representable on coarse grids, reducing iteration counts from $O(\sqrt{\kappa})$ to $O(1)$. For problems with $\kappa = 10^6$ on 10^5 processors, this reduces synchronization time from 200 ms to 2 ms.

5.4.1 Complementary Properties

Applying domain decomposition without multigrid reduces bytes per iteration but leaves iteration count at $O(p^{1/3})$ for one-level methods, reintroducing latency costs as processor count grows. Applying multigrid without domain decomposition reduces iteration count but each V-cycle still requires $O(n)$ data movement across processor boundaries, leaving bandwidth as the bottleneck. Composing both methods bounds interface communication through spatial partitioning while bounding iteration count through hierarchical error correction.

Interface volume scaling as $O(n^{2/3}p^{1/3})$ and coarse dimension scaling as $O(p)$ impose fundamental limits at extreme scale. These bounds reflect the geometry of three-dimensional domains and the necessity of propagating global error information; algorithmic improvements can approach these limits through better constants but cannot change the scaling. Communication reduction enables exascale simulations like this, but does not address memory growth from matrix factorization, which the methods in §4 target. Production solvers compose memory-reduction techniques with communication-reduction techniques, applying reordering and condensation within subdomains while using domain decomposition and multigrid across the distributed system.

5.5 KEY INSIGHT

Distributed methods pay communication costs every iteration, exchanging data across boundaries and synchronizing globally for inner products. Domain decomposition exploits spatial locality and multigrid exploits smooth error structure, both reducing communication through different mechanisms. Improving the communication pattern often requires composing both methods.

6 Composition

The methods examined in prior sections each exploit a single structural property to address a single bottleneck. Nested dissection exploits geometric separators to reduce memory footprint. Domain decomposition exploits spatial locality to distribute memory and expose parallelism. Multigrid exploits smooth error modes to reduce iteration counts and thereby communication frequency. No single method addresses both memory and communication constraints simultaneously, yet production codes at exascale face both.

6.1 A Model Problem

Consider a 3D Poisson problem with 10^9 unknowns from finite difference discretization on a 1000^3 grid. Storing the matrix requires 80 GB in CSR format, which fits on a single GPU with 80 GB HBM2e but leaves insufficient workspace for factorization.

Nested dissection reduces factorization storage by exploiting geometric separators. The $O(n^{2/3})$ separators guaranteed by the 3D mesh embedding yield $10\times$ storage reduction, from approximately 800 GB to 80 GB. However, the reordered factorization still exceeds single-GPU capacity and remains inherently sequential. Conse-

quently, nested dissection addresses memory but not distribution or parallelism.

Domain decomposition distributes memory by exploiting spatial locality. Partitioning into 1,000 subdomains of 10^6 unknowns each reduces per-processor storage to 80 MB, well within GPU capacity. But the global interface system requires iterative solution, and without preconditioning, convergence takes 500–1,000 iterations with MPI synchronization dominating runtime. Thus, domain decomposition addresses distribution but not iteration count.

Multigrid reduces iteration counts by exploiting smooth error modes. For elliptic operators, error after relaxation is geometrically smooth and well-represented on coarser grids, enabling convergence in 10–15 iterations. Yet multigrid hierarchy construction requires the problem to fit in accessible memory, so it fails on the full 10^9-unknown system before iteration begins. Multigrid therefore addresses iteration count but presupposes that memory distribution has already occurred.

6.2 Limitations & Challenging Problem Regimes

Idealized examples only go so far as to demonstrate why composition is necessary. In practice, each method exploits specific structure and fails when that structure is absent or minimal. Several problem characteristics commonly cause any number of these methods to have be significantly less effective. Recognizing these characteristics narrows the viable compositions for a given problem.

6.2.1 Directional Coupling

Some problems have coupling strengths that vary dramatically by direction. In the matrix graph, edges in one direction have weights 10^2–10^6 times larger than edges in perpendicular directions. This creates error patterns that are smooth along the strongly-coupled direction but oscillatory along weakly-coupled directions.

Such problems arise in simulations where physical processes act preferentially along certain axes. Thin computational domains (*e.g.*, aspect ratios of 100:1 or more) create similar directional bias. Flow problems where material moves primarily in one direction exhibit coupling aligned with the flow.

Multigrid struggles on these problems because its coarsening strategy assumes error is smooth in all directions after relaxation. Standard coarsening removes every other grid point in each direction, but when coupling is 10^6 times stronger horizontally than vertically, the coarse grid cannot represent the horizontally-correlated error. Consequently, convergence degrades from 10–15 iterations to hundreds, or the method diverges entirely. Coarsening only in the smooth direction helps but fails when the strong-coupling direction varies across the domain.

Domain decomposition with direct subdomain solves, however, remains robust because exact factorization has no assumptions about error smoothness. ILU also survives when the matrix is otherwise well-conditioned. Specialized multigrid variants that align coarsening with the coupling direction can work but require problem-specific tuning.

6.2.2 Variable Coefficients

Some problems have matrix entries that vary by factors of 10^6 or more across the domain, reflecting material properties that differ dramatically between regions. This creates error modes that are highly localized near the boundaries between regions, spanning only a few grid points rather than varying smoothly across the domain.

Such problems arise whenever the physical domain contains materials with very different properties. Geological formations have permeability varying by 10^6–10^{12} between rock types. Composite materials have stiffness ratios of 10^3–10^6 between constituents. Any simulation coupling regions with different physics (fluid next to solid, for instance) exhibits similar jumps.

Multigrid coarsening cannot handle these problems because it skips over the localized modes at material interfaces. These modes have small spatial extent but dominate the condition number; coarse grids that miss them produce corrections that ignore the dominant error. Standard domain decomposition coarse spaces also fail because they represent smooth functions over subdomains, not the localized interface modes. As a result, iterations grow from 20 to 200+ as contrast increases.

Domain decomposition with adaptive coarse spaces survives by computing the problematic modes explicitly during setup. These methods solve small eigenproblems on each subdomain to identify modes that the standard coarse space would miss, then include them. ILU and direct subdomain solves also survive because they operate on exact matrix entries without coarse-grid assumptions.

6.2.3 Indefinite Problems

Some matrices have both positive and negative eigenvalues, meaning that moving in certain directions increases the residual rather than decreasing it. Iterative methods that assume all corrections reduce error can amplify error components associated with negative eigenvalues.

Such problems arise in wave propagation at high frequency, where solutions oscillate rapidly. They also arise in constrained problems where the matrix has a block of zeros or negative entries enforcing constraints..

Multigrid diverges on these problems because the coarse-grid correction step assumes that correcting smooth error always helps. When some eigenvalues are negative, the correction amplifies those components instead of reducing them. ILU can also fail because the incomplete factors may have the wrong sign structure, causing the preconditioner to amplify error rather than reduce it.

Domain decomposition with direct subdomain solves, however, handles indefiniteness through exact factorization with pivoting, which correctly handles any sign structure. Specialized preconditioners designed for saddle-point structure also work by preserving the block structure rather than treating the matrix as a single system.

6.2.4 Lack of Geometric Embedding

Methods that exploit geometric structure require the matrix graph to have small vertex separators arising from spatial locality. Standard PDE discretizations have this property because each mesh point couples only to immediate neighbors, yielding separators of size $O(n^{1/2})$ in 2D and $O(n^{2/3})$ in 3D.

Nonlocal models break this assumption. For example, olecular systems discretized by bond connectivity rather than mesh connectivity may also lack small separators when the molecular topology (rings, long chains) does not embed well in 3D.

Nested dissection fails on such problems because its complexity advantage comes entirely from small separators. For graphs with $O(n)$ separators, recursive bisection provides no benefit over other orderings. Geometric multigrid requires regular spatial coarsening that makes no sense without mesh structure.

Algebraic multigrid survives by constructing coarse levels from matrix entries rather than spatial coordinates, though its effectiveness varies with problem structure. ILU operates purely on matrix entries and sparsity pattern. For problems genuinely lacking spatial structure, sparse direct solvers with approximate minimum degree ordering may be the only robust option.

6.2.5 Lack of Block Structure

Static condensation requires the matrix to have a specific block structure where interior unknowns couple only within local regions (elements), not across region boundaries. This creates block-diagonal structure in the interior-interior coupling, enabling parallel elimination.

This structure is a property of how the problem was discretized, not the underlying physics. Discontinuous Galerkin and hybridized methods create this structure by design. Standard continuous finite elements do not: boundary unknowns couple to interiors of multiple adjacent elements, destroying the required block-diagonal structure.

Static condensation is simply unavailable without this structure. No reordering or reformulation can create block-diagonal interior coupling where the discretization did not produce it. Even when the structure exists, condensation only pays off when interior unknowns dominate.

All other methods are unaffected by this constraint. Multigrid, domain decomposition, ILU, and nested dissection operate on the assembled matrix without requiring element-interior structure. Thus, this constraint affects only whether condensation can be added to the composition.

6.2.6 Summary

Table 5 summarizes which methods survive each problem characteristic. Problem characteristics, not method preferences, determine viable compositions. An application with extreme directional coupling cannot use standard multigrid regardless of its theoretical advantages; an application with high coefficient variation requires adaptive coarse spaces regardless of their setup cost.

Table 5: Problem characteristics and method viability. ✓= method works; × = method fails or degrades severely; ○ = method works with modifications.

	MG	DD	ILU	ND	Cond.
Directional coupling	×	✓	✓	✓	✓
Coefficient variation	×	○	✓	✓	✓
Indefiniteness	×	✓	×	✓	✓
Lack of graph structure	○	○	✓	×	✓
Lack of block structure	✓	✓	✓	✓	×

6.3 KEY INSIGHT

Each method addresses at least one bottleneck (*e.g.*, reordering and condensation reduce memory; domain decomposition distributes memory; multigrid reduces iterations), so composition is required to address both memory and communication at scale. In addition, solving a problem at scale requires identifying which methods survive the problem's constraints, then composing those survivors to cover both bottlenecks.

7 Application Spotlight

Production codes navigate memory and communication constraints by identifying available structural properties,

checking robustness requirements, and composing viable methods. Several large-scale problems illustrate this process across a variety of applications with different constraint landscapes. Here, each example identifies the structural properties available, notes which robustness constraints apply, describes the resulting composition, and summarizes performance at scale.

7.1 ExaWind: Hybrid Grids at Exascale

The ExaWind wind farm simulation framework models atmospheric flow around wind turbines, coupling two solvers to resolve length scales from millimeter-thick boundary layers on blade surfaces to kilometer-scale atmospheric turbulence [58]. On Frontier (AMD MI250X GPUs, 8 GCDs per node), production runs reach 40 billion grid points on approximately 4,000 nodes. Both solvers produce elliptic pressure systems that are typically good candidates for domain decomposition and multigrid.

- **Constraints.** The pressure systems have modest directional coupling and smooth coefficients, so multigrid works. However, moving turbine blades require recomputing multigrid hierarchies every timestep, preventing amortization of setup.

- **Composition.** Both solvers use domain decomposition for memory distribution and multigrid for iteration reduction. The block-structured solver uses geometric multigrid; the unstructured solver uses algebraic multigrid (hypre BoomerAMG).

- **Performance.** Solve phases accelerate on GPUs; total time grows due to repeated multigrid setup. Ongoing work targets incremental hierarchy updates to reduce setup cost.

7.2 XGC: Extreme Directional Coupling

The XGC gyrokinetic code simulates edge plasma in tokamak fusion reactors, where particles spiral tightly around magnetic field lines while drifting slowly across them [42]. The simulation domain is a toroidal mesh with 10–100 million vertices; the dominant cost is solving a gyrokinetic Poisson system at each timestep. On Summit (NVIDIA V100 GPUs, 6 per node) and Frontier, XGC runs on 1,000–10,000 nodes for reactor-scale simulations.

- **Constraints.** Coupling ratios exceed 10^6:1 between parallel and perpendicular directions, so multigrid fails. Coarsening selects only strongly-coupled connections, producing coarse grids that miss perpendicular error entirely.

- **Composition.** XGC uses domain decomposition with direct subdomain solves (sparse LU via SuperLU_DIST) instead of multigrid. Direct factorization handles any coupling pattern. A physics-based coarse space reflecting the adiabatic electron response provides global coupling, keeping iterations at 20–40.

- **Performance.** The composition trades multigrid's $O(n)$ per-iteration cost for robustness. Direct subdomain solves are more expensive but feasible for appropriately sized subdomains.

7.3 Reservoir Simulation: High Coefficient Variation

Subsurface flow simulation solves pressure equations on geological meshes representing oil and gas reservoirs [74]. A typical reservoir model contains 1–100 million cells spanning kilometers horizontally and hundreds of meters vertically, with cell aspect ratios of 10:1 to 100:1. The pressure equation is elliptic, typically a good candidate for multigrid.

- **Constraints.** Permeability varies by 10^6–10^{12} between geological layers (shale versus sandstone, for instance), breaking both multigrid and standard domain decomposition coarse spaces. Both miss localized modes at material interfaces.

- **Composition.** Production codes use domain decomposition with adaptive coarse spaces (GenEO or spectral coarse spaces) that compute the problematic modes explicitly. ILU-preconditioned subdomain solves replace multigrid.

- **Performance.** Adaptive coarse spaces restore 20–40 iterations at the cost of per-subdomain solves during setup. Total time is 2–5$\times$ longer than for uniform-coefficient problems.

7.4 NekRS: Rich Block Structure

The NekRS spectral element code solves incompressible Navier-Stokes for turbulent flow simulation [33]. Using polynomial degree $p = 15$, each hexahedral element contains $(p + 1)^3 = 4096$ unknowns. Approximately 67% are interior to elements; 35% lie on faces. On Frontier and Aurora (Intel Data Center Max GPUs), NekRS scales to the full machine for simulations with billions of grid points. The high-order discretization creates block-diagonal interior coupling that condensation requires.

- **Constraints.** All robustness constraints are satisfied: moderate coefficients, block structure for condensation, smooth error for polynomial multigrid.

- **Composition.** Static condensation eliminates interior unknowns through batched dense factorizations, reducing dimension by $5\times$. Domain decomposition distributes the reduced system. Polynomial multigrid achieves 10–20 iterations.
- **Performance.** Batched dense operations achieve high arithmetic intensity on GPUs. The $5\times$ reduction keeps the skeleton system in fast memory. Strong scaling reaches thousands of GPUs with 80%+ parallel efficiency.

7.5 FROSch: Hardware-Efficient Implementation

When all robustness constraints are satisfied, implementation efficiency becomes the primary concern. FROSch implements two-level domain decomposition preconditioners within the Trilinos library, targeting linear elasticity and similar well-conditioned problems [75]. On Summit (NVIDIA V100 GPUs), FROSch demonstrates GPU acceleration strategies for domain decomposition.

- **Constraints.** Test problems are well-conditioned with no extreme coupling or coefficient variation. The challenge is hardware efficiency, not algorithmic robustness.
- **Composition.** FROSch uses two-level domain decomposition with GDSW coarse spaces. Implementation choices target GPUs: NVIDIA MPS runs multiple MPI ranks per GPU to improve occupancy; inexact subdomain solves (ILU) expose more parallelism than exact factorization; mixed-precision (FP32) preconditioner construction reduces memory traffic.
- **Performance.** These choices yield $2\times$ speedup on solve phases versus CPU. Algorithmic composition is necessary but not sufficient: hardware-aware implementation determines whether theoretical advantages are realized.

7.6 KEY INSIGHT

Problem characteristics determine viable compositions. Under difficult problem regimes, codes substitute robust alternatives and accept higher cost for guaranteed convergence.

8 Summary and Future Directions

Sparse linear systems arising from PDE discretizations remain the primary computational bottleneck in large-scale scientific computing, consuming 70-80% of runtime in production applications. On contemporary architectures the dominant cost is no longer arithmetic but data movement through deep memory hierarchies and communication across nodes. Consequently, the effectiveness of a solver depends not only on convergence properties but on how well algorithmic structure aligns with hardware structure.

Across reordering, static condensation, incomplete factorizations, multigrid, and domain decomposition, a consistent theme emerges. Each method exposes a different form of locality, reduced coupling, or parallelism. Yet no single method can simultaneously minimize fill, communication, and iteration count. Large-scale simulations therefore compose multiple techniques routinely, not from theoretical convenience but from architectural necessity.

Understanding when methods fail is as important as understanding when they succeed. Anisotropy, high coefficient contrast, irregular geometry, and indefiniteness are not edge cases; they characterize the problems that matter most. The simulations driving scientific progress are precisely those where standard assumptions break down. Methods that gracefully handle such complexity, rather than assuming it away, determine what problems can actually be solved at scale.

8.1 Toward Architecturally Aligned Solvers

What remains insufficiently understood is how solver structure interacts with heterogeneous architectures. Seemingly similar algebraic choices behave very differently on CPUs, GPUs, and APUs such that achieving performance requires methods exploiting the right parallelism at the right granularity while minimizing data movement and global synchronization.

Several directions have been demonstrated to yield improvements. Hybridized and high-order methods expose dense local kernels mapping naturally to GPUs. Block-structured formulations enable task parallelism and reduce global coupling. Communication-avoiding variants of multigrid and domain decomposition offer paths toward latency-tolerant scalability. Performance models coupling operator structure with memory hierarchy behavior pose opportunities for predicting when and how solver components should be composed.

Solvers integrating these directions would be task-parallel, memory-aware, and designed for heterogeneous environments. Equally important, they would degrade gracefully when standard assumptions fail. They would scale not through algorithmic optimality on idealized problems but because their structure is engineered for the realities of modern computing.

As architectures evolve toward deeper hierarchies and increased heterogeneity, the most effective sparse solvers

will be those bridging mathematical structure and machine structure.

References

[1] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. Performance, design, and autotuning of batched GEMM for GPUs. In *International Conference on High Performance Computing*, pages 21–38, 2016.

[2] Patrick R Amestoy, Timothy A Davis, and Iain S Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[3] Patrick R Amestoy, Iain S Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[4] Hartwig Anzt, Edmond Chow, and Jack Dongarra. ParILUT—a new parallel threshold ILU factorization. *SIAM Journal on Scientific Computing*, 40(4):C503–C519, 2018.

[5] Hartwig Anzt, Terry Cojean, Chen Yen-Chen, Jack Dongarra, Goran Flegar, Pratik Nayak, Stanimire Tomov, Yuhsiang M Tsai, and Weichung Wang. Evaluating the performance of NVIDIA's A100 Ampere GPU for sparse and batched computations. *arXiv preprint arXiv:2008.08478*, 2020.

[6] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. On the performance and energy efficiency of sparse linear algebra on gpus. *The International Journal of High Performance Computing Applications*, 31(5):375–390, 2017.

[7] Santiago Badia, Alberto F Martín, and Javier Principe. Multilevel balancing domain decomposition at extreme scales. *SIAM Journal on Scientific Computing*, 38(1):C22–C52, 2016.

[8] Vahid Badrkhani, Marco F P ten Eikelder, René R Hiemstra, and Dominik Schillinger. The matrix-free macro-element hybridized discontinuous Galerkin method for steady and unsteady compressible flows. *International Journal for Numerical Methods in Fluids*, 97(4):462–483, 2025.

[9] Allison H Baker, Robert D Falgout, Tzanio V Kolev, and Ulrike Meier Yang. Scaling algebraic multigrid solvers: On the road to exascale. In *Competence in High Performance Computing*, pages 215–226. 2011.

[10] Michele Benzi, Gene H Golub, and Jörg Liesen. Numerical solution of saddle point problems. *Acta Numerica*, 14:1–137, 2005.

[11] Amanda Bienz, Robert D Falgout, William Gropp, Luke N Olson, and Jacob B Schroder. Reducing parallel communication in algebraic multigrid through sparsification. *SIAM Journal on Scientific Computing*, 38(5):S332–S357, 2016.

[12] Amanda Bienz, William D Gropp, and Luke N Olson. Reducing communication in algebraic multigrid with multi-step node aware communication. *The International Journal of High Performance Computing Applications*, 34(5):547–561, 2020.

[13] Matthias Bollhöfer, Olaf Schenk, Radim Janalik, Steve Hamm, and Kiran Gullapalli. High performance block incomplete LU factorization. *Applied Numerical Mathematics*, 157:399–417, 2020.

[14] Marian Brezina, Andrew J Cleary, Robert D Falgout, Van E Henson, Jim E Jones, Thomas A Manteuffel, Stephen F McCormick, and John W Ruge. Algebraic multigrid based on element interpolation (AMGe). *SIAM Journal on Scientific Computing*, 22(5):1570–1592, 2000.

[15] Marian Brezina, Robert Falgout, Scott MacLachlan, Tom Manteuffel, Steve McCormick, and John Ruge. Adaptive smoothed aggregation (αSA) multigrid. *SIAM Review*, 47(2):317–346, 2005.

[16] William L Briggs, Van Emden Henson, and Steve F McCormick. *A multigrid tutorial*. SIAM, 2nd edition, 2000.

[17] Matteo Caldana, Paola F Antonietti, and Luca Dede'. A deep learning algorithm to accelerate algebraic multigrid methods in finite element solvers of 3D elliptic PDEs. *Computers & Mathematics with Applications*, 167:217–231, 2024.

[18] Léopold Cambier, Chao Chen, Erik G Boman, Sivasankaran Rajamanickam, Raymond S Tuminaro, and Eric Darve. An algebraic sparsified nested dissection algorithm using low-rank approximations. *SIAM Journal on Matrix Analysis and Applications*, 41(2):715–746, 2020.

[19] Ümit V Çatalyürek, Cevdet Aykanat, and Enver Kayaaslan. Hypergraph partitioning-based fill-reducing ordering for symmetric matrices. *SIAM Journal on Scientific Computing*, 33(4):1996–2023, 2011.

[20] Yen-Hsiang Chang, Aydın Buluç, and James Demmel. Parallelizing the approximate minimum degree ordering algorithm: Strategies and evaluation. *arXiv preprint arXiv:2504.17097*, 2025.

[21] Ru Chen, Jinchao Xu, and Ludmil Zikatanov. Learning optimal multigrid smoothers via neural networks. Technical Report LLNL-PROC-819678, Lawrence Livermore National Laboratory, 2020.

[22] Edmond Chow and Aftab Patel. Fine-grained parallel incomplete LU factorization. *SIAM Journal on Scientific Computing*, 37(2):C169–C193, 2015.

[23] Bernardo Cockburn, Jayadeep Gopalakrishnan, and Raytcho Lazarov. Unified hybridization of discontinuous galerkin, mixed, and continuous galerkin methods for second order elliptic problems. *SIAM Journal on Numerical Analysis*, 47(2):1319–1365, 2009.

[24] Roy R Craig Jr and Mervyn CC Bampton. Coupling of substructures for dynamic analyses. *AIAA Journal*, 6(7):1313–1319, 1968.

[25] Robert Cummings, Matthew Fahrbach, and Animesh Fatehpuria. A fast minimum degree algorithm and matching lower bound. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 724–734. SIAM, 2021.

[26] Elizabeth Cuthill and James McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, pages 157–172. ACM, 1969.

[27] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, 2011.

[28] Clark R Dohrmann. A preconditioner for substructuring based on constrained energy minimization. *SIAM Journal on Scientific Computing*, 25(1):246–258, 2003.

[29] Jack Dongarra et al. *Applied mathematics at the US Department of Energy: Past, present and a view to the future*. SIAM, 2014.

[30] Oliver G Ernst and Martin J Gander. Why it is difficult to solve Helmholtz problems with classical iterative methods. In *Numerical Analysis of Multiscale Problems*, volume 83 of *Lecture Notes in Computational Science and Engineering*, pages 325–363. Springer, 2012.

[31] Charbel Farhat, Michel Lesoinne, Patrick LeTallec, Kendall Pierson, and Daniel Rixen. FETI-DP: a dual-primal unified FETI method—part I: A faster alternative to the two-level FETI method. *International Journal for Numerical Methods in Engineering*, 50(7):1523–1544, 2001.

[32] Luca Fedeli, Axel Huebl, France Boillod-Cerneux, Thomas Clark, Kevin Gott, Conrad Hillairet, Stephan Jaure, Adrien Leblanc, Rémi Lehe, Andrew Myers, et al. Pushing the frontier in the design of laser-based electron accelerators with groundbreaking mesh-refined particle-in-cell simulations on exascale-class supercomputers. In *SC22: international conference for high performance computing, networking, storage and analysis*, pages 1–12. IEEE, 2022.

[33] Paul F Fischer, Ali Karakus, Noel A Phillips, Janine C Hoffman, Elia Merzari, and Stefan Kerkemeier. Scalability of high-order spectral element solvers for incompressible flow. *The International Journal of High Performance Computing Applications*, 36(1):44–61, 2022.

[34] Alice Gatti, Zhixiong Hu, Tess Smidt, Esmond G Ng, and Pieter Ghysels. Graph partitioning and sparse matrix ordering using reinforcement learning and graph neural networks. *Journal of Machine Learning Research*, 23(303):1–28, 2022.

[35] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[36] John R. Gilbert and Robert Endre Tarjan. The analysis of a parallel nested dissection algorithm. *Numerische Mathematik*, 50:377–404, 1987.

[37] Robert J Guyan. Reduction of stiffness and mass matrices. *AIAA Journal*, 3(2):380–380, 1965.

[38] Huan He, Owen Queen, Theodoros Koker, Cristian Cuevas, Theodoros Tsiligkaridis, and Marinka Zitnik. Reducing operator complexity of Galerkin coarse-grid operators with machine learning. *SIAM Journal on Scientific Computing*, 46(4):A2622–A2649, 2024.

[39] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14. IEEE, 2014.

[40] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[41] Seid Koric and Anshul Gupta. Sparse matrix factorization in the implicit finite element method on petascale architecture. *Computer Methods in Applied Mechanics and Engineering*, 302:281–292, 2016.

[42] S Ku, CS Chang, R Hager, RM Churchill, GR Tynan, Istvan Cziegler, M Greenwald, J Hughes, Scott E Parker, MF Adams, et al. A fast low-to-high confinement mode bifurcation dynamics in the boundary-plasma gyrokinetic code xgc1. *Physics of Plasmas*, 25(5), 2018.

[43] Richard J Lipton, Donald J Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16(2):346–358, 1979.

[44] Yuechen Lu, Lijie Zeng, Tengcheng Wang, Xu Fu, Wenxuan Li, Helin Cheng, Dechuang Yang, Zhou Jin, Marc Casas, and Weifeng Liu. AmgT: Algebraic multigrid solver on tensor cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '24)*. IEEE, 2024.

[45] Stephen F McCormick, John Ruge, and Ben S Southworth. Algebraic error analysis for mixed-precision multigrid solvers. *SIAM Journal on Scientific Computing*, 43(5):S392–S419, 2021.

[46] Jozef A Meijerink and Henk A Van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric m-matrix. *Mathematics of Computation*, 31(137):148–162, 1977.

[47] Mellanox Technologies. Introducing 200G HDR InfiniBand Solutions. Technical report, Mellanox Technologies, 2018. White paper.

[48] Maxim Naumov, Marat Arsaev, Patrice Castonguay, Jonathan Cohen, Julien Demouth, Joe Eaton, Simon Layton, Nikolai Marber, Robert Strzodka, Stanimire Tomov, et al. AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing*, 37(5):S602–S626, 2015.

[49] Ngoc Cuong Nguyen, Jaume Peraire, and Bernardo Cockburn. An implicit high-order hybridizable discontinuous Galerkin method for linear convection-diffusion equations. *Journal of Computational Physics*, 228(9):3232–3254, 2009.

[50] NVIDIA Corporation. NVIDIA A100 Tensor Core GPU Architecture. Technical report, NVIDIA Corporation, 2020. Whitepaper version 1.0.

[51] Wolfgang Ost, Christian Schulz, and Darren Strash. Engineering data reduction for nested dissection. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 46–60. SIAM, 2021.

[52] Anthony T Patera. A spectral element method for fluid dynamics: Laminar flow in a channel expansion. *Journal of Computational Physics*, 54(3):468–488, 1984.

[53] John W Ruge and Klaus Stüben. Algebraic multigrid. In *Multigrid Methods*, pages 73–130. SIAM, 1987.

[54] Yousef Saad. Ilut: A dual threshold incomplete lu factorization. *Numerical Linear Algebra with Applications*, 1(4):387–402, 1994.

[55] Yousef Saad. ILUM: A multi-elimination ILU preconditioner for general sparse matrices. *SIAM Journal on Scientific Computing*, 17(4):830–847, 1996.

[56] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2nd edition, 2003.

[57] Yousef Saad and Brian Suchomel. ARMS: An algebraic recursive multilevel solver for general sparse linear systems. *Numerical Linear Algebra with Applications*, 9(5):359–378, 2002.

[58] Ashesh Sharma, Michael J Brazell, Ganesh Vijayakumar, Shreyas Ananthan, Lawrence Cheung, Nathaniel deVelder, Marc T Henry de Frahan, Neil Matula, Paul Mullowney, Jon Rood, et al. Exawind: Open-source cfd for hybrid-rans/les geometry-resolved wind turbine simulations in atmospheric flows. *Wind Energy*, 27(3):225–257, 2024.

[59] Barry F Smith, Petter E Bjorstad, and William D Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

[60] Nicole Spillane, Victorita Dolean, Patrice Hauret, Frédéric Nataf, Clemens Pechstein, and Robert Scheichl. Abstract robust coarse spaces for systems of PDEs via generalized eigenproblems in the overlaps. *Numerische Mathematik*, 126(4):741–770, 2014.

[61] Klaus Stüben. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 128(1-2):281–309, 2001.

[62] Ali Taghibakhshi, Scott MacLachlan, Luke Olson, and Matthew West. Optimization-based algebraic

multigrid coarsening using reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 34, pages 12129–12140, 2021.

[63] Mark Taylor, Peter M Caldwell, Luca Bertagna, Conrad Clevenger, Aaron Donahue, James Foucar, Oksana Guba, Benjamin Hillman, Noel Keen, Jayesh Krishna, et al. The simple cloud-resolving e3sm atmosphere model running on the frontier exascale system. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–11, 2023.

[64] Nick Thakkar et al. Amd instinct mi250x and mi250 performance. Technical report, AMD, 2023.

[65] WF Tinney and JW Walker. Direct solution of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809, 1967.

[66] Andrea Toselli and Olof B Widlund. *Domain Decomposition Methods: Algorithms and Theory*, volume 34 of *Springer Series in Computational Mathematics*. Springer, 2005.

[67] Yu-Hsiang M Tsai, Natalie Beams, and Hartwig Anzt. Three-precision algebraic multigrid on GPUs. *Future Generation Computer Systems*, 149:280–293, 2023.

[68] Xuemin Tu. Three-level BDDC in three dimensions. *SIAM Journal on Scientific Computing*, 29(4):1759–1780, 2007.

[69] Petr Vaněk, Jan Mandel, and Marian Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing*, 56(3):179–196, 1996.

[70] Panayot S Vassilevski and Ulrike Meier Yang. Reducing communication in algebraic multigrid using additive variants. *Numerical Linear Algebra with Applications*, 21(2):275–296, 2014.

[71] Wei-Lai Wan, Tony F Chan, and Barry Smith. An energy-minimizing interpolation for robust multigrid methods. *SIAM Journal on Scientific Computing*, 21(4):1632–1649, 2000.

[72] Pieter Wesseling. *An Introduction to Multigrid Methods*. R.T. Edwards, 2004.

[73] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[74] Li Wu, Junqiang Wang, Deli Jia, Jiqun Zhang, Ruichao Zhang, Yiqun Yan, Yanfang Yin, and Shuoliang Wang. An efficient multiscale simulation framework integrating dynamic heterogeneity for accurate waterflooding prediction. *Scientific Reports*, 15(1):23680, 2025.

[75] Ichitaro Yamazaki, Alexander Heinlein, and Sivasankaran Rajamanickam. An experimental study of two-level schwarz domain-decomposition preconditioners on gpus. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 680–689. IEEE, 2023.

[76] Mihalis Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic Discrete Methods*, 2(1):77–79, 1981.

[77] Fan Yuan, Chuanfu Xu, Jie Liu, Xiaoqiang Yue, Shengguo Li, and Hongxia Wang. CRAMG: A communication-reduced algebraic multigrid method. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS '25)*, pages 397–411, Salt Lake City, UT, USA, 2025. ACM.

[78] Behrooz Zarebavani, Danny M Kaufman, David I W Levin, and Maryam Mehri Dehnavi. Adaptive algebraic reuse of reordering in Cholesky factorizations with dynamic sparsity patterns. *ACM Transactions on Graphics*, 44(4):1–17, 2025.