A GPU-Based Multilevel Additive Schwarz Preconditioner for Cloth and Deformable Body Simulation

BOTAO WU, Style3D Research, China
ZHENDONG WANG, Style3D Research, China
HUAMIN WANG, Style3D Research, China

(a) A front view (b) A side view (c) A closeup top view (d) A closeup bottom view

Fig. 1. A wool coat example with 92K vertices and 182K triangles. We develop a GPU-based multilevel additive Schwarz preconditioner that empowers our simulator to animate this example with the time step $\Delta t = 1/100$s at 37FPS, or more than 40FPS without collision handling. Compared with other preconditioners, our preconditioner is not only fast and effective, but also cheap to precompute. Because of these strengths, our preconditioner is suitable for both linear and nonlinear solvers in cloth and deformable body simulation, especially with dynamic contacts.

In this paper, we wish to push the limit of real-time cloth and deformable body simulation to a higher level with 50K to 500K vertices, based on the development of a novel GPU-based multilevel additive Schwarz (MAS) preconditioner. Similar to other preconditioners under the MAS framework, our preconditioner naturally adopts multilevel and domain decomposition concepts. But contrary to previous works, we advocate the use of small, non-overlapping domains that can well explore the parallel computing power on a GPU. Based on this idea, we investigate and invent a series of algorithms for our preconditioner, including multilevel domain construction using Morton codes, low-cost matrix precomputation by one-way Gauss-Jordan elimination, and conflict-free symmetric-matrix-vector multiplication in runtime preconditioning. The experiment shows that our preconditioner is effective, fast, cheap to precompute and scalable with respect to stiffness and problem size. It is compatible with many linear and nonlinear solvers used in cloth and deformable body simulation with dynamic contacts, such as PCG, accelerated gradient descent and L-BFGS. On a GPU, our preconditioner speeds up a PCG solver by approximately a factor of four, and its CPU version outperforms a number of competitors, including ILU0 and ILUT.

CCS Concepts: • **Computing methodologies** → **Physical simulation**; **Multiscale systems**; • **Mathematics of computing** → **Solvers**.

Additional Key Words and Phrases: additive Schwarz, multilevel method, preconditioning, conjugate gradient, cloth and deformable body simulation

ACM Reference Format:
Botao Wu, Zhendong Wang, and Huamin Wang. 2022. A GPU-Based Multilevel Additive Schwarz Preconditioner for Cloth and Deformable Body Simulation. *ACM Trans. Graph.* 41, 4, Article 63 (July 2022), 14 pages. https://doi.org/10.1145/3528223.3530085

Authors' addresses: Botao Wu, Style3D Research, China, botaow24@outlook.com; Zhendong Wang, Style3D Research, China, wang.zhendong.619@gmail.com; Huamin Wang, Style3D Research, China, wanghmin@gmail.com.

0730-0301/2022/7-ART63 $15.00
https://doi.org/10.1145/3528223.3530085

1 INTRODUCTION

Many cloth and deformable body simulation problems can be boiled down to large-scale nonlinear optimization problems, and how to solve these problems fast has been a major challenge bothering mathematicians and scientists for decades. In the past, they have studied this topic extensively and developed a variety of optimization methods. Graphics researchers [Bouaziz et al. 2014; Fratarcangeli et al. 2016; Wang and Yang 2016; Wu et al. 2020] have borrowed these methods to accomplish real-time simulation of cloth and deformable bodies with under 50K vertices. But for real-time simulation of even

higher qualities, the existing methods are insufficient to address the associated large-scale, ill-conditioned problems. Given the imminent desire of real-time simulation in various graphics applications, the need of faster solvers is more urgent than ever before.

Multilevel and *domain decomposition* methods have gained substantial popularity in recent years, because of their effectiveness in solving large-scale, ill-conditioned problems. Among these methods, multilevel additive Schwarz (MAS) preconditioning with a (nonlinear) iterative method is a particularly favorable one, not only because it adopts both concepts, but also because it is simple. A MAS preconditioner naturally avoids the domain discontinuity issue involved in other domain decomposition methods [Kim and James 2011; Li et al. 2019; Wu et al. 2015; Yang et al. 2013], and it does not rely heavily on hierarchical coarsening and propagation operators [Tamstorf et al. 2015; Wang et al. 2018]. Compared with other preconditioners, such as multigrid ones, a MAS preconditioner is also notably parallelizable: all of the domains across different levels can run in parallel, rather than in any cyclic order. This makes a MAS preconditioner an attractive candidate for GPU acceleration.

The benefits of a MAS preconditioner motivate us to investigate a critical question: *what is an ideal form of a MAS preconditioner that can well explore the GPU computing power in cloth and deformable body simulation*? After all, mathematicians and scientists have studied MAS preconditioning for decades [Dolean et al. 2015], but the performances of their preconditioners, not to mention the performances on GPUs, are not always satisfactory. In general, the total cost spent by a preconditioned solver in one time step is:

$$C_{\text{total}} = N_0 \cdot C_0 + N_1 \cdot C_1 + K \cdot (C_2 + C_{\text{solver}}), \tag{1}$$

where C_0, C_1 and C_2 are the costs spent by the preconditioner on analyzing the matrix structure, precomputing needed data and performing runtime preconditioning, C_{solver} is the solver's intrinsic cost, N_0 and N_1 are the numbers of structure analysis and matrix precomputation calls, and K is the number of iterations. We can treat N_0, N_1 and C_{solver} as solver-specific constants. In particular, a nonlinear solver may update the system matrix multiple times in one time step, causing N_1 to be greater than one. To handle dynamic contacts and fractures, a solver may even need to modify the matrix structure over time, resulting in a nonzero or even greater N_0. Therefore, an ideal preconditioner should make all of its relevant factors in Eq. 1, including C_0, C_1, C_2 and K, as small as possible. Improving the convergence rate, i.e., decreasing K, while ignoring other factors, would fail to meet the key point of preconditioning. In fact, if our goal is to achieve the fastest convergence only, why not use a direct method like Intel MKL PARDISO instead?

In this paper, we would like to develop an ideal MAS preconditioner and we face many design and implementation choices. Among them, perhaps the most significant one is the domain size. In previous works, researchers often use large domains to reduce inter-domain connections, i.e., the part of the system matrix not covered by domains. Here we are thinking about the opposite: using many but small non-overlapping domains at different levels, so that each domain can be sufficiently and efficiently solved on a GPU. Based on this idea, our research on MAS preconditioning includes the following contributions.

(a) School dress (b) Party dress (c) Leather coat

(d) Down coat (e) Hiking jacket (f) T-shirt

Fig. 2. The cloth simulation examples. Our simulator efficiently simulates these examples with the numbers of vertices ranging from 76K to 467K. The efficiency comes largely from the use of our MAS preconditioner.

- *Multilevel domain construction.* Based on Morton code sorting, we present a supernode splitting method and a skipping approach, for fast and simple domain partitioning and coarse space construction.
- *Matrix precomputation.* We propose a one-way elimination algorithm and a compact matrix format, to calculate the sub-matrix inverse of each domain with low computational and storage costs. We also develop a selective update scheme to address slight matrix modifications.
- *Runtime preconditioning.* Given the precomputed sub-matrix inverses, we perform runtime preconditioning by simple matrix-vector multiplication. To further reduce this cost, we invent a symmetric-matrix-vector multiplication method with balanced workload and zero write conflict.

We implement our preconditioner for both GPUs and CPUs. Our experiment demonstrates its effectiveness on the convergence rate, its remarkably low runtime cost and its high scalability with respect to stiffness and problem size. Because of these, although it does not achieve the fastest convergence rate, it still significantly outperforms other preconditioners, including multigrid AmgX on a GPU [Naumov et al. 2015; NVIDIA 2021], ichol and ILUT on a CPU [Chen et al. 2021; Saad 1994]. Furthermore, our preconditioner has a low precomputation overhead, making it suitable for both

linear and nonlinear solvers, especially in accurate simulation with dynamic contacts. We validate the performance of our preconditioner in many cloth and deformable body simulation examples as shown in Fig. 1, 2 and 15, including real-time ones. (CPU code for this work is downloadable at https://wanghmin.github.io/Wu-2022-AGM/Wu-2022-AGM.zip.)

2 RELATED WORK

Cloth and deformable body dynamics. Fast and realistic simulation of cloth and deformable body dynamics [Terzopoulos et al. 1987] have been active graphics research topics for decades. Early simulation techniques [Bridson et al. 2002; Teran et al. 2003] often choose to use explicit time integration with small time steps, but they suffer from numerical instability as stiffness or nonlinearity increases. To achieve numerical stability even when the time step is large, researchers [Baraff and Witkin 1998; Teran et al. 2005; Volino et al. 2009] explored implicit time integration based on Newton iterations and their simulators solve one or multiple linearized systems in every simulation step. Inspired by the variational formulation [Ortiz and Stainier 1999], Kharevych et al. [2006] and Martin et al. [2011] formulated implicit time integration as a nonlinear optimization problem. In recent years, researchers [Bouaziz et al. 2014; Fratarcangeli et al. 2016; Liu et al. 2013, 2017] become particularly interested in developing optimization techniques for fast cloth and deformable body simulation. For example, Wang et al. [2015; 2016] showed that a series of nonlinear iterative methods can benefit from the use of a good preconditioner for fast GPU-based simulation, including gradient descent, nonlinear conjugate gradient and L-BFGS.

Multilevel methods. The most popular type of multilevel methods is multigrid. Multigrid methods and preconditioners have been extensively studied in the past, but their usages in computer graphics are limited in practice. The main challenge associated with multigrid is a need of effective coarsening and prolongation operators, especially for non-uniform, unstructured meshes. In the past, graphics researchers [Lee et al. 2010; Wang et al. 2018; Xian et al. 2019; Zhu et al. 2010] investigated many geometric multigrid methods based on the construction of a mesh hierarchy, and they experienced a variety of implementation issues. Compared with geometric multigrids, algebraic multigrids [Naumov et al. 2015; Tamstorf et al. 2015] are advantageous in their independency of mesh topology. However, they often need considerable precomputation time to analyze the matrix sparsity structure, which makes them more costly when they handle time-varying systems with contacts.

On a GPU, multigrid methods and preconditioners suffer from two fundamental limitations affecting their parallelizability. First, they must process all of the levels sequentially, e.g., from fine to coarse and then from coarse to fine in a so-called V-cycle. Second, at the coarsest level, a single coarsened problem is too small to feed up all of the GPU cores. These two limitations do not exist in multilevel additive Schwarz preconditioners.

Recently, researchers have also investigated other multilevel preconditioners, including a multilevel preconditioner for Laplacian matrices [Krishnan et al. 2013] and a multilevel incomplete Cholesky preconditioner [Chen et al. 2021]. But similar to multigrid methods and preconditioners, they suffer from a variety of problems, such as algorithmic complexity, extended precomputation time and/or incompatibility with parallelization.

Domain decomposition. The idea of domain decomposition has been well explored by mathematicians and scientists, since the seminal work by Schwarz [1870] more than one century ago. Specifically relevant to our work, an additive Schwarz (AS) method [Cai and Saad 1996; Dryja and Widlund 1989; Rodrigue et al. 1989] divides a large, sparse linear system into smaller ones and solves them in parallel. Intuitively, an AS method [Dolean et al. 2015; Gander 2008] can be viewed as the block Jacobi method, if the domains do not overlap and the sub-problems are exactly solved in every iteration. But in general, it is suggested to use overlapping domains for an AS method to converge robustly [Frommer and Szyld 1999]. An alternative way of improving the convergence of an AS method is to turn it into an asymmetric variant, i.e., restricted additive Schwarz (RAS) [Cai and Sarkis 1999]. Both AS and RAS can work as effective preconditioners. Since RAS is asymmetric, it is often used with GMRES, an iterative method for solving asymmetric systems.

Besides additive Schwarz, researchers have developed other domain decomposition methods, such as balancing domain decomposition [Mandel 1993], Schur complement-based domain decomposition [Haase et al. 1991; Li and Saad 2017] and finite element tearing and interconnect (FETI) [Farhat et al. 2001; Farhat and Roux 1991]. In computer graphics, researchers have also studied domain decomposition for deformable body simulation [Li et al. 2019], especially in a reduced space [Barbič and Zhao 2011; Kim and James 2011; Wu et al. 2015; Yang et al. 2013]. Broadly speaking, these techniques can be viewed as FETI, which couples independently solved domains by Neumann boundaries. Recently, Wang [2021] used an overlapping AS method to simulate high-resolution grid cloth meshes.

An interesting idea is to combine multilevel and domain decomposition methods together, so as to achieve even higher scalability with respect to problem size. This idea is not new and many existing domain decomposition methods have already considered adopting corrections at a coarser level. What makes additive Schwarz special is that it works flexibly not only at two levels [Edwards and Bridson 2015], but at multiple levels [Dryja et al. 1996; Dryja and Widlund 1991; Zhang 1992]. As a result, a multilevel additive Schwarz (MAS) preconditioner can more effectively fix the error associated with inter-domain connections [Dolean et al. 2015]. This inspires us to develop a new MAS preconditioner in this work, by using small and parallelizable domains on a GPU.

3 BACKGROUND

Without loss of generality, we consider cloth and deformable body dynamics integrated by the implicit Euler method as an unconstrained nonlinear optimization problem:

$$\mathbf{q} = \arg\min_{\mathbf{q}} F(\mathbf{q}), \tag{2}$$

where $\mathbf{q} \in \mathbb{R}^{3N}$ is the state vector of N nodes (or called vertices) to be solved for the next update and $F(\mathbf{q})$ is the objective function. As shown in [Martin et al. 2011], $F(\mathbf{q})$ contains a momentum term and a potential term. To handle collisional contacts, it can include a repulsive potential term [Li et al. 2020; Wu et al. 2020] as well.

3.1 Preconditioning

One way to solve the problem in Eq. 2 is to apply Newton's method. In every Newton step, this method solves a linear system: $\mathbf{Ax} = \mathbf{b}$, in which $\mathbf{A} = \partial^2 F/\partial \mathbf{q}^2 \in \mathbb{R}^{3N\times 3N}$ and $\mathbf{b} = -\left(\partial F/\partial \mathbf{q}\right)^{\mathsf{T}} \in \mathbb{R}^{3N}$ are the Hessian and the negative gradient of F at the current state $\mathbf{q}$, and $\mathbf{x}$ provides an update to it: $\mathbf{q} \leftarrow \mathbf{q} + \mathbf{x}$. We assume that $\mathbf{A}$ is positive definite. If it is not, we can still enforce it through per-element correction [Choi and Ko 2002; Teran et al. 2005], which turns the method into a (broadly defined) quasi-Newton method.

A popular way of solving a linear system is to use the preconditioned conjugate gradient (PCG) method. The performance of the PCG method depends heavily on the preconditioner $\mathbf{M}$, a symmetric positive definite matrix. Intuitively, we can view preconditioning as converting the original system into a better conditioned one: $\mathbf{M}^{-1}\mathbf{Ax} = \mathbf{M}^{-1}\mathbf{b}$. Ideally, the use of $\mathbf{M}$ should significantly improve the convergence rate, while introduce little computational overhead in every iteration.

Preconditioning is also applicable to many nonlinear iterative methods [Wang and Yang 2016] solving Eq. 2 directly, such as gradient descent, nonlinear PCG and L-BFGS. Similar to PCG, these methods need a suitable $\mathbf{M}$ to achieve fast convergence rates with low per-iteration costs.

3.2 Multilevel Additive Schwarz (MAS)

Let $\{\Omega_d\}$ be a set of domains covering all of the nodes Ω, such that $\Omega = \bigcup \Omega_d$, and $\mathbf{S}_d \in \mathbb{R}^{3N_d\times 3N}$ be the selection matrix of domain d that pulls the N_d nodes in Ω_d out of the N nodes in Ω. We define an additive Schwarz (AS) preconditioner as:

$$\mathbf{M}_{(0)}^{-1} = \sum_d \mathbf{S}_d^{\mathsf{T}} \mathbf{M}_{d,(0)}^{-1} \mathbf{S}_d, \tag{3}$$

where $\mathbf{M}_{d,(0)} \cong \mathbf{S}_d \mathbf{A} \mathbf{S}_d^{\mathsf{T}} \in \mathbb{R}^{3N_d\times 3N_d}$ is a symmetric positive definite sub-matrix defined within domain d. An AS preconditioner can also work as a standalone stationary iterative solver. Frommer and Szyld [1999] pointed out that such a solver is guaranteed to converge, if the domains overlap and $\mathbf{A}$ is symmetric positive definite. In a special case when the domains do not overlap and $\mathbf{M}_{d,(0)} = \mathbf{S}_d \mathbf{A} \mathbf{S}_d^{\mathsf{T}}$, the AS solver is equivalent to the block Jacobi method and it converges as long as $\mathbf{A}$ is block diagonal dominant.

To improve the effectiveness of AS preconditioning in dealing with large systems, researchers have considered to construct a two-level AS preconditioner as:

$$\mathbf{M}_{\mathrm{TAS}}^{-1} = \mathbf{M}_{(0)}^{-1} + \mathbf{C}_{(1)}^{\mathsf{T}} \mathbf{M}_{(1)}^{-1} \mathbf{C}_{(1)}, \tag{4}$$

where $\mathbf{C}_{(1)} \in \mathbb{R}^{3N_{(1)}\times 3N}$ is a coarsener matrix and $\mathbf{M}_{(1)}$ is a single-level additive Schwarz preconditioner built for the coarsened system matrix $\mathbf{A}_{(1)} = \mathbf{C}_{(1)} \mathbf{A} \mathbf{C}_{(1)}^{\mathsf{T}}$. The rightmost part of Eq. 4 is commonly known as coarse space correction. If we coarsen $\mathbf{M}_{(0)}$ more, we can add more corrections and formulate a MAS preconditioner [Dolean et al. 2015] as:

$$\mathbf{M}_{\mathrm{MAS}}^{-1} = \mathbf{M}_{(0)}^{-1} + \sum_{l=1}^{L} \mathbf{C}_{(l)}^{\mathsf{T}} \mathbf{M}_{(l)}^{-1} \mathbf{C}_{(l)}, \tag{5}$$

in which $\mathbf{C}_{(l)} \in \mathbb{R}^{3N_{(l)}\times 3N}$ is the coarsener at level l. Unlike the coarseners used in other multilevel methods, the coarseners in Eq. 5 establish maps from level 0 directly to all of the levels. Therefore, hierarchical coarse space construction is not necessary, but often helpful in practice. Eq. 5 also suggests that the preconditioning process can be parallelized at all of the levels, rather than being sequenced in any specific order.

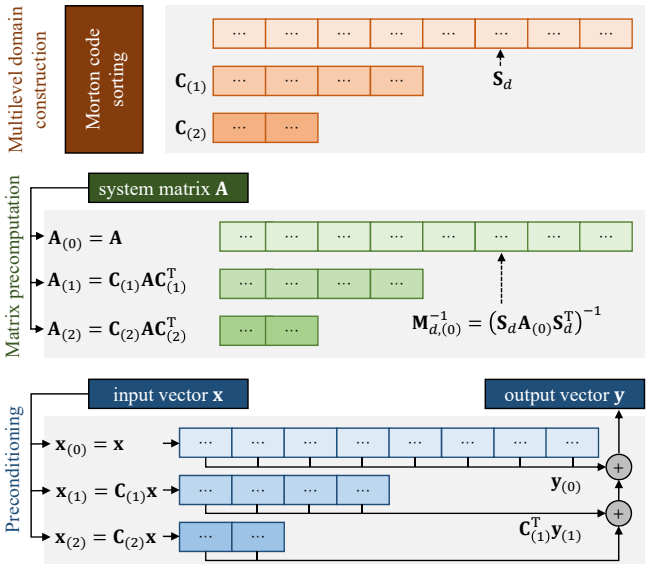


Fig. 3. The algorithmic pipeline. The pipeline of our MAS preconditioner contains three stages. We explore its parallelizability for fast and accurate simulation on a GPU. Here each block represents a domain.

4 OVERVIEW

To begin with, we would like to discuss the key choices in the development of our preconditioner under the multilevel additive Schwarz (MAS) framework. First, we choose not to use overlapping domains. When the domain size stays the same, the use of overlapping domains allows a preconditioner to converge faster, but it also increases the number of domains and it is unfriendly with our GPU-based domain partitioning approach in Subsection 5.1. Second, we avoid asymmetric variants, known as restricted additive Schwarz, and stick to the symmetric positive definite version outlined in Subsection 3.2. This ensures that our preconditioner is applicable to many linear and nonlinear solvers. Finally, we propose to adopt the Nicolaides' coarse space [Nicolaides 1987], which can be viewed simply as aggregating nodes into a supernode. Compared with a spectral coarse space [Spillane and Rixen 2013; Willems 2013], the Nicolaides' coarse space is more compatible with GPU implementation and more convenient to construct on the fly, as discussed in Subsection 5.2.

Based on the aforementioned choices, we formulate the algorithmic pipeline of our preconditioner in three stages as Fig. 3 shows. At the very beginning, the algorithm sorts all of the nodes by their Morton codes in the multilevel domain construction stage (in Section 5). Once the nodes are sorted, domain partitioning and coarse space construction become straightforward linear segmentation processes. In the sub-matrix inverse precomputation stage (in Section 6),

the algorithm calculates the coarsened system matrix $\mathbf{A}_{(l)}$ at each level and directly computes the sub-matrix inverse of each domain: $\mathbf{M}_{d,(l)}^{-1} = \left(\mathbf{S}_d \mathbf{A}_{(l)} \mathbf{S}_d^{\mathsf{T}}\right)^{-1}$. Finally at runtime, the preconditioning process (in Section 7) executes three steps in every call: distributing the input $\mathbf{x}$ to every level, performing the matrix-vector product in each domain and gathering the sub-vectors into the output $\mathbf{y}$.

Our preconditioner is highly parallelizable thanks to the MAS framework. Its two most expensive steps, i.e., the sub-matrix inverse step in the precomputation stage and the matrix-vector multiplication step in the preconditioning stage, can be well parallelized for all of the domains across different levels. This is crucial to the overall performance of our preconditioner, as shown in Section 8.

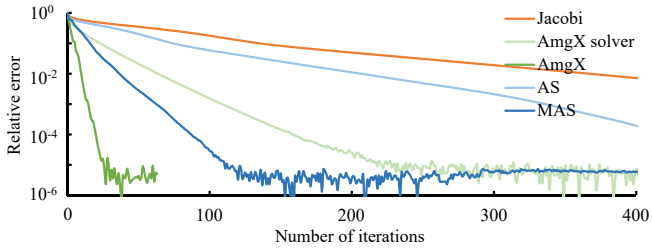
4.1 Why Not Factorization

The reason we compute the inverse of the sub-matrix, rather than its Cholesky or LDLT factorization, is because a triangular solver is difficult to parallelize. While we can still assign one thread to each domain, we cannot easily reach parallelization capacity unless the problem size is large (with at least 500K vertices). Meanwhile, no parallelization within a domain means the preconditioning stage must read the factorization matrix twice from the global memory, which greatly downgrades the preconditioner's performance.

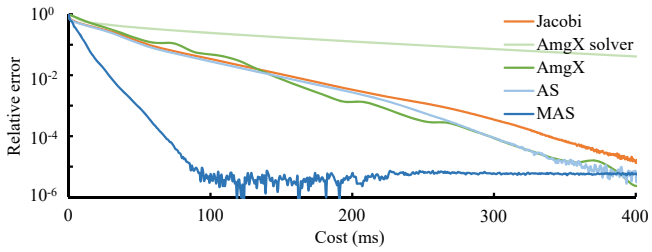
Matrix inverse computation is known for the numerical stability issue caused by floating point precision. By default, our preconditioner uses single-precision floating point accuracy, but it barely suffers from numerical stability in our experiment for two reasons. First, the sub-matrix is relatively small and error accumulation is limited. Second, as inter-domain connections being dropped off, the sub-matrix becomes much better conditioned than the whole system matrix. We note that our preconditioner can still experience the numerical stability issue in extremely ill-conditioned cases irrelevant to simulation, such as those provided in [Chen et al. 2021]. Therefore, the preconditioner is not supposed to be a general-purpose one for arbitrary problems.

4.2 Cloth and Deformable Body Simulation

While our MAS preconditioner is applicable to a wide range of cloth and deformable body simulators, we would like to describe our in-house ones here for reference purposes. Our cloth simulator handles unstructured triangle meshes. It uses the co-rotational linear model [Müller et al. 2005] for planar elasticity and the quadratic bending model [Bergou et al. 2006] for bending elasticity. Meanwhile, our deformable body simulator handles tetrahedral meshes and it supports multiple elastic models, including projective dynamics [Bouaziz et al. 2014], the co-rotational FEM model and hyperelastic models. To handle collisions, our simulators use spatial hashing [Tang et al. 2018] to detect colliding element pairs and integrate their repulsive potentials into the optimization objective in Eq. 2. To handle frictions, our simulators adopt both the impulse-based method [Bridson et al. 2002] and the adhesive method [Wu et al. 2020], the latter of which is particularly useful for static frictions. We note that our simulators handle collisions and frictions fast, but not safely nor accurately. They can be strengthened by applying safer and more accurate methods, but that is irrelevant to the speedup provided by our preconditioner in this work.



(a) The convergence rate



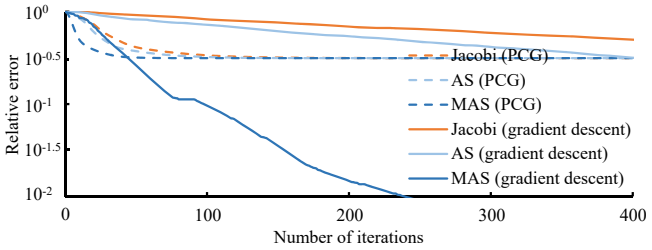
(b) The convergence speed (GPU)

Fig. 4. The performances of preconditioners with the PCG solver and the performance of the AmgX solver on a GPU. While the AmgX preconditioner has the fastest convergence rate, the MAS preconditioner achieves the highest convergence speed. Here we define the relative error of the result $\mathbf{x}^{[k]}$ in the k-th iteration as $\left(E(\mathbf{x}^{[k]}) - E(\mathbf{x}^*)\right)/\left(E(\mathbf{x}^{[0]}) - E(\mathbf{x}^*)\right)$, in which E is the quadratic objective of the linear system and $\mathbf{x}^*$ is the exact solution obtained after many iterations. By default, we use the T-shirt example for evaluation in this paper. Note that the curves start to oscillate below 10^{-5}, due to numerical errors in residual vectors and objective evaluation.

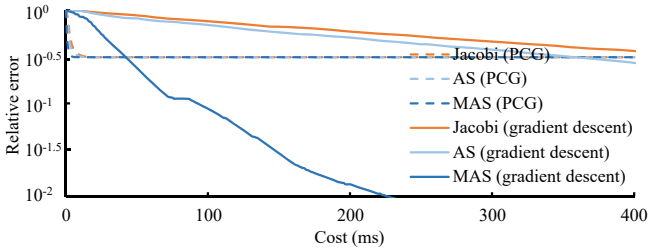
A critical problem is how to represent the large, sparse system matrix $\mathbf{A}$ involved in simulation. Our answer to this question is a hybrid matrix representation: the static ELLPACK format [Bell and Garland 2008] for storing the total of the mass matrix and the tangent stiffness matrix, and an extra array for storing contact node pairs and their contributions to $\mathbf{A}$. Given this representation, we can handle dynamic contacts over time by modifying the node pairs in the array. To perform matrix-vector multiplication as needed by many solvers, we first calculate the products with the ELLPACK matrix and the array matrix separately, and then sum them up.

4.3 Preconditioning in Simulation

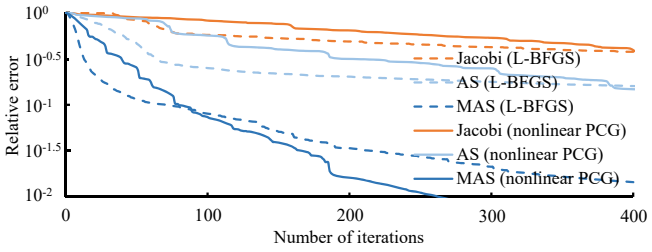
Our MAS preconditioner can be used by simulators in many ways. Perhaps the most straightforward way is to use it together with the PCG solver and solve the linear system of a single Newton iteration in every time step [Baraff and Witkin 1998]. Fig. 4 compares the performances of multiple preconditioners with the PCG solver and the performance of the multigrid AmgX solver [Naumov et al. 2015] on a GPU. It shows that our MAS preconditioner is approximately four times faster than the others when reaching the same convergence goal, thanks to a high convergence rate and a low runtime preconditioning cost. What is not illustrated in Fig. 4 is that our preconditioner also has a low construction/precomputation overhead: only 6.267ms. In comparison, the AmgX preconditioner/solver needs 161.7ms/176.6ms to preprocess the system, respectively.



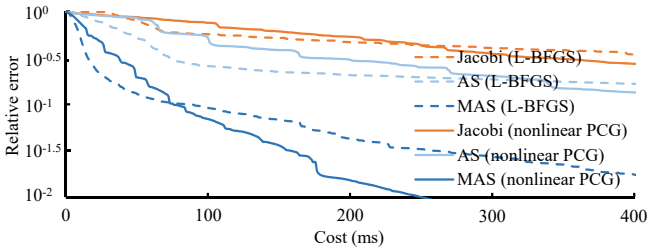
(a) The convergence rate



(b) The convergence speed (GPU)



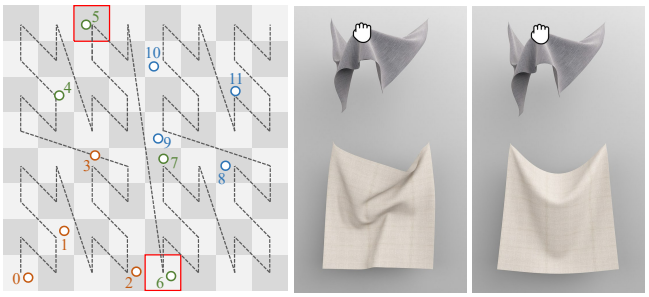
(c) The convergence rate



(d) The convergence speed (GPU)

Fig. 5. The performances of various preconditioners with GPU-based nonlinear solvers. This figure shows that our MAS preconditioner provides convergence improvements to all of these nonlinear solvers. Here we define the relative error of the state $\mathbf{q}^{[k]}$ in the k-th iteration as $(F(\mathbf{q}^{[k]}) - F(\mathbf{q}^*))/(F(\mathbf{q}^{[0]}) - F(\mathbf{q}^*))$, in which F is the nonlinear simulation objective and $\mathbf{q}^*$ is the exact solution.

When the computational budget is tight, it is efficient to just solve the linear system of a single Newton iteration by (linear) PCG as shown in Fig. 5b, thanks to small matrix evaluation and precomputation costs. But for accurate simulation, especially if the time step is large, we cannot overlook the nonlinearity of the optimization objective and we should apply more Newton iterations or switch to use other nonlinear solvers. We can apply our preconditioner to



(a) Sorted Nodes by Morton codes (b) With artifact (c) Without artifact

Fig. 6. A 2D example illustrating the use of Morton codes in domain partitioning and coarse space construction. When we use a uniform supernode size ($S = 4$) to construct the coarse space, two disjoint nodes may be grouped into the same supernode as shown in (a). As a result, user interaction with one cloth patch can falsely disturb another, as (b) shows.

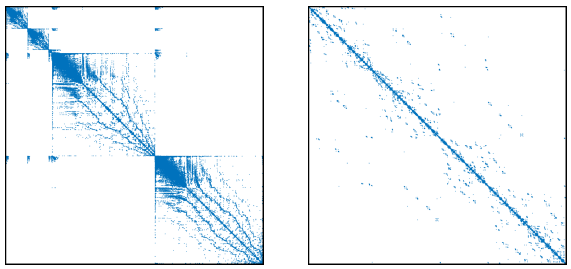
improve the convergence of other nonlinear solvers as well, such as accelerated gradient descent [Wang and Yang 2016], L-BFGS and nonlinear PCG. Since our preconditioner relies on the system matrix, we must redo MAS precomputation once the system matrix is updated. To prevent this from greatly affecting the simulation performance, we choose to redo MAS precomputation once every eight nonlinear iterations. Fig. 5 demonstrates the effectiveness of our preconditioner with nonlinear solvers in the "Schwarz" example and it confirms with [Wang and Yang 2016] that accelerated gradient descent, L-BFGS and nonlinear PCG have similar performances. In practice, we give our preference to an inexact Newton method with a fixed number of inner PCG iterations per Newton step, due to its close relationship with a PCG solver.

5 MULTILEVEL DOMAIN CONSTRUCTION

In the very first multilevel domain construction stage, the algorithm is responsible for two tasks: partitioning nodes into domains, i.e., setting up the selection matrices $\{\mathbf{S}_d\}$, and constructing the coarse space, i.e., creating the coarseners $\{\mathbf{C}_{(l)}\}$. We formulate this stage by symbolically analyzing the system matrix.

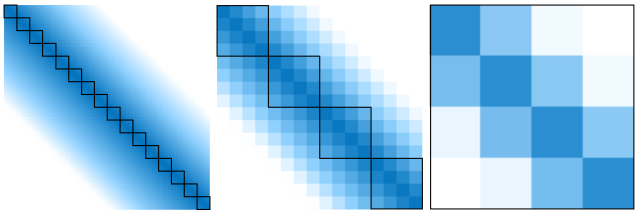
5.1 Domain Partitioning

If we view every node as a 3×3 diagonal block and every node-node connection as an off-diagonal block in the system matrix, we can then treat an additive Schwarz preconditioner as an approximation to the system matrix, with all of the node-node connections between two different domains ignored. The key question is: *how can we minimize such inter-domain connections, so that the preconditioner approximates the matrix well?* A naïve idea is to create a domain as a cluster of nodes in their reference states, as in [Wang 2021]. But this idea fails to consider connection changes caused by contacts and fractures. Inspired by linear bounding volume hierarchy [Lauterbach et al. 2009], we choose to sort the nodes by their Morton codes first. Specifically, we calculate the axis-aligned bounding box of the nodes, and divide the box into $2^{20} \times 2^{20} \times 2^{20}$ cells. We then obtain the 60-bit Morton code of each node by simply interleaving the bits of its cell indices. After we sort the nodes by Morton codes, we achieve



(a) Before sorting (b) After sorting

Fig. 7. The sparsity pattern of a 779K×779K matrix with 30.2M nonzero entries. Before we sort the nodes by their Morton codes, off-diagonal blocks are scattered as shown in (a). After we sort the nodes, they become closer to the diagonal line, as shown in (b). This practice allows our preconditioner to be highly effective. This figure is based on the T-shirt example.



(a) Level 0 (b) Level 1 (c) Level 2

Fig. 8. A matrix pattern and the MAS preconditioner applied to it. In this figure, every pixel represents a node or a supernode, and every dark frame represents a domain with four nodes (or supernodes) at one level. The pixel intensity illustrates the magnitude of a non-zero node (or supernode).

better spatial locality among the nodes, as a 2D example shows in Fig. 6a. Therefore, we can partition them immediately in a linear fashion. Intuitively, as spatial locality improves, the likelihood of a connection between two distant nodes in the sorted array drops. As a result, off-diagonal blocks should be closer to the diagonal of the system matrix as illustrated in Fig. 7b, making the preconditioner a better approximation to the matrix. Without Morton code sorting, our preconditioner exhibits an extremely low convergence rate when it directly handles the matrix in Fig. 7a.

In our implementation, we use radix sort for sorting. Because of its high performance on a GPU, our approach can work at runtime based on node positions in the current state. This allows our partitioning to correctly address contacts and fractures among spatially close but topologically distant nodes, which evolve over time.

5.2 Coarse Space Construction

Thanks to the spatial locality achieved by Morton code sorting, we propose to construct a Nicolaides' coarse space by grouping nodes (or supernodes) sequentially and hierarchically into supernodes. Fig. 8 illustrates the concept of our coarse space, when both the domain size M and the supernode size S are equal to four, i.e., four supernodes at level l grouped into the same supernode at level $l+1$. This coarse space provides an effective coverage of the whole system matrix, by placing many off-diagonals in the domains at finer levels while only a few off-diagonals in the domains at coarser levels.

The simplicity of our coarse space enables us to represent it in a compact form as $\mathtt{map}_{l\to l+1}[i]$, i.e., a coarsening map that converts the index of every supernode i at level l to the index of its parent supernode at level $l+1$. We collapse $\{\mathtt{map}_{l\to l+1}\}$ to compute $\mathtt{map}_{(l)}$, the coarsening map from level 0 directly to every level l. Using these maps, we can easily coarsen a vector by atomic additions and prolongate a coarsened vector by retrieving multiple copies of data on a GPU. We can also easily coarsen the system matrix $\mathbf{A}$ to $\mathbf{A}_{(l)}$, which will be discussed later in Subsection 6.2.

By default, we prefer the supernode size S to be equal to the domain size M, so that a domain naturally becomes a supernode at the coarser level. But a naïve implementation of this idea suffers from false coupling artifacts, commonly found between two disjoint bodies as Fig. 6b shows. On the surface, they are caused by the residual error when the number of iterations is not sufficiently large; but fundamentally, the reason is because two adjacent nodes in the same supernode may be well apart and the search direction would fail to treat them separately, as shown in Fig. 6a. To fix this problem, one solution is to check the Euclidean distance between two adjacent nodes (or the distance between the bounding boxes of two adjacent supernodes), and use that as a metric to decide if it is an acceptable split for a supernode at the coarser level. But this solution needs proper distance thresholds at different levels, which are often difficult to find in practice. Our solution instead is to use node-node connections. Specifically, two nodes at level 0 are connected if they are linked by an off-diagonal block in the system matrix, due to topological proximity or contact. According to these connections, we check every domain-sized supernode candidate at level 1 and split it into multiple supernodes, if they are not actually connected. We then construct the coarsening map at level 1. After that, we coarsen node-node connections to level 1 and repeat the process to construct more maps at coarser levels hierarchically. This solution removes false coupling artifacts as shown in Fig. 6c.

5.3 A Skipping Approach

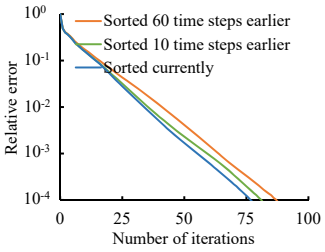
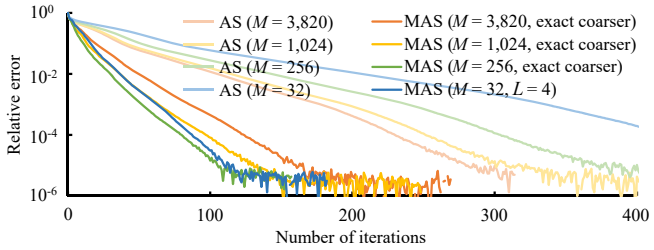
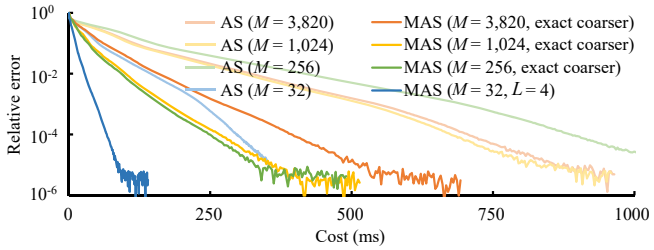


Fig. 9. The convergence of PCG with our MAS preconditioner, when it uses nodes sorted at different times.

The computational cost of the whole multilevel domain construction stage is low but still not negligible. Since it starts from Morton code sorting and Morton codes rely on node positions, an interesting question is whether we can skip the construction stage in a number of time steps, assuming that object movement within a single time step is small. Fig. 9 shows that the convergence of PCG with our preconditioner deteriorates, as it starts to use nodes sorted earlier in time. On the other hand, even if it uses the nodes sorted 60 time steps earlier, the convergence is still not terribly bad. This is because spatial locality among topologically close nodes is still preserved in every set of sorted nodes, regardless of when they are sorted. What is not preserved is spatial locality among contact node



(a) The convergence rate



(b) The convergence speed (GPU)

Fig. 10. The performances of AS and MAS preconditioners with different choices of M. There exist two ways to reduce the error associated with inter-domain connections: using a large M as (a) shows, and using coarse space corrections. Overall, we think it is more efficient to use a small M with coarse space corrections, so that our MAS preconditioner can be highly compatible with GPU acceleration.

pairs, which vary over time. Therefore, we anticipate the convergence problem associated with our skipping approach to become severer in collision-intensive scenes. To play safe, we choose to run the multilevel domain construction stage once every ten time steps, and this effectively reduces its overhead from six percent of the total solver cost to under one percent.

6 MATRIX PRECOMPUTATION

Before we are able to perform runtime preconditioning in a solver, we must obtain the sub-matrix $\mathbf{A}_{d,(l)} = \mathbf{S}_d \mathbf{A}_{(l)} \mathbf{S}_d^{\mathsf{T}}$ of each domain d at every level l, and then formulate the per-domain preconditioner $\mathbf{M}_{d,(l)}^{-1}$. To begin with, we will discuss the choice of the domain size M and how that affects the formulation of our preconditioner in Subsection 6.1. In Subsection 6.2 and 6.3, we will then present fast algorithms to compute and maintain the sub-matrix inverse of each domain, as the precomputation stage.

6.1 The Choice of the Domain Size M

Perhaps the most significant choice involved in the design of our preconditioner is the choice of the domain size M. In computer graphics and computational mathematics literature, most of the domain decomposition methods prefer using large domains over small domains. The use of large domains reduces the error associated with inter-domain connections, at the cost of more difficulty in solving each sub-problem. Fig. 10a justifies this idea by showing that an AS preconditioner achieves a faster convergence rate as the domain size M grows from 256 to 3,820. Since it becomes less practical to solve each domain exactly when M gets large, we define the preconditioning process of each domain as eight multi-color Gauss-Seidel iterations, for $M > 32$.

But the situation becomes different once we begin to consider parallel computing and coarse space corrections. If there are many small domains, we can explore their parallelization on a GPU and even achieve $\mathbf{M}_{d,(l)}^{-1} = \mathbf{A}_{d,(l)}^{-1}$ by computing each sub-matrix inverse directly. Meanwhile, we can also effectively lessen the inter-domain connection issue by coarse space corrections. In fact, if a MAS preconditioner is able to solve its coarse-level problem exactly, it would converge even faster as M gets smaller as Fig. 10 shows. Essentially, we need a suitable domain size M, which should be sufficiently small to take the aforementioned benefits, while still be large enough to minimize the inter-domain connection error. In this work, we set $M = 32$. This number provides the right sub-matrix size for our algorithm in Subsection 6.2 to utilize the shared memory in the computation of the sub-matrix inverse. Fig. 10b shows that our MAS preconditioner with $M = 32$ and $L = 4$ outperforms others, even though it does not solve each coarse problem exactly. Here L is the number of levels.

ALGORITHM 1: Matrix Precomputation

Input: System matrix $\mathbf{A}$ and coarsening maps $\{\mathtt{map}_{(l)}\}$
Result: Sub-matrix inverses $\{\mathbf{M}_{d,(l)}^{-1}\}$
$\{\mathbf{A}_{d,(l)}\} \leftarrow \mathbf{0}$;
foreach node i **and** node j **and** level l **do**
  if $\lfloor \mathtt{map}_{(l)}[i]/32 \rfloor = \lfloor \mathtt{map}_{(l)}[j]/32 \rfloor$ **then**
    $d \leftarrow \lfloor \mathtt{map}_{(l)}[i]/32 \rfloor$;
    $i' \leftarrow \mathtt{map}_{(l)}[i] - 32d$;
    $j' \leftarrow \mathtt{map}_{(l)}[j] - 32d$;
    $\mathbf{A}_{d,(l)}[i', j'] \leftarrow \mathbf{A}_{d,(l)}[i', j'] + \mathbf{A}[i, j]$;
  end
end
foreach domain d **and** level l **do**
  $\{\mathbf{L}_{d,(l)}^{-1}, \mathbf{D}_{d,(l)}\} \leftarrow \mathtt{Gauss\_Jordan\_Elimination}(\mathbf{A}_{d,(l)})$;
end
foreach domain d **and** level l **do**
  $\mathbf{M}_{d,(l)}^{-1} \leftarrow \mathbf{L}_{d,(l)}^{-\mathsf{T}} \mathbf{D}_{d,(l)}^{-1} \mathbf{L}_{d,(l)}^{-1}$;
end

6.2 Per-Domain Sub-Matrix Inverse Calculation

Alg. 1 provides the pseudo code that precomputes the submatrix inverse of every domain in three steps. In the first step, we check every 3×3 system matrix block $\mathbf{A}[i, j]$ and add it into the 96×96 sub-matrix $\mathbf{A}_{d,(l)}$, if node i and node j are mapped to the same domain d at level l. In the second step, we apply Gauss-Jordan elimination to row-reduce the 96×192 matrix $\left[\mathbf{A}_{d,(l)} \middle| \mathbf{I}\right]$ into a row echelon form $\left[\mathbf{U}_{d,(l)} \middle| \mathbf{L}_{d,(l)}^{-1}\right]$, such that $\mathbf{L}_{d,(l)} \mathbf{U}_{d,(l)} = \mathbf{A}_{d,(l)}$. To decrease the memory footprint, we apply elimination

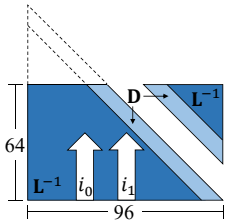


Fig. 11. A compact format of a lower triangular matrix. We use this format to do matrix multiplication in the shared memory.

to the 96×96 matrix $\mathbf{A}_{d,(l)}$ directly, and overwrite it by $\mathbf{U}_{d,(l)}$ and $\mathbf{L}^{-1}_{d,(l)}$. Since $\mathbf{A}_{d,(l)}$ is symmetric, $\mathbf{U}_{d,(l)} = \mathbf{D}_{d,(l)}\mathbf{L}^{\mathsf{T}}_{d,(l)}$ and $\mathbf{D}_{d,(l)}$ is the diagonal of $\mathbf{U}_{d,(l)}$. The output is a lower triangular matrix stored in a compact format, as Fig. 11 shows. Finally, in the third step, we calculate $\mathbf{L}^{-\mathsf{T}}_{d,(l)}\mathbf{D}^{-1}_{d,(l)}\mathbf{L}^{-1}_{d,(l)}$ by visiting every two columns i_0 and i_1 in parallel. Thanks to this format, we can perform the third step completely in the GPU shared memory. The final output is a per-domain submatrix inverse $\mathbf{M}^{-1}_{d,(l)} = \mathbf{A}^{-1}_{d,(l)}$, stored back to the global memory in an alternative compact format to be discussed in Subsection 7.1.

The reason we do not use Gauss-Jordan elimination to obtain $\mathbf{A}^{-1}_{d,(l)}$ directly is because doing so requires two row-reductions. Every row-reduction involves $32\sum_{i=1}^{31} i \approx \frac{1}{2}32^3$ node-node matrix multiplications and the rows must be visited sequentially. In comparison, the $\mathbf{L}^{-\mathsf{T}}_{d,(l)}\mathbf{D}^{-1}_{d,(l)}\mathbf{L}^{-1}_{d,(l)}$ product uses $\sum_{i_0=1}^{32}\sum_{i_1=1}^{i_0} i_1 \approx \frac{1}{6}32^3$ node-node multiplications and it can be well parallelized. We also note that we do not store a lower triangular matrix linearly by collapsing all of the rows, because such a format is less efficient to visit as our experiment indicates, probably due to bank conflicts in the shared memory.

6.3 Selective Sub-Matrix Inverse Update

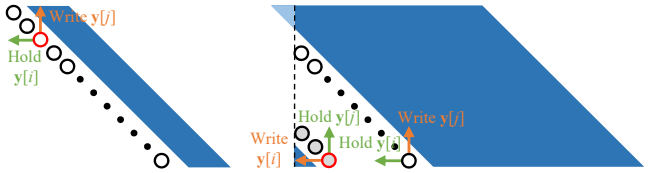
Since the evaluation of the tangent stiffness matrix can be expensive, researchers have also developed simulation techniques that temporarily or permanently skip stiffness matrix updates, such as quasi-Newton methods [Brown and Brune 2013; Wang and Yang 2016] and projective dynamics [Bouaziz et al. 2014]. These methods may still need to update the whole system matrix as in [Fratarcangeli et al. 2016], to more properly handle contacts over time. We make our preconditioner suitable for these methods by developing a selective matrix inverse update function. To do so, we use the changed contact pairs to detect the domains with changes and recalculate the matrix inverses of those domains only. In our experiment, when the stiffness matrix stays unchanged, this practice reduces the cost of matrix precomputation to be under five percent of its original cost.

7 RUNTIME PRECONDITIONING

Our runtime preconditioning process handles all of the domains at all of the levels in three steps: first, it coarsens the input vector into different copies at all of the levels; second, it performs preconditioning inside of each domain; and finally, it sums up the results at all levels into a joint output. Among the three steps, the second step is the most complex and expensive one. Since runtime preconditioning is executed in every iteration, we require it, especially its second step, to be fast.

7.1 Per-Domain Preconditioning

The preconditioning task inside of each domain is essentially matrix-vector multiplication by the submatrix inverse $\mathbf{A}^{-1}_{d,(l)}$, precomputed in Subsection 6.2. While the MAS framework allows us to easily parallelize these tasks for all of the domains at different levels, we still would like to know if there exists a faster way to perform matrix-vector multiplication here.



(a) Phase 1 (b) Phase 2

Fig. 12. The phases involved in a pass. Our symmetric-matrix-vector multiplication method parallelizes the 32 threads (shown as dots) in the same warp with balanced workload and zero write conflict.

Our basic idea is to halve the memory access by exploring matrix symmetry, so hopefully we can halve the computational time as well. But since our matrix is only 96×96, we cannot simply launch threads as blocks along matrix rows or columns like [Nath et al. 2011], or that would introduce too much load imbalance and write conflict. In the T-shirt example, the experiment shows our implementation of full-matrix-vector multiplication takes about 400μs, while a naïve implementation of symmetric-matrix-vector multiplication outlined above takes even more time, i.e., 802μs.

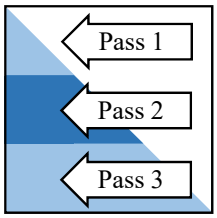
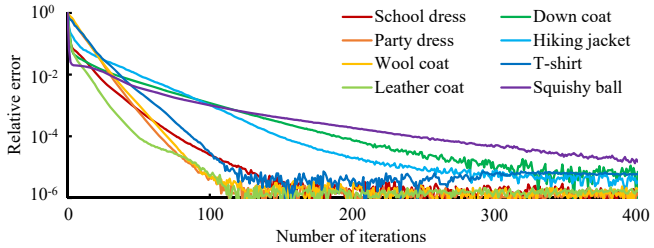


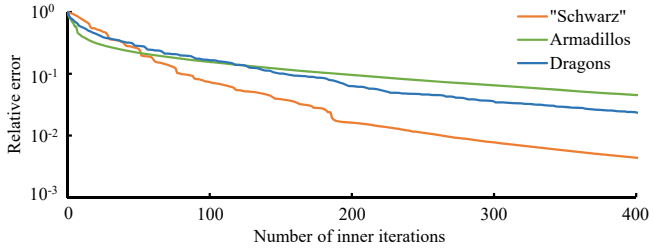
Fig. 13. The three passes in per-domain preconditioning.

Our solution is a novel symmetric-matrix-vector multiplication method with balanced workload and zero write conflict, as Fig. 12 and 13 show. In this approach, the multiplication process is performed through three passes, each of which covers 32 rows and contains three phases. Without loss of generality, let us consider the second pass in Fig. 12a. In Phase 1, 32 (white) threads in the same warp process 32 matrix rows backward from the diagonal. Let $\mathbf{x}$ and $\mathbf{y}$ be input and output vectors in the shared memory, and $\mathbf{M}[i, j]$ be the matrix entry currently processed by one thread. This thread calculates $\mathbf{M}[i, j]\mathbf{x}[i]$ and $\mathbf{M}[i, j]\mathbf{x}[j]$, and needs to add the two products into $\mathbf{y}[j]$ and $\mathbf{y}[i]$ respectively. But instead of writing both $\mathbf{y}[j]$ and $\mathbf{y}[i]$ in the shared memory immediately, the thread writes $\mathbf{y}[j]$ only while accumulates $\mathbf{M}[i, j]\mathbf{x}[j]$ in its own register. Since every thread writes different $\mathbf{y}[j]$, there is no write conflict. In Phase 2, a top (gray) thread reaches the left side of the matrix and begins to process matrix columns upward starting from the lower left corner, as Fig. 12b shows. This shifted thread uses a different write/hold strategy: it adds $\mathbf{M}[i, j]\mathbf{x}[j]$ into $\mathbf{y}[i]$ in the shared memory, while accumulates $\mathbf{M}[i, j]\mathbf{x}[i]$ in another register. Again, there is no write conflict among the threads. Finally in Phase 3, the method disables the shifted threads and finishes the last 16 matrix entries. After the three phases, the 32 threads write the values in their registers back to $\mathbf{y}$ in the shared memory, with no write conflict. The experiment shows that this method reduces the multiplication time to 220μs, almost half of the full-matrix-vector multiplication time.

According to the visiting order described by our multiplication method, we arrange the entries in $\mathbf{A}^{-1}_{d,(l)}$ to achieve optimal spatial locality. This is also the compact format of $\mathbf{A}^{-1}_{d,(l)}$ for the precomputation stage in Subsection 6.2 to output.

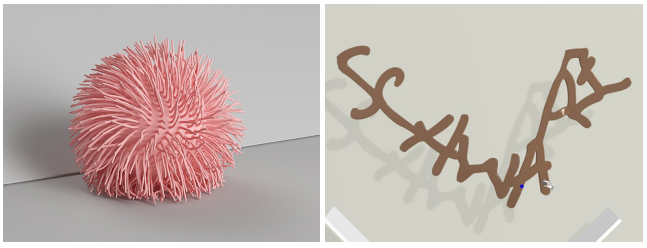


(a) The convergence of linearized examples



(b) The convergence of nonlinear examples

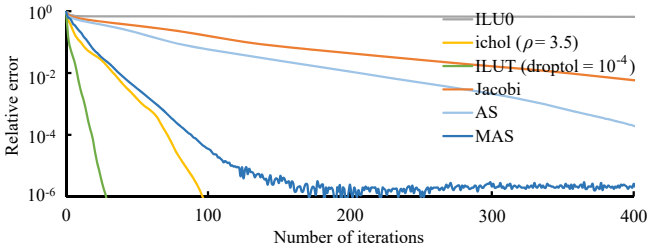
Fig. 14. The convergence of our simulator. This figure summarizes the convergence of our simulator in all of the examples, including both linearized (with one linear system per time step) and nonlinear ones. Note that their convergence rates are different, due to different relative error definitions.



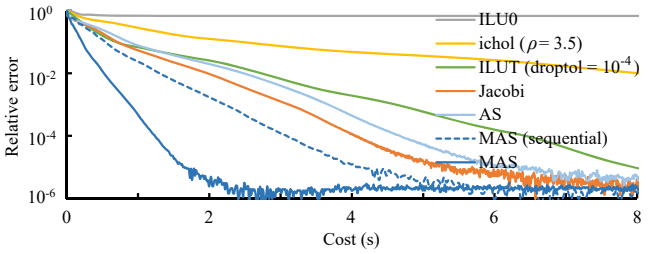
(a) Squishy ball (b) "Schwarz"

(c) Armadillos (d) Dragons

Fig. 15. The deformable body simulation examples. Our MAS preconditioner is compatible with a variety of nonlinear solvers in the simulation of these examples, ranging from 52K vertices to 707K vertices.



(a) The convergence rate



(b) The convergence speed (CPU)

Fig. 16. The performances of various preconditioners on a CPU. Although our preconditioner does not achieve the fastest convergence rate, its runtime performance is the highest as (b) shows, because of its low preconditioning cost. In this experiment, we test multiple parameter values and find $\rho = 3.5$ and droptol=10^{-4} to be the optimal ones for ichol and ILUT. Note that ichol does not always work at $\rho = 3.5$. In the rest of the paper, we choose other ρ values, if ichol fails at $\rho = 3.5$.

8 RESULTS AND DISCUSSIONS

We implement our MAS preconditioner with single-precision floating point accuracy for both GPUs and CPUs. Our GPU implementation uses CUDA 11.51 and our CPU implementation uses OpenMP and SIMD SSE. To perform fast multiplication with a large system matrix on a CPU, we adopt the two-step approach [Gkountouvas et al. 2013]. Table 1 provides the statistics and the performances of our examples, and Fig. 14 summarizes their convergence behaviors. Fig. 15 illustrates our deformable body simulation examples, including those simulated by nonlinear solvers. We evaluate the performances of our preconditioner in these examples on an NVIDIA GeForce RTX 3080 GPU and an Intel Core i7-9700 3.0GHz CPU. By default, we simulate linearized examples by the PCG solver, and we simulate nonlinear examples by the inexact Newton solver with 32 inner PCG iterations per Newton step. All of our examples use the same time step $\Delta t = 1/100$s, and fixed numbers of Newton steps and (inner) PCG iterations per time step, as shown in Table 1. In all of the examples, the ratio between material stiffness and collision repulsion stiffness is 1:1000.

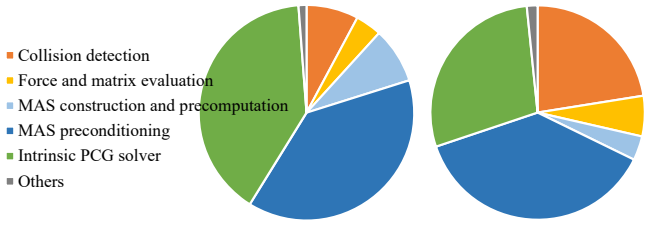
According to NVIDIA Nsight profiler, the memory SOL of our preconditioner in the runtime preconditioning stage is 92%, very close to the maximum theoretical throughout.

8.1 Performance Evaluation on a CPU

While our preconditioner is designed for a GPU, it is also effective on a CPU. In the following experiment, we compare the performance of our preconditioner with those of factorization-based preconditioners, including vanilla incomplete LU (ILU0) provided by Intel Math Kernel Library, multiscale incomplete Cholesky [Chen et al. 2021] and dual threshold incomplete LU (ILUT) [Saad 1994] provided by the Eigen library. The experiment shows that our preconditioner

Table 1. The statistics and the performances of our examples. We simulate linearized examples by using 64 to 96 PCG iterations to solve a single Newton step, and nonlinear examples by using 32 inner PCG iterations to solve each of the three Newton steps. This table summarizes the total computational costs spent in one time step. Here C_1, C_2 and C_{solver} are matrix precomputation, runtime preconditioning and intrinsic PCG costs per call.

Name	#Verts. (#Tri. or #Tet.)	Model type	Nonzero rate	Condition number	Const. cost (ms)	MAS-preconditioned solver (ms) #Newton $\times(C_1+$ #Inner $\times(C_2+C_{\text{solver}}))=C_{\text{total}}$	Total cost (ms)
School dress	96K (191K)	Linearized	0.0135%	2.64×10^7	0.206	$1\times(3.388+64\times(0.133+0.164))=22.427$	31.523
Party dress	76K (151K)	Linearized	0.0170%	1.52×10^6	0.158	$1\times(2.519+64\times(0.114+0.143))=18.967$	25.889
Wool coat	92K (182K)	Linearized	0.0141%	2.23×10^6	0.214	$1\times(2.918+64\times(0.129+0.152))=20.902$	27.013
Leather coat	140K (278K)	Linearized	0.0093%	8.96×10^6	0.230	$1\times(6.999+64\times(0.196+0.245))=35.223$	46.190
Down coat	467K (924K)	Linearized	0.0029%	2.75×10^8	0.638	$1\times(15.322+64\times(0.599+0.767))=102.746$	146.049
Hiking jacket	80K (158K)	Linearized	0.0165%	1.06×10^8	0.156	$1\times(3.065+64\times(0.122+0.149))=20.409$	26.069
T-shirt	260K (519K)	Linearized	0.0050%	3.34×10^6	0.319	$1\times(8.292+64\times(0.331+0.544))=64.292$	79.211
Squishy ball	707K (2.21M)	Linearized	0.0016%	8.78×10^7	1.212	$1\times(28.161+96\times(0.861+1.158))=221.985$	274.340
"Schwarz"	52K (222K)	Baraff-Witkin	0.0242%	1.47×10^6	0.222	$3\times(1.891+32\times(0.085+0.094))=22.857$	32.033
Armadillos	122K (474K)	neo-Hookean	0.0100%	2.13×10^9	0.287	$3\times(4.520+32\times(0.166+0.223))=50.904$	73.133
Dragons	125K (496K)	neo-Hookean	0.0098%	6.92×10^6	0.308	$3\times(6.411+32\times(0.171+0.295))=63.969$	90.142



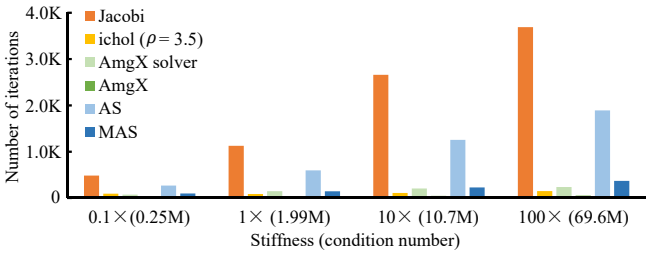
(a) GPU breakdown (b) CPU breakdown

Fig. 17. The breakdowns of the computational time spent by our simulator in the T-shirt example. This figure shows that runtime preconditioning and intrinsic PCG solver are the two bottlenecks, on both a GPU and a CPU.

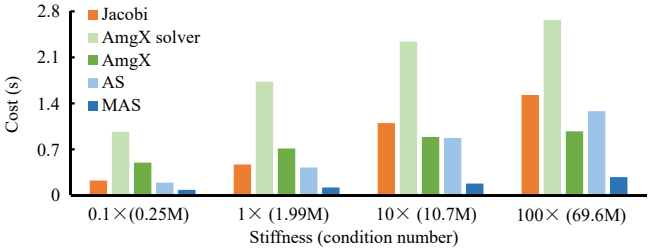
converges slower than ichol and ILUT in Fig. 16a, but it still outperforms them in Fig. 16b, thanks to its low runtime preconditioning cost. In fact, our preconditioner outperforms other preconditioners even without CPU parallelization. Another strength of our preconditioner is a low construction/precomputation overhead: 0.118s with parallelization or 0.549s without. In comparison, the factorization costs of ILU0, ichol and ILUT are 2.50s, 33.8s and 154s, while complete LU factorization by Intel MKL PARDISO takes 4.87s only. Fig. 16 also suggests that the convergence speedup gained by the use of an AS preconditioner does not outweight the cost of matrix-vector multiplication on a CPU. This partially explains why AS preconditioners were not popular in the past.

8.2 Breakdown Analysis

Fig. 17 shows the breakdowns of the computational time of one time step in the T-shirt example. Compared with other components, MAS preconditioning and intrinsic PCG solver are the two most expensive ones, due to the number of iterations needed for reaching the convergence. On a CPU, the portions contributed by the two components are smaller as Fig. 17b shows, since the other components are not well parallelized. We note that our simulator uses a simple repulsive collision method and solves a single linear system per time step in this example. If the simulator applies safe collision handling and nonlinear solvers instead, it should spend more time on collision handling, force/matrix evaluation and MAS precomputation.



(a) Stiffness vs. number of iterations

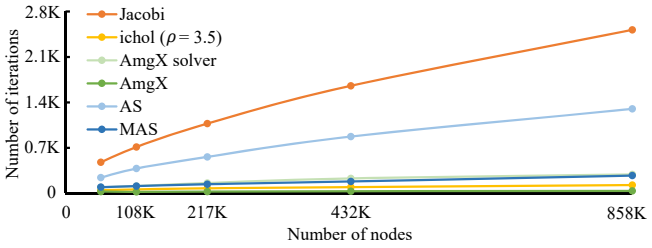


(b) Stiffness vs. solver cost (GPU)

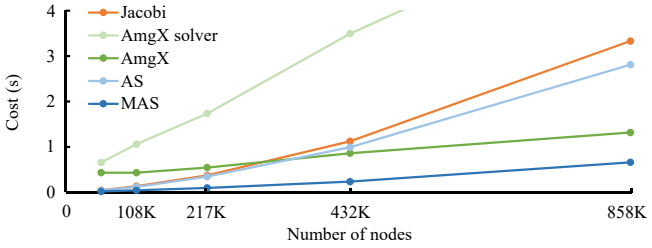
Fig. 18. The scalability of various preconditioners with respect to stiffness. This experiment demonstrates the high scalability of our MAS preconditioner with respect to stiffness, i.e., an 237% solver cost increase as stiffness parameters become 1,000 times larger.

8.3 Scalability with Respect to Stiffness

To evaluate the scalability of our MAS preconditioner with respect to stiffness, we increase stiffness parameters in the T-shirt example and record the performance of the PCG method with different preconditioners, as Fig. 18 shows. In this experiment, we set the convergence condition to be $\left\|\mathbf{b}-\mathbf{A}\mathbf{x}^{[k]}\right\|_\infty \le 5\times10^{-4}\left\|\mathbf{b}-\mathbf{A}\mathbf{x}^{[0]}\right\|_\infty$, where $\mathbf{x}^{[0]}$ is the initial state. The experiment shows the scalability of icho is the best: an approximately 60% cost increase as stiffness parameters grow from 0.1× to 100×. The scalability of our preconditioner is comparable to that of AmgX: an approximately 200% cost increase. In contrast, the solver costs associated with Jacobi and AS preconditioners increase by 600% to 700%. We note that the

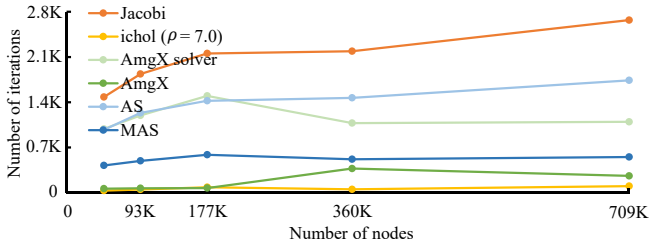


(a) Mesh resolution vs. number of iterations

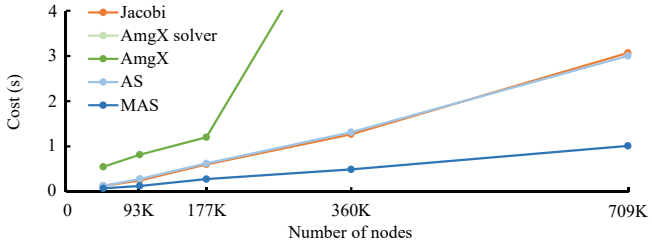


(b) Mesh resolution vs. solver cost (GPU)

Fig. 19. The scalability of various preconditioners with respect to mesh resolution in the T-shirt example. As the number of nodes becomes 16 times larger, the solver costs associated with Jacobi, AmgX, AS and MAS preconditioners become 76, 4, 69 and 35 times larger, respectively.



(a) Mesh resolution vs. number of iterations



(b) Mesh resolution vs. solver cost (GPU)

Fig. 20. The scalability of various preconditioners with respect to mesh resolution in the armadillo example. As the number of nodes becomes 15 times larger, the solver costs associated with Jacobi, AmgX, AS and MAS preconditioners become 24, 3, 23 and 14 times larger, respectively.

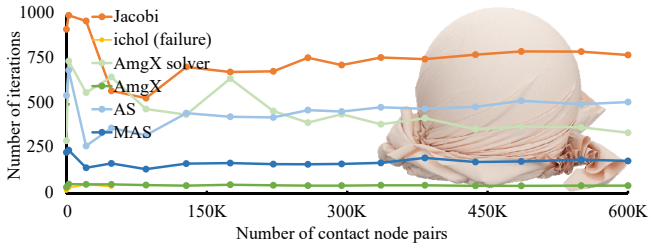
increase of stiffness parameters is not proportional to the increase of the condition number, once stiffness becomes sufficiently large. We also note that we do not include ichol in Fig. 18b, 19b, 20b and 21b, because it is CPU only.

8.4 Scalability with Respect to Mesh Resolution

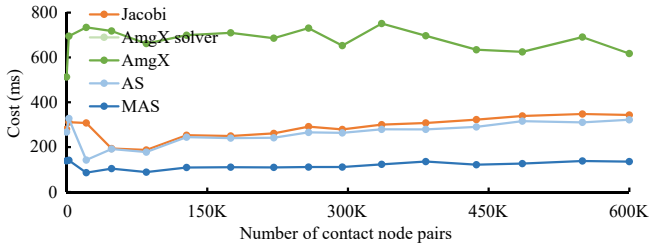
In the next experiment, we evaluate the scalability of our preconditioner with respect to mesh resolution in both cloth and deformable body examples, as shown in Fig. 19 and 20 respectively. As mesh resolution increases, both problem size and system stiffness increase. In this experiment, we use the same convergence condition as in Subsection 8.3. It shows that the scalability of our preconditioner with respect to mesh resolution is better than those of Jacobi and AS preconditioners, but worse than that of AmgX. Therefore, our preconditioner may be outperformed by the AmgX preconditioner in an example with multi-million nodes. But such an example is not our focus yet and it is possible to optimize our preconditioner for very high-resolution problems further.

8.5 Scalability with Respect to Collision Complexity

As collision complexity grows, the collision handling cost increases and the overall performance gain of using our preconditioner drops. Since more contact pairs introduce more off-diagonals in the system matrix, we also would like to know how collision complexity affects the solver cost directly. To do so, we design an example by draping a square cloth patch with 200K vertices onto a rotating sphere, during which the number of contact node pairs grows from 0 to 600K over time. As before, we use the convergence condition outlined in Subsection 8.3. Interestingly, Fig. 21 shows that the influence of collision complexity on the number of iterations and the solver cost



(a) Collision complexity vs. number of iterations



(b) Collision complexity vs. solver cost (GPU)

Fig. 21. The scalability of various preconditioners with respect to collision complexity. While the number of contact node pairs in a rotating sphere example increases from 0 to 600K, the performance of a PCG solver barley changes, regardless of its preconditioners.

is quite small, regardless of the preconditioners. We guess this is because the residual error associated with collision repulsion is in high frequency and can be quickly reduced by a PCG solver, even when using the Jacobi preconditioner.

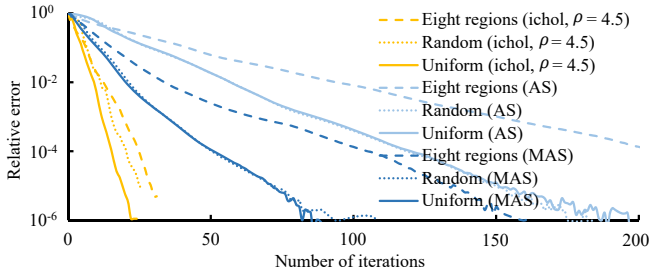


Fig. 23. The convergence of ichol, AS and MAS preconditioners in heterogeneous material cases. The figure shows that the convergence rate of our preconditioner is still acceptable when it handles a randomly heterogeneous armadillo model, but decreases notably when it handles a model made of eight stiffness regions shown in Fig. 22. This is a linearized example.

8.6 Effectiveness against Heterogeneity

Fig. 22. An armadillo model with different stiffness values in eight regions.

Similar to other MAS preconditioners, our preconditioner experiences a performance drop when it handles heterogeneous materials. To evaluate this problem, we create three armadillo models: a homogeneous one made of uniform stiffness, a randomly heterogeneous one made by assigning each tetrahedron with random stiffness, and another heterogeneous one with different stiffness parameters in eight regions shown in Fig. 22. The mean stiffness parameters of the three models are the same. Fig. 23 shows that the performance of our preconditioner barely changes when it handles the randomly heterogeneous model, but drops notably when it handles the eight-region model. Fundamentally, this is because stiff regions need more iterations to solve and the construction of our coarse space offers limited help.

8.7 Limitations

Under the MAS framework, our preconditioner shares limitations with other MAS preconditioners, such as incapability of running as a standalone solver and ineffectiveness against heterogeneous materials as discussed in Subsection 8.6. Since its scalability with respect to mesh resolution is limited as Subsection 8.4 shows, our preconditioner should be less effective against adaptively refined meshes as well. When our preconditioner works with nonlinear solvers, matrix evaluation and precomputation can introduce large computational costs, if the system matrix is updated in every nonlinear iteration. Our current solution in Subsection 4.3 is to delay the system matrix update by a fixed number of iterations, but a more plausible solution is to decide the matrix update based on node changes. In this paper, we design and evaluate the application of our preconditioner in cloth and deformable body simulation only. It is possible that our preconditioner can improve the convergence of linear/nonlinear solvers in other nodal systems, but we are uncertain how it handles general positive definite or nonlinear optimization problems.

9 CONCLUSIONS AND FUTURE WORK

In this paper, we demonstrate the power of MAS preconditioning and GPU computing in the development of an effective preconditioner for cloth and deformable body simulation applications. A distinctive difference between our preconditioner and previous ones is that ours uses sufficiently small domains, so that each domain can be directly solved with high-performance parallelism on a GPU. We believe this idea is suitable for more advanced multilevel domain decomposition techniques in the future.

Following this work, we would like to study how to incorporate matrix stiffness into coarse space construction, to make our preconditioner more effective against heterogeneous materials and adaptively refined meshes. We also would like to learn if there exists a convenient way of adopting spectral coarse space construction, to further improve our preconditioner's performance. Currently, our preconditioner is not well optimized for high-resolution problems and we plan to explore more advanced techniques for performance improvement, such as other triangular matrix representations [Charara et al. 2017] and fast symmetric sparse matrix-vector multiplication [Alappat et al. 2020]. Finally, we will investigate the use of our preconditioner in more complex simulators and other nodal systems, and develop its suitable variants if needed.

ACKNOWLEDGMENTS

We wish to thank the digital content team at Style3D for helping with rendering and scene creation. We also would like to thank NVIDIA for hardware support.

REFERENCES

Christie Alappat, Achim Basermann, Alan R. Bishop, Holger Fehske, Georg Hager, Olaf Schenk, Jonas Thies, and Gerhard Wellein. 2020. A Recursive Algebraic Coloring Technique for Hardware-Efficient Symmetric Sparse Matrix-Vector Multiplication. *ACM Trans. Parallel Comput.* 7, 3, Article 19 (jun 2020), 37 pages.

David Baraff and Andrew Witkin. 1998. Large Steps in Cloth Simulation. In *Proceedings of SIGGRAPH 98 (Computer Graphics Proceedings, Annual Conference Series)*, Eugene Fiume (Ed.). Association for Computing Machinery, New York, NY, USA, 43–54.

Jernej Barbič and Yili Zhao. 2011. Real-Time Large-Deformation Substructuring. *ACM Trans. Graph. (SIGGRAPH)* 30, 4, Article 91 (jul 2011), 8 pages.

Nathan Bell and Michael Garland. 2008. *Efficient Sparse Matrix-Vector Multiplication on CUDA.* NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation.

Miklos Bergou, Max Wardetzky, David Harmon, Denis Zorin, and Eitan Grinspun. 2006. A Quadratic Bending Model for Inextensible Surfaces. In *Proceedings of SGP* (Cagliari, Sardinia, Italy). 227–230.

Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. 2014. Projective Dynamics: Fusing Constraint Projections for Fast Simulation. *ACM Trans. Graph. (SIGGRAPH)* 33, 4, Article 154 (July 2014), 11 pages.

Robert Bridson, Ronald Fedkiw, and John Anderson. 2002. Robust Treatment of Collisions, Contact and Friction for Cloth Animation. *ACM Trans. Graph. (SIGGRAPH)* 21, 3 (July 2002), 594–603.

Jed Brown and Peter Brune. 2013. Low-Rank Quasi-Newton Updates for Robust Jacobian Lagging in Newton Methods. *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering* (2013).

Xiao-Chuan Cai and Yousef Saad. 1996. Overlapping Domain Decomposition Algorithms for General Sparse Matrices. *Numerical Linear Algebra with Applications* 3, 3 (1996), 221–237.

Xiao-Chuan Cai and Marcus Sarkis. 1999. A Restricted Additive Schwarz Preconditioner for General Sparse Linear Systems. *SIAM Journal on Scientific Computing* 21, 2 (1999), 792–797.

Ali Charara, David E. Keyes, and Hatem Ltaief. 2017. A Framework for Dense Triangular Matrix Kernels on Various Manycore Architectures. *Concurrency and Computation: Practice and Experience* 29, 15 (2017).

Jiong Chen, Florian Schäfer, Jin Huang, and Mathieu Desbrun. 2021. Multiscale Cholesky Preconditioning for Ill-Conditioned Problems. *ACM Trans. Graph. (SIGGRAPH)* 40, 4, Article 81 (July 2021), 13 pages.

Kwang-Jin Choi and Hyeong-Seok Ko. 2002. Stable but Responsive Cloth. *ACM Trans. Graph. (SIGGRAPH)* 21, 3 (July 2002), 604–611.

Victorita Dolean, Pierre Jolivet, and Frédéric Nataf. 2015. *An Introduction to Domain Decomposition Methods.* Society for Industrial and Applied Mathematics, Philadelphia, PA.

Maksymilian Dryja, Marcus V. Sarkis, and Olof B. Widlund. 1996. Multilevel Schwarz Methods for Elliptic Problems with Discontinuous Coefficients in Three Dimensions. *Numer. Math.* 72, 3 (01 Jan 1996), 313–348.

Maksymilian Dryja and Olof B Widlund. 1989. Some Domain Decomposition Algorithms for Elliptic Problems. In *Iterative methods for large linear systems.* Elsevier, San Diego, CA, 273–291.

Maksymilian Dryja and Olof B. Widlund. 1991. Multilevel Additive Methods for Elliptic Finite Element Problems. In *Parallel Algorithms for Partial Differential Equations, Proceedings of the Sixth GAMM-Seminar.* Friedr. Vieweg and Sohn, Braunschweig, Germany, 58–69.

Essex Edwards and Robert Bridson. 2015. The Discretely-Discontinuous Galerkin Coarse Grid for Domain Decomposition. *CoRR* abs/1504.00907 (2015). arXiv:1504.00907

Charbel Farhat, Michel Lesoinne, Patrick LeTallec, Kendall Pierson, and Daniel Rixen. 2001. FETI-DP: A Dual–Primal Unified FETI Method—Part I: A Faster Alternative to the Two-Level FETI Method. *Internat. J. Numer. Methods Engrg.* 50, 7 (2001), 1523–1544.

Charbel Farhat and Francois-Xavier Roux. 1991. A Method of Finite Element Tearing and Interconnecting and Its Parallel Solution Algorithm. *Internat. J. Numer. Methods Engrg.* 32, 6 (1991), 1205–1227.

Marco Fratarcangeli, Valentina Tibaldo, and Fabio Pellacini. 2016. Vivace: A Practical Gauss-Seidel Method for Stable Soft Body Dynamics. *ACM Trans. Graph. (SIGGRAPH Asia)* 35, 6, Article 214 (Nov. 2016), 9 pages.

Andreas Frommer and Daniel B. Szyld. 1999. Weighted Max Norms, Splittings, and Overlapping Additive Schwarz Iterations. *Numer. Math.* 83, 2 (01 Aug 1999), 259–278.

Martin J. Gander. 2008. Schwarz Methods over the Course of Time. *Electronic Transactions on Numerical Analysis* 31 (2008), 228–255.

Theodoros Gkountouvas, Vasileios Karakasis, Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. 2013. Improving the Performance of the Symmetric Sparse Matrix-Vector Multiplication in Multicore. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing.* 273–283.

Gundolf Haase, Ulrich Langer, and Arnd Meyer. 1991. The Approximate Dirichlet Domain Decomposition Method. Part I: An Algebraic Approach. *Computing* 47, 2 (01 Jun 1991), 137–151.

Liliya Kharevych, Weiwei Yang, Yiying Tong, Eva Kanso, Jerrold E. Marsden, Peter Schröder, and Mathieu Desbrun. 2006. Geometric, Variational Integrators for Computer Animation. In *Proceedings of SCA* (Vienna, Austria). 43–51.

Theodore Kim and Doug L. James. 2011. Physics-Based Character Skinning Using Multi-Domain Subspace Deformations. In *Proceedings of SCA* (Vancouver, British Columbia, Canada). 63–72.

Dilip Krishnan, Raanan Fattal, and Richard Szeliski. 2013. Efficient Preconditioning of Laplacian Matrices for Computer Graphics. *ACM Trans. Graph.* 32, 4, Article 142 (July 2013), 15 pages.

Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. 2009. Fast BVH Construction on GPUs. *Computer Graphics Forum* 28, 2 (2009), 375–384.

Yongjoon Lee, Sung-Eui Yoon, Seungwoo Oh, Duksu Kim, and Sunghee Choi. 2010. Multi-Resolution Cloth Simulation. *Comput. Graph. Forum (Pacific Graphics)* 29, 7 (2010), 2225–2232.

Minchen Li, Zachary Ferguson, Teseo Schneider, Timothy Langlois, Denis Zorin, Daniele Panozzo, Chenfanfu Jiang, and Danny M. Kaufman. 2020. Incremental Potential Contact: Intersection-and Inversion-Free, Large-Deformation Dynamics. *ACM Trans. Graph. (SIGGRAPH)* 39, 4, Article 49 (jul 2020), 20 pages.

Minchen Li, Ming Gao, Timothy Langlois, Chenfanfu Jiang, and Danny M. Kaufman. 2019. Decomposed Optimization Time Integrator for Large-Step Elastodynamics. *ACM Trans. Graph. (SIGGRAPH)* 38, 4, Article 70 (jul 2019), 10 pages.

Ruipeng Li and Yousef Saad. 2017. Low-Rank Correction Methods for Algebraic Domain Decomposition Preconditioners. *SIAM J. Matrix Anal. Appl.* 38, 3 (2017), 807–828.

Tiantian Liu, Adam W. Bargteil, James F. O'Brien, and Ladislav Kavan. 2013. Fast Simulation of Mass-Spring Systems. *ACM Trans. Graph. (SIGGRAPH Asia)* 32, 6, Article 214 (Nov. 2013), 7 pages.

Tiantian Liu, Sofien Bouaziz, and Ladislav Kavan. 2017. Quasi-Newton Methods for Real-Time Simulation of Hyperelastic Materials. *ACM Trans. Graph.* 36, 3, Article 23 (may 2017), 16 pages.

Jan Mandel. 1993. Balancing Domain Decomposition. *Communications in Numerical Methods in Engineering* 9, 3 (1993), 233–241.

Sebastian Martin, Bernhard Thomaszewski, Eitan Grinspun, and Markus Gross. 2011. Example-Based Elastic Materials. *ACM Trans. Graph. (SIGGRAPH)* 30, 4, Article 72 (July 2011), 8 pages.

Matthias Müller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. 2005. Meshless Deformations Based on Shape Matching. *ACM Trans. Graph. (SIGGRAPH)* 24, 3 (July 2005), 471–478.

Rajib Nath, Stanimire Tomov, Tingxing "Tim" Dong, and Jack Dongarra. 2011. Optimizing Symmetric Dense Matrix-Vector Multiplication on GPUs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (Seattle, Washington). Article 6, 10 pages.

Maxim Naumov, Marat Arsaev, Patrice Castonguay, Jonathan Cohen, Julien Demouth, Joe Eaton, Simon Layton, Nikolay Markovskiy, Istvan Reguly, Nikolai Sakharnykh, Vijay Sellappan, and Robert Strzodka. 2015. AmgX: A Library for GPU Accelerated Algebraic Multigrid and Preconditioned Iterative Methods. *SIAM Journal on Scientific Computing* 37, 5 (2015), 602–626.

Roy A. Nicolaides. 1987. Deflation of Conjugate Gradients with Applications to Boundary Value Problems. *SIAM J. Numer. Anal.* 24, 2 (1987), 355–365.

NVIDIA. 2021. AmgX. https://developer.nvidia.com/amgx.

Michael Ortiz and Laurent Stainier. 1999. The Variational Formulation of Viscoplastic Constitutive Updates. *Computer Methods in Applied Mechanics and Engineering* 171, 3 (1999), 419–444.

Garry Rodrigue, Kang LiShan, and Liu Yu-Hui. 1989. Convergence and Comparison Analysis of Some Numerical Schwarz Methods. *Numer. Math.* 56, 2 (01 Feb 1989), 123–138.

Yousef Saad. 1994. ILUT: A Dual Threshold Incomplete LU Factorization. *Numerical Linear Algebra with Applications* 1, 4 (1994), 387–402.

Hermann A. Schwarz. 1870. Ueber Einen Grenz Übergang Durch Alternirendes Verfahren. Wolf J. XV. 272-286.

N. Spillane and D.J. Rixen. 2013. Automatic Spectral Coarse Spaces for Robust Finite Element Tearing and Interconnecting and Balanced Domain Decomposition Algorithms. *Internat. J. Numer. Methods Engrg.* 95, 11 (2013), 953–990.

Rasmus Tamstorf, Toby Jones, and Stephen F. McCormick. 2015. Smoothed Aggregation Multigrid for Cloth Simulation. *ACM Trans. Graph. (SIGGRAPH Asia)* 34, 6, Article 245 (Oct. 2015), 13 pages.

Min Tang, Zhongyuan Liu, Ruofeng Tong, and Dinesh Manocha. 2018. PSCC: Parallel Self-Collision Culling with Spatial Hashing on GPUs. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 1, Article 18 (July 2018), 18 pages.

Joseph Teran, Silvia Blemker, Victor Ng-Thow-Hing, and Ronald Fedkiw. 2003. Finite Volume Methods for the Simulation of Skeletal Muscle. In *Proceedings of SCA 2003* (San Diego, California). 68–74.

Joseph Teran, Eftychios Sifakis, Geoffrey Irving, and Ronald Fedkiw. 2005. Robust Quasistatic Finite Elements and Flesh Simulation. In *Proceedings of SCA* (Los Angeles, California). 181–190.

Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. 1987. Elastically Deformable Models. *Computer Graphics (SIGGRAPH 87)* 21, 4 (Aug. 1987), 205–214.

Pascal Volino, Nadia Magnenat-Thalmann, and Francois Faure. 2009. A Simple Approach to Nonlinear Tensile Stiffness for Accurate Cloth Simulation. *ACM Trans. Graph.* 28, 4, Article 105 (Sept. 2009), 16 pages.

Huamin Wang. 2015. A Chebyshev Semi-Iterative Approach for Accelerating Projective and Position-Based Dynamics. *ACM Trans. Graph. (SIGGRAPH Asia)* 34, 6, Article 246 (Oct. 2015), 9 pages.

Huamin Wang. 2021. GPU-Based Simulation of Cloth Wrinkles at Submillimeter Levels. *ACM Trans. Graph.* 40, 4, Article 169 (July 2021), 14 pages.

Huamin Wang and Yin Yang. 2016. Descent Methods for Elastic Body Simulation on the GPU. *ACM Trans. Graph. (SIGGRAPH Asia)* 35, 6, Article 212 (Nov. 2016), 10 pages.

Zhendong Wang, Longhua Wu, Marco Fratarcangeli, Min Tang, and Huamin Wang. 2018. Parallel Multigrid for Nonlinear Cloth Simulation. *Comput. Graph. Forum (Pacific Graphics)* 37, 7 (2018), 131–141.

Joerg Willems. 2013. Spectral Coarse Spaces in Robust Two-Level Schwarz Methods. In *Numerical Solution of Partial Differential Equations: Theory, Algorithms, and Their Applications. Springer Proceedings in Mathematics & Statistics*, Vol. 45. Springer, 303—-326.

Longhua Wu, Botao Wu, Yin Yang, and Huamin Wang. 2020. A Safe and Fast Repulsion Method for GPU-Based Cloth Self Collisions. *ACM Trans. Graph.* 40, 1, Article 5 (Dec. 2020), 18 pages.

Xiaofeng Wu, Rajaditya Mukherjee, and Huamin Wang. 2015. A Unified Approach for Subspace Simulation of Deformable Bodies in Multiple Domains. *ACM Trans. Graph. (SIGGRAPH Asia)* 34, 6, Article 241 (Oct. 2015), 9 pages.

Zangyueyang Xian, Xin Tong, and Tiantian Liu. 2019. A Scalable Galerkin Multigrid Method for Real-Time Simulation of Deformable Objects. *ACM Trans. Graph.* 38, 6, Article 162 (Nov. 2019), 13 pages.

Yin Yang, Weiwei Xu, Xiaohu Guo, Kun Zhou, and Baining Guo. 2013. Boundary-Aware Multi-Domain Subspace Deformation. *IEEE Transactions on Visualization and Computer Graphics* 19, 10 (2013), 1633–1645.

Xuejun Zhang. 1992. Multilevel Schwarz Methods. *Numer. Math.* 63, 1 (01 Dec 1992), 521–539.

Yongning Zhu, Eftychios Sifakis, Joseph Teran, and Achi Brandt. 2010. An Efficient Multigrid Method for the Simulation of High-resolution Elastic Solids. *ACM Trans. Graph.* 29, 2, Article 16 (April 2010), 18 pages.