Computers & Graphics (2022)

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Parallel Block Neo-Hookean XPBD using Graph Clustering

Quoc-Minh Ton-That[a,*], Paul G. Kry[b], Sheldon Andrews[c]

[a] *École de Technologie Supérieure, Montreal, Canada*
[b] *McGill University, Montreal, Canada*
[c] *École de Technologie Supérieure, Montreal, Canada*

ARTICLE INFO

Article history:
Received 23 September 2022

finite element method, physics-based animation, soft body simulation, elasticity, real-time physics

ABSTRACT

The eXtended Position Based Dynamics algorithm (XPBD) enables unified simulation of various materials from fluids to both elastic solids and stiff solids. In particular, finite element based neo-Hookean models can simulate near incompressible materials by means of a decoupled compliant constraint formulation. Due to XPBD's reliance on local constraint projections in the solver loop, its computational nature lends itself to parallelization by means of graph coloring algorithms used to determine partitions of independent constraints which can be solved simultaneously. However, minimal graph coloring is bounded from below by the maximum valence of the finite element mesh, thus hindering parallelization opportunities. In this paper, we propose a novel graph clustering approach on the constraint graph which groups highly dependent constraints into supernodes. By applying graph coloring on the supernodal constraint graph, we are able to significantly reduce the number of partitions, thus enhancing parallelization of the solver. Furthermore, we accelerate convergence of the neo-Hookean XPBD solver by a coupled constraint formulation, resulting in enhanced stability and efficiency compared to previous approaches.

1. Introduction

Physics-based animation is increasingly used by computer graphics, robotics, and VR training applications as a way to generate realistic and complex animations of soft or deformable objects. In the case of interactive applications, where real-time frame rates are often required, extended position based dynamics (XPBD) [16] has proven to be a useful framework for efficiently simulating many different physical phenomena. However, even though parallel implementations of XPBD are straightforward to realize, simulations involving large and complex models may still struggle to achieve performance requirements or remain stable. Improving simulation efficiency is therefore a continuing research goal.

State-of-the-art methods for simulating elastic objects using XPBD employ continuum-based constraints [3] applied to a finite element discretization. Recently, a decoupled constraint-pair formulation for stable neo-Hookean materials [17] was shown to be particularly well-suited for simulating nearly incompressible materials. Real-time performance is obtained by a massively parallel implementation of the XPBD algorithm, which is made possible by a local Gauss-Seidel type solver.

*Corresponding author:
e-mail: tonthat.quocminh@gmail.com (Quoc-Minh Ton-That)

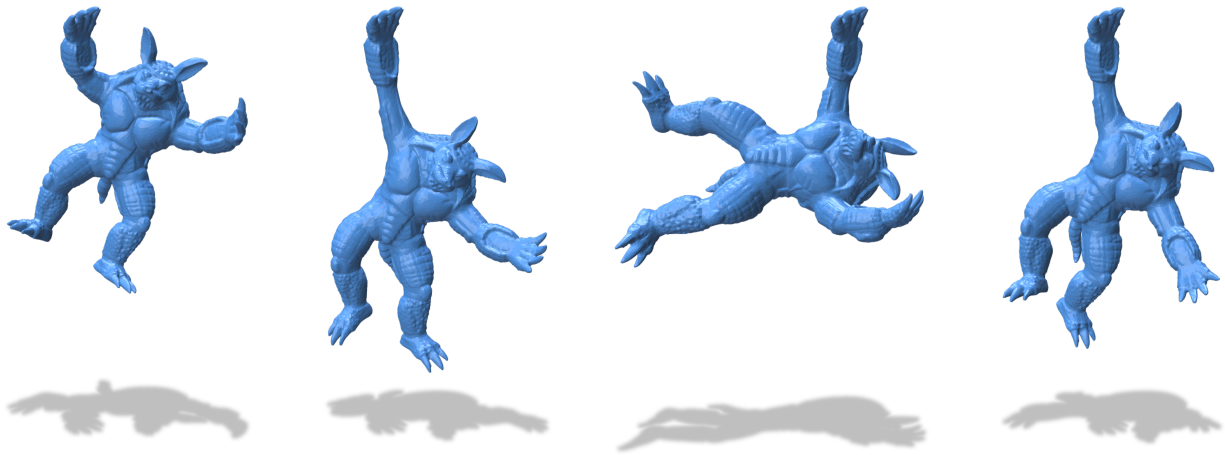

Fig. 1: We simulate a deformable tetrahedral mesh composed of 165k elements at 15 frames per second. Our coupled constraint formulation significantly improves convergence of state of the art XPBD, while our greedy graph clustering algorithm further reduces computational cost compared to traditional graph coloring approaches, reducing solve times by an order of magnitude.

However, performance may still be impacted if the solver does not converge to a sufficiently accurate solution within an allotted time budget.

Gauss-Seidel type solver algorithms are known to demonstrate faster convergence, yet are difficult to parallelize since sub-sets of independent constraints, or partitions, must first be identified. Graph coloring approaches [8, 9, 10] may be used to obtain independent constraint sets. However, the continuum-based elastic constraints used for simulating tetrahedral models often yield large numbers of partitions, thus limiting acceleration potential that comes with a parallel solver since partitions must be processed sequentially.

In this paper, we accelerate the convergence of elastic body simulation using stable neo-Hookean constraints by solving the hydrostatic and deviatoric constraints in a coupled fashion, which exhibits faster convergence rates surpassing current methods. Figure 1 shows a preview of our results. Our proposed algorithm requires only trivial modifications to existing XPBD implementations by using 2×2 block solves to compute the Lagrange multiplier corrections rather than computing them individually. Solver efficiency is further improved by means of a graph clustering method that significantly reduces the number of constraint partitions and that facilitates a massively parallel solver implementation. The clustering method has the added benefit that it generalizes to arbitrary constraint models and geometrical discretizations.

2. Related Work

Our work builds on previous acceleration methods for soft-body simulation. Acceleration methods in this context either improve computational efficiency, improve convergence of the underlying numerical methods, or reduce the complexity of the original problem. In this section, we briefly review work that focuses on employing these strategies, or combinations of them, to improve the performance of soft-body simulations.

Model Reduction. Model reduction approaches compute soft-body deformations using a mapping from a low-dimensional deformation space to high-resolution deformations in the full space of the model. By reformulating the equations of motion in the reduced space, the size of the system of non-linear equations to be solved for time integration is significantly reduced, leading to major speed ups. Barbič and James [2] apply modal analysis to the stiffness matrix, keeping only the most relevant deformation modes, and derive a second-order Taylor expansion around a given configuration to account for non-linear elastic models. An et al. [1] accelerate the computation of internal forces in a reduced space by choosing only a coarse set of representative integration points in the finite element mesh and compute energy gradients at those points. Kim and James [14] enable updating and downdating

the mapping dynamically. Fulton et al. [11] introduce neural networks for non-linear model reduction by expressing the mapping as a variational auto-encoder. Shen et al. [30] build on the work of Fulton et al. [11] by introducing a complex-step finite difference approach with reverse mode automatic differentiation to compute higher-order derivatives of the mapping function, thus enhancing the expressivity of the deformations. While model reduction techniques are particularly efficient, handling large mesh deformations in real-time, they require non-trivial precomputation, and recently proposed model reduction methods based on machine learning often require manual intervention during the training phase. Additionally, model reduction techniques do not lend themselves well to varying boundary conditions and topological changes due to the precomputation phase assuming their invariance.

Constraint-based Methods. Projective dynamics [5] reformulates the elastic potential as a sum of quadratic constraint energies. In doing so, the optimization problem associated with the backward Euler time stepping scheme becomes a convex quadratic optimization problem with a constant Hessian matrix that can be prefactored (e.g., by a sparse Cholesky decomposition). A local-global iterative approach is employed at each time step, where the local step consists of constraint projections that may be computed in a parallel fashion, and the global step solves the prefactored linear system. Brandt et al. [6] apply model reduction to projective dynamics and achieve simulation rates an order of magnitude faster than the standard algorithm. Fratarcangeli et al. [9] and Fratarcangeli et al. [10] accelerate the global solve using graph coloring techniques with iterative linear solvers. Wang [33] accelerate convergence using the Chebyshev semi-iterative method by estimating a spectral radius associated to projective dynamics solves. Peng et al. [26] express the local-global steps as a fixed point iteration, enabling the use of a stable Anderson acceleration scheme to improve convergence. Although projective dynamics is highly stable and efficient, constraints are restricted to a quadratic form, and the global solve is not easily parallelized. Additionally, dynamic constraints must be incorporated by refactorizing the lead matrix, or by incremental updates and downdates. These properties pose performance challenges for multi-body simulations compared to other position based methods [4].

Extended position based dynamics [16, 23] (XPBD) is another constraint-based framework that provides a unified approach for multi-phase multi-body simulation. However, it requires only that the constraint functions are rigid motion invariant and first order differentiable. XPBD time integration consists of a symplectic Euler step, followed by an iterative solver loop that projects positions onto individual constraint manifolds one at a time along constraint gradients. Exploiting the sparsity of these gradients yields a massively parallel solver loop, which is the key to XPBD's performance. Although PBD [21] methods were first used with geometric constraints, recent methods favor continuum-based elasticity constraints [4, 17, 22].

It is also possible to employ global linear solves of the compliant constraint formulation [28], which yields better convergence. However, such approaches forfeit parallelizability of the solver loop. Inclusion of a geometric stiffness term may drastically improve stability in this case [32].

Graph Partitioning. Fratarcangeli and Pellacini [7] use graph coloring [20] to determine independent sets of XPBD constraints that can be solved in parallel on the GPU. The number of colors determines the number of independent sets, and consequently determines the number of GPU kernel calls per solver iteration. Fratarcangeli and Pellacini [8] observed that a large number of GPU kernel calls is the main culprit for parallel execution time overhead, and succeed in reducing the number of colors on the constraint graph by introducing ghost constraints and ghost particles. Advantageous speedups are thus obtained, but the dynamics of the physics change. Matula and Beck [19] show that vertex reorderings can bring the color count closer to the lower bound. They propose a smallest degree last vertex reordering which may yield lower numbers of colors in practice. Peiret et al. [25] and Liu and

Andrews [15] use graph partitioning algorithms to parallelize the simulation of highly connected systems of rigid bodies. Our proposed method builds on the work of Macklin and Müller [17]. They mention that it is possible to reduce the number of constraint partitions by considering hexahedral meshes in which each hexahedron contains 5 or 6 constrained tetrahedra. A lower bound of 8 sets of hexahedra is attainable, thus significantly reducing GPU kernel call overhead, making simulation maximize parallelizability.

3. Background

For deformable tetrahedral meshes with vertices V and tetrahedra T, XPBD simulations integrate in time Newton's equations of motion

$$\mathbf{M}\ddot{\mathbf{x}} = \mathbf{f}_{\text{int}}(\mathbf{x}) + \mathbf{f}_{\text{ext}}(\mathbf{x}) \,, \tag{1}$$

where $\mathbf{M} \in \mathbb{R}^{n\times n}$ is the diagonal nodal mass matrix, $\mathbf{f}_{\text{int}}(\mathbf{x}) \in \mathbb{R}^n$ is the internal elastic force vector, $\mathbf{f}_{\text{ext}}(\mathbf{x}) \in \mathbb{R}^n$ is the external force vector, and $\mathbf{x} \in \mathbb{R}^n$ are the degrees of freedom corresponding to the vector of stacked tetrahedral mesh vertex coordinates, where $n = 3|V|$.

The internal forces $\mathbf{f}_{\text{int}}(\mathbf{x})$ are expressed as the negative gradient of the potential

$$U(\mathbf{x}) = \frac{1}{2} C(\mathbf{x})^T \boldsymbol{\alpha}^{-1} C(\mathbf{x}) \,, \tag{2}$$

such that

$$\begin{aligned} \mathbf{f}_{\text{int}}(\mathbf{x}) &= -\nabla U(\mathbf{x})^T \\ &= -\nabla C(\mathbf{x})^T \boldsymbol{\alpha}^{-1} C(\mathbf{x}) \,, \end{aligned} \tag{3}$$

where $C(\mathbf{x}) = \left[C_1(\mathbf{x}), C_2(\mathbf{x}), \ldots, C_m(\mathbf{x})\right]^T \in \mathbb{R}^m$ is a vector of m constraint functions $C_j(\mathbf{x}) \in \mathbb{R}$, and $\boldsymbol{\alpha} \in \mathbb{R}^{m\times m}$ is a block diagonal compliance matrix.

Following a finite difference time discretization, the equations of motion in Equation 1 are reformulated by introducing Lagrange multipliers $\boldsymbol{\lambda} \in \mathbb{R}^m$ defined as

$$\boldsymbol{\lambda} = -\tilde{\boldsymbol{\alpha}}^{-1} C(x) \,, \tag{4}$$

where $\tilde{\boldsymbol{\alpha}} = \frac{\boldsymbol{\alpha}}{\Delta t^2}$ and Δt is the time step.

Algorithm 1 XPBD time integration for a single time step.

```
h ← Δt/numSubSteps                          ▷ Δt is time step size
for s = 1 ... numSubSteps do
    λ ← 0
    x ← x^t + h v^t + h^2 M^{-1} f_ext(x)
    α̃ ← (1/h^2) α
    for j = 1 ... m do                      ▷ Solver iteration
        A ← [∇C_j(x) M^{-1} ∇C_j(x)^T + α̃_jj]
        Δλ_j ← −A^{-1} (C_j(x) + α̃_jj λ_j)
        Δx ← M^{-1} ∇C_j^T(x) Δλ_j
        λ_j ← λ_j + Δλ_j
        x ← x + Δx                          ▷ constraint projection
    end for
    v^{t+1} ← (x − x^t)/h
    x^{t+1} ← x
end for
```

Integrating Equation 1 by one step in time then requires a non-linear solve with the fixed point iteration

$$\begin{aligned} \boldsymbol{\lambda}_{k+1} &= \boldsymbol{\lambda}_k + \Delta\boldsymbol{\lambda} \\ \mathbf{x}_{k+1} &= \mathbf{x}_k + \Delta\mathbf{x} \end{aligned} \tag{5}$$

starting from the initial iterates $\mathbf{x}_0 = \mathbf{x}^t + \Delta t \mathbf{v}^t + \Delta t^2 \mathbf{M}^{-1} \mathbf{f}_{\text{ext}}(\mathbf{x})$, where $\mathbf{x}^t$ are the degrees of freedom at time step t and $\mathbf{v}^t$ are the nodal velocities at time step t, and $\boldsymbol{\lambda}_0 = 0$.

Instead of using a global linear solve to obtain $\Delta\boldsymbol{\lambda}$ and $\Delta\mathbf{x}$ at every iteration, XPBD opts for a Gauss-Seidel type local solver which projects every constraint $C_j(\mathbf{x})$, updating their associated Lagrange multiplier and degrees of freedom sequentially. The XPBD algorithm is presented in Algorithm 1, where substepping is used [18].

Recent methods simulate hyper-elastic continuum-based materials in a position based framework [3] by using tetrahedral meshes and a piece-wise linear finite element discretization of the deformation function $\phi(X)$. They constrain each tetrahedral element $e \in T$ by some strain energy density function $\Psi(\mathbf{F})$ integrated over the domain Ω^e of e, where $\mathbf{F} = \nabla_X \phi(X) \in \mathbb{R}^{3\times 3}$. Due to the piece-wise linear basis, $\mathbf{F}$ is constant over Ω^e, such that element e's constraint function trivially becomes

$$\begin{aligned} C_e(\mathbf{x}) &= \int_{\Omega^e} \Psi(\mathbf{F}) \partial\Omega^e \\ &= \omega_e \Psi(\mathbf{F}), \end{aligned} \tag{6}$$

where ω_e is the volume of the element.

Macklin and Müller [17] use the stable neo-Hookean model of Smith et al. [31], where the strain energy density is defined

as

$$\Psi_{\text{neo}}(\mathbf{F}) = \underbrace{\frac{\lambda}{2}(\det(\mathbf{F}) - \gamma)^2}_{\Psi_H(\mathbf{F})} + \underbrace{\frac{\mu}{2}(\text{tr}(\mathbf{F}^T\mathbf{F}) - 3)}_{\Psi_D(\mathbf{F})}, \tag{7}$$

with $\gamma = 1 + \frac{\mu}{\lambda}$, where μ, λ are the Lamé coefficients.

The strain energy density may be decomposed into a hydrostatic term, $\Psi_H(\mathbf{F})$, and deviatoric term $\Psi_D(\mathbf{F})$. Instead of defining a single constraint function per element, Macklin and Müller [17] exploit Equation 7 to determine a pair of hydrostatic and deviatoric constraint function and compliance for every element:

$$C^H(\mathbf{x}) = \det(\mathbf{F}) - \gamma \tag{8a}$$

$$\alpha^H = \frac{1}{\lambda\omega_e} \tag{8b}$$

$$C^D(\mathbf{x}) = \sqrt{\text{tr}(\mathbf{F}^T\mathbf{F})} \tag{9a}$$

$$\alpha^D = \frac{1}{\mu\omega_e} \tag{9b}$$

We thus have that $m = 2|T|$, $\boldsymbol{\alpha}$ is a diagonal matrix with coefficient pairs α^H and α^D, and $C(\mathbf{x})$ is a vector of constraint pairs $C^H(\mathbf{x})$ and $C^D(\mathbf{x})$, for every element e.

3.1. Parallel Solver

Because constraint projection in solver loop iterations directly updates $\mathbf{x}$ based on its current value, we cannot trivially parallelize the algorithm. However, each constraint gradient $\nabla C_j(\mathbf{x})$ is sparse. Two constraints that have non-overlapping sparsity patterns can update $\mathbf{x}$ in any order because the constraint projections modify independent degrees of freedom.

To better understand how the sparsity of the constraint gradients can be leveraged for parallel updates, we define

$$\mathscr{C}_j = \left\{ i \,\middle|\, \frac{\partial C_j(\mathbf{x})}{\partial \mathbf{x}_i} \neq 0, \forall\, i \in [1, n] \right\} \tag{10}$$

as the sparsity pattern of $\nabla C_j(\mathbf{x})$ giving its non-zero structure. In other words, $\mathscr{C}_j$ gives the degrees of freedom that influence constraint C_j. Therefore, to parallelize the solver loop, we seek to partition the constraints into a set of partitions $\mathscr{P}$, where each partition $\mathscr{P}_c \subset [1, m]$, for $c \in [1, |\mathscr{P}|]$, contains independent constraints, i.e., $\mathscr{C}_j \cap \mathscr{C}_k = \emptyset$ for any two constraint indices j and k in $\mathscr{P}_c$.

Algorithm 2 Parallel XPBD solver iteration.

```
for c ∈ 𝒫 do
    for j ∈ 𝒫_c do                                   ▷ Parallelize loop
        A ← [∇C_j(x) M^{-1} ∇C_j(x)^T + α̃_jj]
        Δλ_j ← −A^{-1}(C_j(x) + α̃_jj λ_j)
        Δx ← M^{-1} ∇C_j^T(x) Δλ_j
        λ_j ← λ_j + Δλ_j
        x ← x + Δx
    end for
end for
```

Algorithm 1 can then be modified to loop over each partition $\mathscr{P}_c$ and perform a parallel projection of all constraints in $\mathscr{P}_c$. The parallel variant of the XPBD solver is provided in Algorithm 2.

3.2. Graph Coloring

Let us next consider how to construct the constraint partitions $\mathscr{P}_c$ from a constraint graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where the graph nodes $\mathcal{V}$ are constraints, and $\mathcal{E}$ is the set of edges. An edge exists between two graph nodes (constraints) iff they share at least one degree of freedom, that is,

$$(C_j, C_k) \in \mathcal{E} \iff \mathscr{C}_j \cap \mathscr{C}_k \neq \emptyset. \tag{11}$$

It is possible to find one of many feasible constraint partitions by a graph coloring algorithm acting on the constraint graph $\mathcal{G}$. A graph coloring algorithm will assign a color to each constraint such that it is different from the color of any neighboring constraint. That is, if (C_j, C_k) is an edge in $\mathcal{E}$ then the constraint gradient patterns overlap and the two constraints must have different colors. This property permits the construction of partitions by grouping constraints of the same color, i.e., C_j for $j \in \mathscr{P}_c$ will all have the same color, and that color will be different from the color of constraints in other partitions. Thus, all constraints of the same color may be projected in parallel in the solver loop.

We refer the reader to the work by Fratarcangeli and Pellacini [8] for further details on graph coloring approaches suitable to position based dynamics simulations. The notation we use in this section and the rest of the paper is summarized in Table 1.

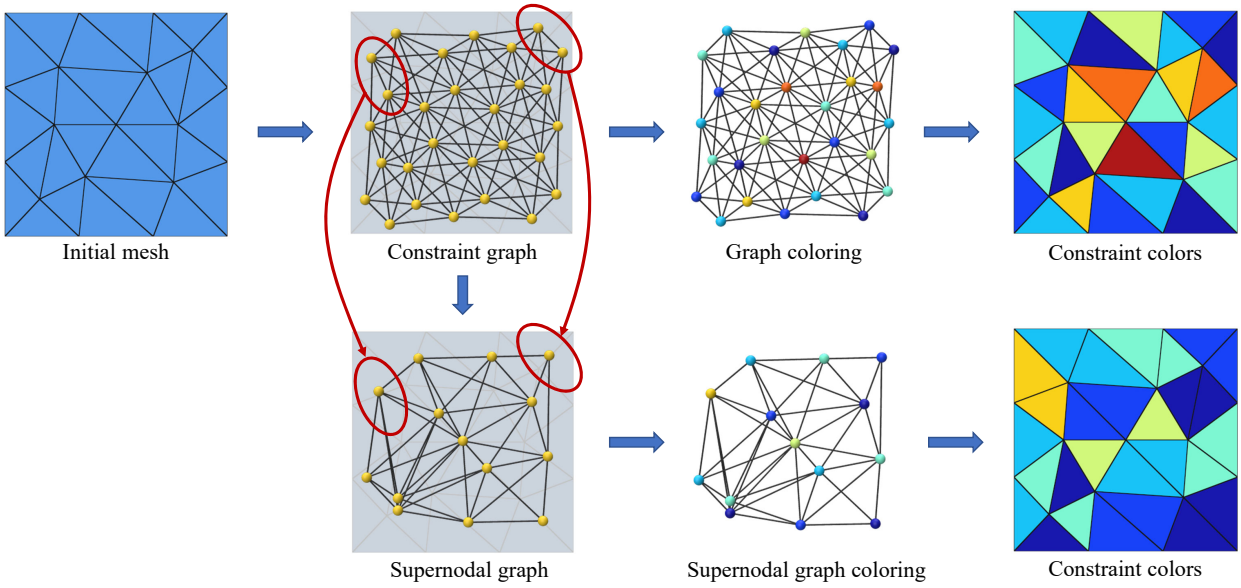

Fig. 2: Schematic overview of our graph clustering based constraint partitioning method. The top row shows the traditional partitioning approach based on directly coloring the constraint graph $\mathcal{G}$. In contrast, our method builds a supernodal graph $\bar{\mathcal{G}}$ from $\mathcal{G}$ by means of a clustering algorithm. The coloring of $\bar{\mathcal{G}}$ (sparse graph bottom row), in comparison to the coloring of $\mathcal{G}$ (dense graph top row), generates significantly fewer constraint partitions and leads to superior solver performance.

Table 1: Notation and symbols used throughout this paper.

Symbol	Definition
C_j	j^{th} constraint in vector C, where $j \in \mathcal{V}$
$\mathscr{C}_j$	Non-zero pattern of $\nabla C_j(x)$
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	Constraint graph
$\mathscr{P}_c$	Constraint partition c
$\mathscr{P}$	Set of constraint partitions
r	r^{th} constraint cluster
$\chi(r)$	Constraint set of cluster r
$\pi(j)$	Cluster of constraint j
$\bar{\mathcal{G}} = (\bar{\mathcal{V}}, \bar{\mathcal{E}})$	Constraint cluster graph
$\bar{\mathscr{P}}_c$	Cluster partition c
$\bar{\mathscr{P}}$	Set of cluster partitions

4. Graph Clustering

We observe from Section 3.2 and Algorithm 2 that parallelization is directly related to the number of colors obtained from the graph coloring process. In the limit, if we have as many colors as constraints, our parallel solver reverts to its sequential counterpart. In contrast, for a single color, all constraints can be projected simultaneously. Hence, as the number of colors decreases, parallelism is enhanced.

4.1. Bounds on Parallelizability

For a tetrahedral discretization, each continuum constraint is parameterized by exactly 12 degrees of freedom comprised of the x, y, z components of the four tetrahedral nodes. Because a node is shared by all of its incident tetrahedra, its corresponding degrees of freedom are also shared by all constraints associated with these incident tetrahedra. The number of colors is therefore bounded below by the largest vertex one-ring neighbourhood of tetrahedra. For the stable neo-Hookean constraints, there are exactly two constraints per tetrahedra. The following lower bound on the number of colors of a tetrahedral mesh model using this neo-Hookean material thus applies

$$|\mathscr{P}| \geq \max_{i \in [1,|V|]} 2|\mathcal{N}_i|, \tag{12}$$

where $\mathcal{N}_i$ is the one-ring neighbourhood of tetrahedra around vertex i. In graph terminology, in the general case, the minimum number of colors in a graph $\mathcal{G}$ is the size of the largest clique in $\mathcal{G}$.

In practice, this lower bound can be prohibitively high. For instance, a tetrahedral mesh built from a regular grid of voxels, with each grid cell containing 5 tetrahedra, has a lower bound of around 40 colors. For general meshes, the number of necessary colors can be much higher. Thus, GPU implementations of XPBD are hindered by simulating tetrahedral finite element meshes, which require a significant number of GPU kernel calls per solver iteration.

We know that hexahedral meshes have much better lower bounds on coloring, such that it is possible to obtain only 8 colors, and each hexahedron can be chosen to encapsulate 5 or 6 tetrahedral elements. This is due to the fact that for a regular hexahedral mesh, interior vertices have exactly 8 incident hexahedra, while boundary vertices have even less. Thus, for stable neo-Hookean constraints derived from a hexahedral mesh, it is possible for every hexahedron to encapsulate either 10 or 12 constraints derived from their associated 5 or 6 tetrahedra. Rather than coloring the constraint graph, we may then color the graph of hexahedra instead and thus obtain much better parallelism due to the low number of hexahedral partitions created. Each partition can then project groups of constraints encapsulated by all hexahedra of the same color (albeit in a sequential manner for those constraints within the same hexahedron).

This analysis allows us to conclude that for the same constraint set, it is possible to significantly enhance parallelism by forming partitions of groups of constraints, rather than partitions of constraints. However, we will show that these constraints need not be derived from a finite element hexahedral mesh. We thus propose to generalize this constraint grouping approach as a graph clustering approach.

4.2. Grouping Constraints

Regardless of the underlying spatial discretization and constraint set, we aim to identify non-overlapping clusters of constraints in $\mathcal{G}$. We thus define $\bar{\mathcal{G}} = (\bar{\mathcal{V}}, \bar{\mathcal{E}})$ as the *supernodal graph* derived from the original constraint graph $\mathcal{G}$, where $\bar{\mathcal{V}}$ is the set of constraint clusters, or *supernodes*, such that coloring $\bar{\mathcal{G}}$ will result in fewer partitions. The supernodal graph $\bar{\mathcal{G}}$ is related to the constraint graph $\mathcal{G}$ by a parent map $\pi(\cdot)$ and a children map $\chi(\cdot)$. Given a constraint index $j \in \mathcal{V}$, $\pi(j) = r$ tells us that constraint C_j belongs to cluster $r \in \bar{\mathcal{V}}$, while $\chi(r) = \{j \mid j \in \mathcal{V}\ , \pi(j) = r\}$ lists all constraint indices belonging to cluster $r \in \bar{\mathcal{V}}$.

Hence, our supernodal graph forms a topology of clusters $r \in \bar{\mathcal{V}}$ whose connectivity is defined by the property

$$(r, s) \in \bar{\mathcal{E}} \iff \exists\, (j, k) \in \mathcal{E} \mid j \in \chi(r) \wedge k \in \chi(s). \tag{13}$$

Algorithm 3 Clustered parallel XPBD solver iteration.

```
for 𝒫̄_c ∈ 𝒫̄ do
    for r ∈ 𝒫̄_c do                                  ▷ Parallelize loop
        for C_j ∈ χ(r) do
            A ← [∇C_j(x) M^{-1} ∇C_j(x)^T + α̃_jj]
            Δλ_j ← −A^{-1}(C_j(x) + α̃_jj λ_j)
            Δx ← M^{-1} ∇C_j^T(x) Δλ_j
            λ_j ← λ_j + Δλ_j
            x ← x + Δx
        end for
    end for
end for
```

In other words, two clusters are neighbours in the supernodal graph $\bar{\mathcal{G}}$ if and only if they contain constraints (one from each cluster) with overlapping sparsity pattern (i.e., overlapping influence). Equivalently, $\bar{\mathcal{E}}$ can be defined using the parent map π as

$$\bar{\mathcal{E}} = \{(\pi(j), \pi(k)) \mid (j, k) \in \mathcal{E}, \pi(j) \neq \pi(k)\}. \tag{14}$$

A graph coloring algorithm applied to $\bar{\mathcal{G}}$ thus outputs a set $\bar{\mathscr{P}}$ of partitions $\bar{\mathscr{P}}_c$ of clusters r. The clustered parallel XPBD solver variant is presented in Algorithm 3.

Once again, we have assumed the existence of such a set of clusters $\bar{\mathcal{V}}$. How should one find such a convenient clustering of $\mathcal{G}$? Ideally, we would like $|\bar{\mathscr{P}}| \ll |\mathscr{P}|$. Certain classical clustering techniques [27] fail on constraint graphs derived from meshes. This is due to the fact that classical clustering techniques look for natural clusters in non-uniform graphs, such as those formed in social networks, while meshes are especially uniform. Computing exact solutions for such clustering problems remains NP-hard [27]. Consequently, we choose to develop a simple greedy clustering algorithm derived from an intuitive heuristic.

4.3. Greedy Algorithm

By noticing that the number of colors resulting from a graph coloring process increases as the amount of dependencies between constraints increases, we define the distance metric

$$d(C_j, C_k) = 1 - \frac{|\mathscr{C}_j \cap \mathscr{C}_k|}{|\mathscr{C}_j \cup \mathscr{C}_k|}. \tag{15}$$

Intuitively, this metric tells us that constraints sharing many degrees of freedom in their parameterization should be

considered *close*, while constraints sharing few degrees of freedom or none at all should be considered *far*. Our clustering approach then seeks to find clusters of constraints near one another. In doing so, individual clusters contain highly dependent constraints, while constraints from separate clusters are less likely to depend on each other, thus reducing the lower bound on optimal coloring for $\bar{\mathcal{G}}$.

However, as a cluster grows, so does its computational workload in the parallel solver. Specifically, there is more work to be done sequentially. Furthermore, uneven cluster sizes can also negatively impact parallel implementations, since smaller clusters scheduled in the same parallel batch must wait for larger clusters in the same batch to finish executing, thus occupying computational resources without actually executing meaningful computations. Hence, we make the maximum cluster size K_s a user-defined parameter to help keep cluster sizes balanced.

We start our greedy clustering approach by initializing the parents of constraints to be unassigned (i.e., $\pi(C_i) = \emptyset$), and then choose a seed vertex u_0 in $\mathcal{G}$ as the first constraint satisfying

$$u_0 = \underset{C_j \in \mathcal{V},\, \pi(C_j)=\emptyset}{\arg\min} \sum_{(C_j, C_k) \in \mathcal{E},\, \pi(C_k)=\emptyset} d(C_j, C_k). \tag{16}$$

Then, starting from u_0, we traverse the graph $\mathcal{G}$ in breadth first order while greedily adding unclaimed neighbouring constraints in increasing order with respect to the distance metric, and starting a new cluster each time the maximum cluster size K_s is reached.

If not all nodes are assigned to clusters, but no clusters of size K_s can be created, the process repeats with a seed constraint identified by Equation 16, and with the the maximum cluster size K_s decreased by 1.

This algorithm thus produces as output a set $\bar{\mathcal{V}}$ of constraint clusters r, with $1 \leq |\chi(r)| \leq K_s$, where K_s denotes the initial user-defined maximum cluster size. Traversing the constraint graph in breadth-first order enables the clustering process to exploit spatial locality in the underlying geometry from which the topology of $\mathcal{G}$ is inherently derived. See Algorithm 4 for the pseudo-code and Figure 3 for a visual example of the clustering procedure.

Algorithm 4 Greedy K-Neighbour Clustering.

```
V̄ ← ∅
while K_s ≥ 1 do
    u_0 ← Equation 16
    for C_j ∈ BFS_G(u_0) do          ▷ Breadth first order from u_0 in G
        if π(j) = ∅ then
            r ← |V̄|
            χ(r) ← χ(r) ∪ {j}
            π(j) ← r
            for (C_j, C_k) ∈ E do    ▷ Sorted by increasing d(C_j, C_k)
                if π(k) = ∅ ∧ |χ(r)| < K_s then
                    χ(r) ← χ(r) ∪ {k}
                    π(k) ← r
                end if
            end for
            if |χ(r)| = K_s then
                V̄ ← V̄ ∪ {r}
            else                     ▷ Cluster too small, undo
                for k ∈ χ(r) do
                    π(k) = ∅
                end for
                χ(r) = ∅
            end if
        end if
    end for
    K_s ← K_s − 1                    ▷ Make smaller clusters with leftovers
end while
```

Using our graph clustering algorithm in a precomputation phase, the parallel solver variant in Algorithm 3 is observed to be at least as efficient as the classical parallel variant in Algorithm 2, depending on the underlying hardware. In theory, our variant becomes faster compared to the classical graph coloring based parallel XPBD as the hardware parallelism capacity increases. We further note that our clustering can be used with any constraint type.

5. Neo-Hookean Constraint Coupling

While clustering improves computational efficiency, the solver convergence is largely unaffected. In this section, we propose a modification to the constraint formulation of the stable neo-Hookean model that significantly improves the convergence of the solver.

Although the constraint formulation for the stable neo-Hookean material model of Macklin and Müller [17] exhibits

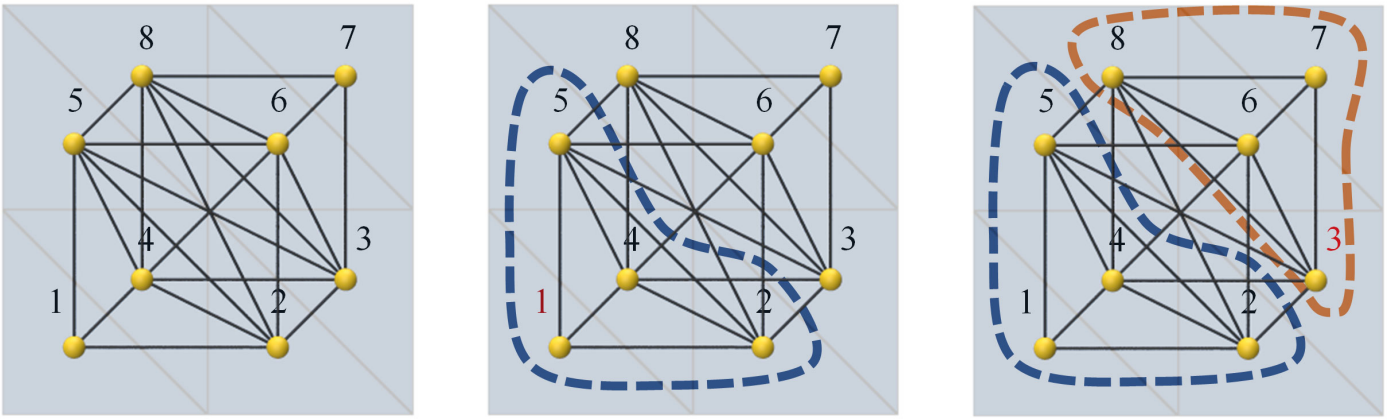

Fig. 3: An example of how greedy clustering with Algorithm 4 computes supernodes with maximum cluster size $K_s = 4$. Given a constraint graph (left) derived from triangle constraints, we use $[1, 2, 4, 5, 3, 6, 8, 7]$ as a breadth-first traversal of nodes starting from the root node $u_0 = 1$. The algorithm begins with node $C_j = 1$ and attempts to create a cluster with the surrounding neighbours of C_j sorted by distance (Equation 15). Node $C_j = 1$ has neighbours 2, 4, and 5, and successfully forms a cluster $\{1, 2, 4, 5\}$ of exactly the maximum size (middle). The next three nodes in the breadth first search (2, 4, and 5) are skipped as they already belong to a cluster. The next node $C_j = 3$ is adjacent to all nodes but 1. While nodes 2, 4, 5 are already in a cluster, the remaining neighbours form the cluster $\{3, 6, 7, 8\}$ of maximum size $K_s = 4$. With all nodes claimed, the algorithm terminates.

desirable properties such as stability in the face of near incompressibility, its convergence behavior fares poorly. Early and mid stage solver iterations yield major artifacts such as significant volume loss, wrinkling, and oscillations in deformation. This behavior is attributed to the *decoupled* constraint formulation where the hydrostatic constraint $C^H(\mathbf{x})$ and the deviatoric constraint $C^D(\mathbf{x})$ are treated separately.

An important consequence of this decoupling is that rest configurations, that is, zero-energy deformations, result in non-zero constraint gradients $\nabla C^H(\mathbf{x})$ and $\nabla C^D(\mathbf{x})$. Hence, even small perturbations $\delta\mathbf{x}$ from a rest configuration $\mathbf{x}$ are magnified to non-trivial constraint projections $\Delta\mathbf{x}$ when fed into the XPBD solver. Because of the local nature of the XPBD solver, constraint projections $\Delta\mathbf{x}$ do not necessarily descend into solution iterates which simultaneously decrease the energy in surrounding constraints. In other words, the constraint projection $\Delta\mathbf{x}^H$ resulting from $C^H(\mathbf{x})$ might send $\mathbf{x}$ into directions of increasing $C^D(\mathbf{x})$ and vice versa.

It is important to note that this observation is not specific to the stable neo-Hookean constraints. This reasoning could be applied to any constraint type solved with XPBD's Gauss-Seidel like iterations. For example, geometric tetrahedral volume conservation constraints produce erratic gradient directions and must be coupled with tetrahedral mesh edge distance constraints for stabilization. However, in previous continuum-based constraint formulations where every finite element is associated with a single constraint, a configuration $\mathbf{x}$ such that $C_j(\mathbf{x}) = 0$ ensures an element rest configuration $\mathbf{x}$ such that $\int_{\Omega^e} \Psi(\mathbf{x}) \partial\Omega^e = 0$. In contrast, for the decoupled constraints $C^H(\mathbf{x})$ and $C^D(\mathbf{x})$, configurations $\mathbf{x}$ such that $C^H(\mathbf{x}) = 0$ or $\mathbf{x}$ such that $C^D(\mathbf{x}) = 0$ do not imply element rest configuration. Thus, in our method, we wish to find a constraint formulation such that configurations $\mathbf{x}$ where $C_j(\mathbf{x}) = 0$ imply element rest configurations.

Hence, we propose a coupled constraint formulation to address these instability issues. In our method, we allow vector-valued constraints $C^{\text{neo}} : \mathbb{R}^n \to \mathbb{R}^2$, which we refer to as *constraint blocks*, where

$$C^{\text{neo}}(\mathbf{x}) = \begin{bmatrix} C^H(\mathbf{x}) \\ C^D(\mathbf{x}) \end{bmatrix}. \tag{17}$$

Using the convention that gradients are row vectors, we can compute the 2-by-n Jacobian of C^{neo} as

$$\nabla C^{\text{neo}}(\mathbf{x}) = \begin{bmatrix} \nabla C^H(\mathbf{x}) \\ \nabla C^D(\mathbf{x}) \end{bmatrix}. \tag{18}$$

The Schur complement operator and the constraint compliance are now 2×2 matrices, such that

$$\mathbf{A} = (\nabla C^{\text{neo}}(\mathbf{x}))\, \mathbf{M}^{-1}\, (\nabla C^{\text{neo}}(\mathbf{x}))^T + \tilde{\alpha}^{\text{neo}},$$

and

$$\tilde{\alpha}^{\text{neo}} = \frac{1}{\Delta t^2} \begin{bmatrix} \alpha^H & 0 \\ 0 & \alpha^D \end{bmatrix},$$

where α^H and α^D are the compliance parameters associated with the hydrostatic and deviatoric constraints, respectively. Thus, constraint projection corrections incorporate information from both the hydrostatic constraint gradient and the deviatoric constraint gradient *simultaneously* with a simple 2-by-2 block solve.

When coupling constraints into a block solve, the resulting constraint block inherits the union of the sparsity patterns of all its child constraints. In the case of the neo-Hookean constraint pair, the hydrostatic and deviatoric constraints have exactly the same sparsity pattern, so the constraint block does not change the clustered graph topology when we treat these constraints as a coupled pair.

Hence, using constraint blocks C^{neo} does not affect any of our previous XPBD solver variants, graph coloring algorithms, and clustering algorithms, with the exception of requiring a 2×2 matrix solve in order to obtain our Lagrange multipliers corrections

$$\mathbf{A} \begin{bmatrix} \Delta\lambda^H \\ \Delta\lambda^D \end{bmatrix} = - \begin{bmatrix} C^H(x) \\ C^D(x) \end{bmatrix} - \tilde{\boldsymbol{\alpha}}^{\text{neo}} \begin{bmatrix} \lambda^H \\ \lambda^D \end{bmatrix}, \tag{19}$$

where λ^H and λ^D are the lagrange multipliers associated with hydrostatic and deviatoric constraints respectively, and $\tilde{\boldsymbol{\alpha}}^{\text{neo}}$ is the 2×2 compliance matrix associated with constraint block C^{neo}.

An LU solve with partial pivoting can be used to solve for the Lagrange multiplier corrections, which is equivalent to applying direct substitution to Equation 19 with improved conditioning. Although in practice, we have not encountered cases where the Schur complement matrix suffers ill-conditioning issues, in theory, it is easy to craft such ill-conditioned systems. A combination of varying nodal masses and Lamé coefficients directly influence conditioning of the 2×2 system.

6. Results & Discussion

In this section, we present several experiments that highlight the benefits of our graph clustering algorithm and blocked constraint solve for elastic-body simulations with the XPBD framework. We compare against a baseline solver that uses the stable neo-Hookean constraint formulation proposed by Macklin and Müller [17] and that applies a greedy graph coloring algorithm on the constraint graph to parallelize the solver. Animations of the experiments in this section can also be found in the supplementary video.

6.1. Implementation

CPU results were obtained using an 8-Core AMD Ryzen 7 3700X 3.60 GHz processor with 16GB of memory, while GPU results were obtained using an NVIDIA GeForce RTX 2060 Super processor. Our GPU implementation of XPBD is rather straightforward, storing simulation state and constraint parameters in global memory. The simulation state stores positions $\mathbf{x} \in \mathbb{R}^n$, velocities $\mathbf{v} \in \mathbb{R}^n$, forces $\mathbf{f} \in \mathbb{R}^n$, inverse masses $\boldsymbol{w} \in \mathbb{R}^n$ and Lagrange multipliers $\boldsymbol{\lambda} \in \mathbb{R}^m$. The constraint parameters store, for each element, quadrature weights $w_g \in \mathbb{R}$, quadrature points $X_g \in \mathbb{R}^3$, Lamé coefficients $\lambda, \mu \in \mathbb{R}$, basis function gradients $\nabla\phi(X_g) \in \mathbb{R}^3$ for each tetrahedron vertex, the 12 unique non-zero values of the deformation gradient derivatives $\frac{\partial \mathbf{F}}{\partial \mathbf{x}} \in \mathbb{R}^{3\times3\times12}$, as well as compliance parameters $\alpha^H, \alpha^D \in \mathbb{R}$ and $\mathscr{C}_j$, where $|\mathscr{C}_j| = 12$ for tetrahedral constraints. Clusters and cluster partitions must also be stored in order to implement parallel constraint projection. Additional book-keeping data structures are stored for indexing into the various GPU arrays. Table 3 exposes the GPU memory footprint of our XPBD implementation using graph clustering in various scenarios, showing that memory overhead is negligible even for large models.

Although much of the data stored on GPU is read-only, such as constraint parameters, clusters, partitions and indexing data structures, we do not exploit optimized read-only GPU memory regions, which our timings would no doubt benefit from. Our timings are also affected by the fact that we copy memory from GPU to CPU every time step in order to render our animations. Additionally, we use single precision floating point arithmetic.

Our CPU implementations of XPBD use the Eigen library [12] for linear algebra routines, and Intel TBB to parallelize various XPBD solver variants. CUDA 11.4 [24] was used

Table 2: Timing results using constraint blocking and clustering. The abbreviations used in column headings are C for clustered and B for blocked constraints. Timings are reported in milliseconds. Speedups are reported with respect to the baseline timings.

Model	Tetrahedra	Colors			Time (ms)				Speedup		
		Baseline	C	Reduction	Baseline	C	B	B+C	C	B	B+C
Beam	3727	40	17	0.43	285.03	170.49	41.29	20.46	1.67x	6.90x	13.93x
Bunny	56371	52	21	0.40	432.28	315.33	90.86	52.81	1.37x	4.76x	8.19x
Armadillo	45593	62	26	0.42	528.26	374.81	108.89	61.13	1.41x	4.85x	8.64x
Spot	19835	104	25	0.24	885.07	324.18	173.78	54.25	2.73x	5.09x	16.31x
Octopus	22213	78	22	0.28	663.96	299.78	132.76	49.10	2.21x	5.00x	13.52x
Squirrel	64768	56	22	0.39	457.91	332.16	94.78	55.41	1.38x	4.83x	8.26x

Table 3: Total GPU memory usage per tetrahedral mesh using our clustered parallel XPBD implementation with block neo-Hookean constraints.

Model	Vertices	Tetrahedra	Memory (Mb)
Armadillo	987	2922	0.63
Beam	936	3727	0.78
Spot	5135	19835	4.19
Octopus	6731	22213	4.79
Armadillo	9751	45593	9.50
Bunny	13808	56371	11.97
Squirrel	15408	64768	13.73
Armadillo	39062	162385	34.44

for our GPU implementations. We used polyscope [29] to render the animations, and TetWild [13] was used on hand-picked models from the Thingi10K dataset [34] to generate our tetrahedral meshes.

6.2. Performance

We test our clustering algorithm on tetrahedral models of varying size and topology. Table 2 summarizes the performance of blocked and clustered simulations compared to a baseline solver. The reported timings are the average execution time of each timestep over 300 frames using the GPU.

Time step and substeps. A time step $\Delta t = 0.01$ s is used for all performance experiments. The number of substeps changes depending on the example and whether clustering and blocked constraints are being used. We determine the number of substeps by a convergence analysis (see Figure 5), as well as qualitative visual assessments aiming to compare equivalent simulations. For all blocked simulations, 50 substeps are used, except the beam model, which was simulated with 30 substeps. Otherwise, 250 substeps are used for unblocked simulations, except the beam, which was simulated using 200 substeps.

Constraint graph processing. The block neo-Hookean constraints were used to build the constraint graph $\mathcal{G}$, such that each constraint is associated with a tetrahedral element. Prior to assembling $\mathcal{G}$, we reorder the vertices based on the smallest degree last ordering [19]. Then, we apply Algorithm 4 on $\mathcal{G}$ to obtain the supernodal graph $\bar{\mathcal{G}}$ using $K_s = 5$. By applying the same greedy graph coloring algorithm on both $\mathcal{G}$ and $\bar{\mathcal{G}}$, we observe that $\bar{\mathcal{G}}$ yields between 17 to 26 colors, while $\mathcal{G}$ yields between 40 and 104 colors, demonstrating reductions in number of colors between 24% and 41% of the original number of colors.

Effect of clustering. Figure 4 visualizes the partitions using the *jet* colormap for various tetrahedral models when clustering and graph coloring algorithms are applied. When a typical graph coloring algorithm is used, there are many more colors, indicating a larger number of partitions. Yet when clustering is applied, there are fewer colors, thus giving a significant improvement in solver performance. We can see from Table 2 that solve times have much higher correlation with the number of colors, rather than the size of the tetrahedral models. Indeed, one can see that the Spot model is simulated at 54 ms per time step, while the Bunny is actually simulated more efficiently at 52.81 ms per time step, even though the workload triples from 19.8k elements to 56.3k elements, because our clustering yields only 21 colors for the Bunny, which is less than the 25 colors obtained for the Spot model. Note that even though our clustering algorithm is highly effective in enhancing theoretical parallelism, we are still bounded by hardware capacity. Our parallel CPU implementations are limited to 16 threads, such

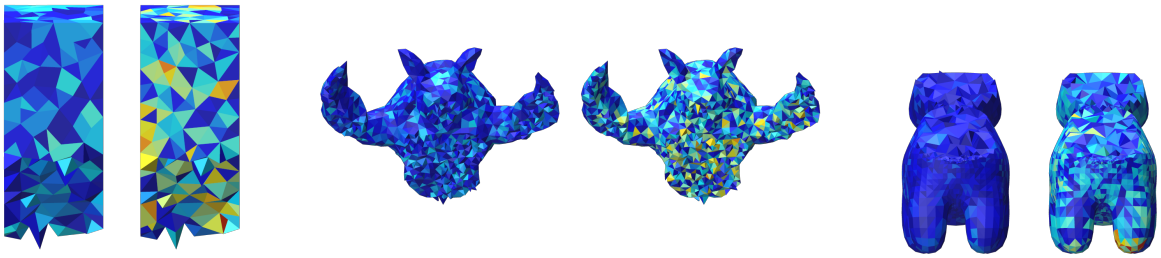

Fig. 4: Each pair of images shows constraint partitions generated with and without clustering for selected examples. Tetrahedral elements of the same color are solved in parallel. Using our clustering approach, there are fewer colors (left images) compared to partitions generated using typical graph coloring approaches (right images).

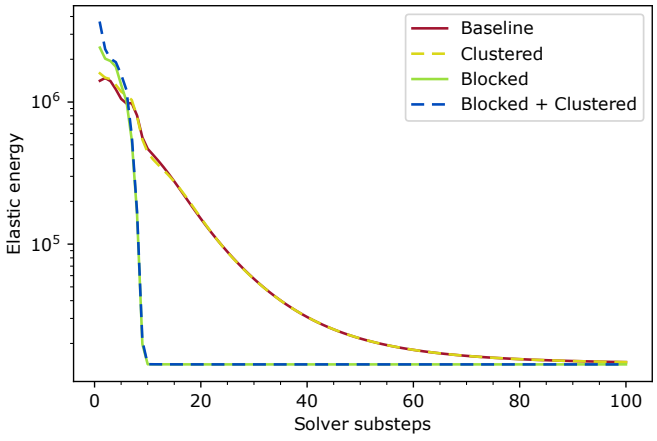

Fig. 5: Convergence for the Beam example using different GPU solver variants shows the superior behavior of block solves of coupled neo-Hookean constraints, while clustering does not hinder convergence.

that there is no speedup between Algorithms 2 and 3, although there are no slowdowns either. While our clustering method does not reach the lower bound obtainable using hexahedral meshes [17], it does not make any assumptions on the constraint model, thus generalizing to arbitrary constraint types.

Effect of coupled constraints. Solving for the neo-Hookean constraints using a block form has the most impact on the performance of all examples. This is due mainly to the notably improved convergence that is observed when the blocked constraint formulation is used. We analyze the convergence in further detail in the next section.

6.3. Solver Convergence

To evaluate solver convergence, we select a simulation frame from the Beam example in which the elastic body exhibits the highest total elastic potential energy and perform 100 solver substeps. We then evaluate the decrease in elastic energy with respect to solver iterations using the decoupled neo-Hookean constraint formulation, and again using our coupled constraints and graph clustering strategy. Recall that the beam is comprised of 3.7k elements with a Young's modulus $Y = 10^7$ Pa, Poisson ratio $\nu = 0.45$, and mass density $\rho = 10^3$ kg m^{-3}. The results of our implementations are shown in Figure 5.

In a single substep, the coupled constraint solve reduces the elastic energy at least one order of magnitude faster than the decoupled approach. This convergence improvement occurs regardless of whether clustering is being used to parallelize the solver or not. After approximately 10 substeps, the elastic energy is reduced to convergence for the blocked constraints, while the decoupled constraint formulation still has not converged after 100 substeps. In our experiments, wrinkling artefacts on the beam model remain noticeable using the decoupled constraint model, and close to 200 solver substeps are required to attain the same visual quality as the blocked solve.

To further reinforce the implications of convergence improvements using our coupled formulations, we conduct experiments in which the same computational budget is allotted to the baseline configuration and its corresponding blocked version. Figures 6 and 7 show the results. For the same material and simulation parameters, tetrahedral meshes in blue use our coupled formulation, while pink ones use the decoupled formulation from Macklin and Müller [17]. The decoupled formulation fails to remain stable given such a tight computational budget, whereas our coupled formulation produces a stable simulation.

The improved convergence due to using coupled constraints is typically an order of magnitude faster than the baseline solver with decoupled constraints, and performance is even further accelerated by using constraint clustering. The

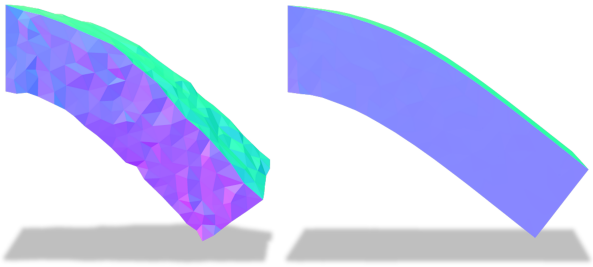

Fig. 6: The Beam example simulated using the baseline method (left) and the block neo-Hookean constraints (right). Both simulations use equivalent time step, 30 substeps, and 1 iteration per substep. The normal map of the boundary surface is displayed to highlight the differences between the two simulations.

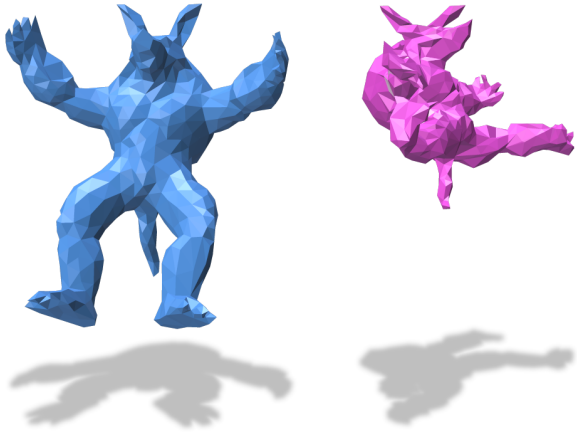

Fig. 7: Our coupled neo-Hookean constraint formulation remains stable using 20 substeps for a 2.9k element Armadillo model, while the decoupled constraints fail for the same computational budget. Simulation with coupled constraints (left) and with decoupled constraints (right).

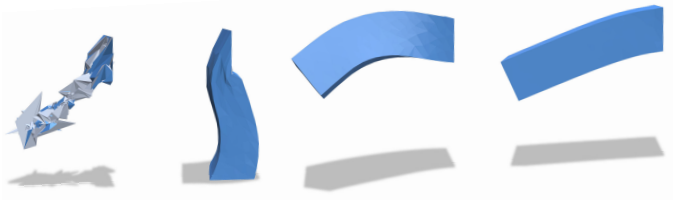

Fig. 8: Crazy scramble unfolding test. Our constraint formulation recovers a smooth shape from degenerate configurations using 30 substeps. Different frames of the simulation are displayed in chronological order from left to right.

block solve plays a significant role in improving the performance due to its better convergence behavior, whereas the clustering consistently reduces computational time through parallelization. Implementing our method requires only minor changes to an existing XPBD framework, specifically 2×2 block solves during constraint projection and a simple breadth-first search algorithm applied to the constraint graph.

6.4. Robustness

As a sanity check, we conduct an experiment to validate that our blocked constraints preserve the robustness of Macklin and Müller [17]. We re-use the bending beam example and scramble the vertex positions erratically. Then, we simulate using our blocked neo-Hookean constraints using 30 substeps and $\frac{1}{60}$ s time steps. Our blocked model recovers from degenerate configurations as expected due to the volume conservation properties of neo-Hookean materials. Using only 30 substeps, our blocked model recovers its initial smooth geometry. Selected frames from this experiment are shown in Figure 8.

7. Conclusion & Future Work

In this paper, we introduce a novel approach to parallelizing XPBD solvers for arbitrary constraint types based on graph clustering. Our graph clustering method enhances the computational efficiency of existing XPBD frameworks using graph coloring based partitioning, and significant performance gains were observed in all our experiments. Furthermore, we introduce a coupled constraint formulation for neo-Hookean materials that only requires using 2×2 linear solves during the constraint projection step. The computational overhead of the approach is negligible, but it is quite

effective for improving solver convergence compared to recent formulations. Combining these two contributions, we observe an order of magnitude faster performance compared to a baseline XPBD solver.

Our experiments did not include dynamic constraints, which frequently occur when contacts are simulated. In this case, an efficient algorithm is required to introduce these dynamic constraints to the supernodal graph such that they too can be solved for in parallel. Additionally, although our clustering approach is general, greedily forming clusters in heterogeneous constraint graphs may be unstable, since constraint projection order affects convergence.

Our approach could further benefit from more sophisticated clustering algorithms, which have been presented in the literature on graph clustering [27]. This could reduce the number of constraint partitions even further compared to our greedy approach, though care must be taken, since the constraint graphs for soft-body simulations are particularly dense and uniform, rendering clustering algorithms which assume non-uniformity ineffective.

Other constraint formulations could also benefit from a coupled blocked solve. However, as the coupled constraint blocks get larger, time complexity of block solving increases as a cubic polynomial if direct solves are used. If iterative solvers are employed, computational cost may be reduced in exchange for approximate solutions. Designing judicious blocking strategies is thus an interesting area of future research for highly parallel local solvers.

Acknowledgements

This work was supported by NSERC grant no. ALLRP-570702-21.

References

[1] Steven S. An, Theodore Kim, and Doug L. James. 2008. Optimizing Cubature for Efficient Integration of Subspace Deformations. *ACM Trans. Graph.* 27, 5, Article 165 (dec 2008), 10 pages. https://doi.org/10.1145/1409060.1409118

[2] Jernej Barbič and Doug L. James. 2005. Real-Time Subspace Integration for St. Venant-Kirchhoff Deformable Models. *ACM Trans. Graph.* 24, 3 (jul 2005), 982–990. https://doi.org/10.1145/1073204.1073300

[3] Jan Bender, Dan Koschier, Patrick Charrier, and Daniel Weber. 2014. Position-based simulation of continuous materials. *Computers & Graphics* 44 (2014), 1–10. https://doi.org/10.1016/j.cag.2014.07.004

[4] Jan Bender, Matthias Müller, Miguel A. Otaduy, Matthias Teschner, and Miles Macklin. 2014. A Survey on Position-Based Simulation Methods in Computer Graphics. *Computer Graphics Forum* 33, 6 (2014), 228–251. https://doi.org/10.1111/cgf.12346

[5] Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. 2014. Projective Dynamics: Fusing Constraint Projections for Fast Simulation. *ACM Trans. Graph.* 33, 4, Article 154 (jul 2014), 11 pages. https://doi.org/10.1145/2601097.2601116

[6] Christopher Brandt, Elmar Eisemann, and Klaus Hildebrandt. 2018. Hyper-Reduced Projective Dynamics. *ACM Trans. Graph.* 37, 4, Article 80 (jul 2018), 13 pages. https://doi.org/10.1145/3197517.3201387

[7] Marco Fratarcangeli and Fabio Pellacini. 2013. A GPU-Based Implementation of Position Based Dynamics for Interactive Deformable Bodies. *Journal of Graphics Tools* 17, 3 (2013), 59–66. https://doi.org/10.1080/2165347X.2015.1030525

[8] Marco Fratarcangeli and Fabio Pellacini. 2015. Scalable Partitioning for Parallel Position Based Dynamics. *Computer Graphics Forum* 34, 2 (may 2015), 405–413. https://doi.org/10.1111/cgf.12570

[9] Marco Fratarcangeli, Valentina Tibaldo, and Fabio Pellacini. 2016. Vivace: A Practical Gauss-seidel Method for Stable Soft Body Dynamics. *ACM Trans. Graph.* 35, 6, Article 214 (Nov. 2016), 9 pages. https://doi.org/10.1145/2980179.2982437

[10] Marco Fratarcangeli, Huamin Wang, and Yin Yang. 2018. Parallel Iterative Solvers for Real-time Elastic Deformations. In *SIGGRAPH Asia 2018 Courses* (Tokyo, Japan) *(SA '18)*. Article 14, 45 pages. https://doi.org/10.1145/3277644.3277779

[11] Lawson Fulton, Vismay Modi, David Duvenaud, David I. W. Levin, and Alec Jacobson. 2019. Latent-space Dynamics for Reduced Deformable Simulation. *Computer Graphics Forum* 38, 2 (2019), 379–391. https://doi.org/10.1111/cgf.13645

[12] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

[13] Yixin Hu, Qingnan Zhou, Xifeng Gao, Alec Jacobson, Denis Zorin, and Daniele Panozzo. 2018. Tetrahedral Meshing in the Wild. *ACM Trans. Graph.* 37, 4, Article 60 (July 2018), 14 pages. https://doi.org/10.1145/3197517.3201353

[14] Theodore Kim and Doug L. James. 2009. Skipping Steps in Deformable Simulation with Online Model Reduction. *ACM Trans. Graph.* 28, 5 (dec 2009), 1–9. https://doi.org/10.1145/1618452.1618469

[15] Yinchu Liu and Sheldon Andrews. 2022. Graph Partitioning Algorithms for Rigid Body Simulations. In *Eurographics 2022 - Short Papers*. https://doi.org/10.2312/egs.20221036

[16] Miles Macklin, Matthias Müller, and Nuttapong Chentanez. 2016. XPBD: Position-Based Simulation of Compliant Constrained Dynamics. In *Proceedings of the 9th International Conference on Motion in Games (MIG '16)*. 49–54. https://doi.org/10.1145/2994258.2994272

[17] Miles Macklin and Matthias Müller. 2021. A Constraint-Based Formulation of Stable Neo-Hookean Materials. In *Motion, Interaction and Games (MIG '21)*. Article 12, 7 pages. https://doi.org/10.1145/3487983.3488289

[18] Miles Macklin, Kier Storey, Michelle Lu, Pierre Terdiman, Nuttapong Chentanez, Stefan Jeschke, and Matthias Müller. 2019. Small Steps in Physics Simulation. In *Proceedings of the 18th Annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '19)*. Article 2, 7 pages. https://doi.org/10.1145/3309486.3340247

[19] David W Matula and Leland L Beck. 1983. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM (JACM)* 30, 3 (1983), 417–427.

[20] David W. Matula, George Marble, and Joel D. Isaacson. 1972. Graph coloring algorithms. In *Graph Theory and Computing*, Ronald C. Read (Ed.). Academic Press, 109–122. https://doi.org/10.1016/B978-1-4832-3187-7.50015-5

[21] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. 2007. Position based dynamics. *Journal of Visual Communication and Image Representation* 18, 2 (2007), 109–118. https://doi.org/10.1016/j.jvcir.2007.01.005

[22] Matthias Müller, Miles Macklin, Nuttapong Chentanez, and Stefan Jeschke. 2022. Physically Based Shape Matching. *Computer Graphics Forum* (2022). https://doi.org/10.1111/cgf.14618

[23] Matthias Müller, Miles Macklin, Nuttapong Chentanez, Stefan Jeschke, and Tae-Yong Kim. 2020. Detailed Rigid Body Simulation with Extended Position Based Dynamics. *Computer Graphics Forum* 39, 8 (2020), 101–112. https://doi.org/10.1111/cgf.14105

[24] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2020. CUDA, release: 10.2.89. https://developer.nvidia.com/cuda-toolkit

[25] Albert Peiret, Sheldon Andrews, József Kövecses, Paul G. Kry, and Marek Teichmann. 2019. Schur Complement-Based Substructuring of Stiff Multibody Systems with Contact. *ACM Trans. Graph.* 38, 5, Article 150 (oct 2019), 17 pages. https://doi.org/10.1145/3355621

[26] Yue Peng, Bailin Deng, Juyong Zhang, Fanyu Geng, Wenjie Qin, and Ligang Liu. 2018. Anderson Acceleration for Geometry Optimization and Physics Simulation. *ACM Trans. Graph.* 37, 4, Article 42 (jul 2018), 14 pages. https://doi.org/10.1145/3197517.3201290

[27] Satu Elisa Schaeffer. 2007. Graph clustering. *Computer Science Review* 1, 1 (2007), 27–64. https://doi.org/10.1016/j.cosrev.2007.05.001

[28] Martin Servin, Claude Lacoursière, and Niklas Melin. 2006. Interactive simulation of elastic deformable materials.. In *SIGRAD 2006. The Annual SIGRAD Conference; Special Theme: Computer Games*.

[29] Nicholas Sharp et al. 2019. Polyscope. www.polyscope.run.

[30] Siyuan Shen, Yin Yang, Tianjia Shao, He Wang, Chenfanfu Jiang, Lei Lan, and Kun Zhou. 2021. High-Order Differentiable Autoencoder for Nonlinear Model Reduction. *ACM Trans. Graph.* 40, 4, Article 68 (jul 2021), 15 pages. https://doi.org/10.1145/3450626.3459754

[31] Breannan Smith, Fernando De Goes, and Theodore Kim. 2018. Stable Neo-Hookean Flesh Simulation. *ACM Trans. Graph.* 37, 2, Article 12 (mar 2018), 15 pages. https://doi.org/10.1145/3180491

[32] Maxime Tournier, Matthieu Nesme, Benjamin Gilles, and François Faure. 2015. Stable Constrained Dynamics. *ACM Trans. Graph.* 34, 4, Article 132 (jul 2015), 10 pages. https://doi.org/10.1145/2766969

[33] Huamin Wang. 2015. A Chebyshev Semi-Iterative Approach for Accelerating Projective and Position-Based Dynamics. *ACM Trans. Graph.* 34, 6, Article 246 (oct 2015), 9 pages. https://doi.org/10.1145/2816795.2818063

[34] Qingnan Zhou and Alec Jacobson. 2016. Thingi10K: A Dataset of 10,000 3D-Printing Models. *arXiv preprint arXiv:1605.04797* (2016).