A matrix-free approach for finite-strain hyperelastic problems using geometric multigrid

SCHOLARONE™
Manuscripts

RESEARCH ARTICLE

A matrix-free approach for finite-strain hyperelastic problems using geometric multigrid

Denis Davydov*[1] | Jean-Paul Pelteret[1] | Daniel Arndt[2,3] | Martin Kronbichler[4] | Paul Steinmann[1,5]

[1]Chair of Applied Mechanics, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

[2]Interdisciplinary Center for Scientific Computing (IWR), Heidelberg University, Germany

[3]Computational Engineering and Energy Sciences Group, Oak Ridge National Laboratory, Oak Ridge, TN, USA

[4]Institute for Computational Mechanics, Technical University of Munich, Germany

[5]Glasgow Computational Engineering Center (GCEC), University of Glasgow, United Kingdom

Correspondence
*D. Davydov, Egerlandstr. 5, 91058 Erlangen, Germany. Email: denis.davydov@fau.de

Present Address
Chair of Applied Mechanics, Friedrich-Alexander-Universität Erlangen-Nürnberg, Egerlandstr. 5, 91058 Erlangen, Germany.

Summary

This work investigates matrix-free algorithms for problems in quasi-static finite-strain hyperelasticity. Iterative solvers with matrix-free operator evaluation have emerged as an attractive alternative to sparse matrices in the fluid dynamics and wave propagation communities because they significantly reduce the memory traffic, the limiting factor in classical finite element solvers. Specifically, we study different matrix-free realizations of the finite element tangent operator, and determine whether generalized methods of incorporating complex constitutive behavior might be feasible. In order to improve the convergence behavior of iterative solvers, we also propose a method by which to construct level tangent operators and employ them to define a geometric multigrid preconditioner. The performance of the matrix-free operator and the geometric multigrid preconditioner is compared to the matrix-based implementation with an algebraic multigrid preconditioner on a single node for a representative numerical example of a heterogeneous hyperelastic material in two and three dimensions. We find that matrix-free methods for finite-strain solid mechanics are very promising, outperforming linear matrix-based schemes by two to five times, and that is it possible to develop numerically efficient implementations that are independent of the hyperelastic constitutive law.

KEYWORDS:
adaptive finite element method ; geometric multigrid ; finite-strain ; matrix-free ; hyperelasticity

1 | INTRODUCTION

The performance of finite element solvers on modern computer architectures is typically limited by the bandwidth from RAM memory for sufficiently large problems. This is because loading pre-computed matrix elements from memory into the CPU cores is significantly slower than actually performing the arithmetic operations, dictated by hardware properties. In order to improve the performance of iterative solvers, so-called matrix-free methods, where matrix-vector products are formed on-the-fly rather than relying on precomputed matrix entries [1,2,3,4,5], are widely adopted in the fluid mechanics community. Such methods can also be efficiently implemented on GPUs [6,7]. Among matrix-free methods, stencil-based variants are most popular for linear elements

---

2 |

and tetrahedral shapes[5,8], whereas implementations based on integration with sum-factorization techniques are attractive for higher polynomial degrees $p \geq 2$ and tensor product shape functions[1,2,3,9].

Within the solid mechanics community, however, matrix-free methods are currently not widely adopted. One reason is that the tangent operators[1] derived from linearization of the nonlinear balance equations are more elaborate as compared to the common operators in linear and non-linear fluid mechanics. For matrix-free schemes using fast integration via sum factorization, the main challenge is to find a representation of the tangent operator at the quadrature points, either in the referential or current configuration, that is cheap enough to be evaluated on the fly. Another reason could be the frequent use of lower-order elements in solid mechanics due to lacking smoothness of the underlying solution. The finite element method (FEM) with linear or quadratic elements (that reduce the potential for locking), or mixed or enhanced formulations (that avoid locking) are often adopted, whereas higher order elements are rarely employed. This can potentially reduce the advantage of matrix-free methods using sum-factorization techniques, which are generally more competitive for higher order elements[2,10,11,12]. However, we note that, even with linear FEM, large scale computations with as many as 10^{12} unknowns are possible with a matrix-free approach but infeasible with sparse matrices as there is simply not enough memory to store the sparse tangent matrix even on large supercomputers[5]. Therefore, we believe that for large-scale computations in solid mechanics it is crucial to consider matrix-free approaches and develop efficient solvers.

This work aims to fill this gap by investigating different matrix-free representations of the finite-strain hyperelastic tangent operator with respect to the computational cost and the performance on a single node of a high performance computing cluster. Particular emphasis is on the tradeoff between computations at quadrature points versus pre-computation with a higher memory access in terms of the representation of the tangent operator. The `deal.II` finite element library[13], which provides MPI parallelization together with a generalized matrix-free framework that incorporates SIMD vectorization and an interface to high-performance MPI-distributed matrix-based libraries, is used for this study. In order to improve the convergence of iterative solvers, we also propose a method by which to construct level matrix-free tangent operators and employ them to define a geometric multigrid preconditioner.

The paper is organized as follows: In Section 2 we briefly introduce the partial differential equations used in finite-strain solid mechanics, while section 3 covers the finite element discretization. In Section 4 we describe the different matrix-free implementations that are evaluated in this study. Section 5 provides details of the numerical framework this study relies on, where we also propose a method by which to construct level tangent operators for a geometric multigrid preconditioner that is suitable for the matrix-free implementation of finite-strain hyperelasticity. The performance of the matrix-free tangent operator and the geometric multigrid preconditioner is compared to the matrix-based implementation with an algebraic multigrid preconditioner for a representative numerical example of a heterogeneous hyperelastic material simulated in two and three dimensions in Section 6. Finally, the results are summarized in Section 7.

2 | THEORETICAL BACKGROUND

2.1 | Weak form

The deformation of a body $\mathcal{B}$ from the referential configuration $\mathcal{B}_0$ to the current configuration $\mathcal{B}_t$ at time t is defined via the deformation map $\boldsymbol{\varphi}_t : \mathcal{B}_0 \rightarrow \mathcal{B}_t$, which places material points of $\mathcal{B}$ into the Euclidean space $\mathbb{E}^3$. The spatial location of a material particle $\boldsymbol{X}$ is given by $\boldsymbol{x} = \boldsymbol{\varphi}_t(\boldsymbol{X})$. The displacement field reads $\boldsymbol{u} = \boldsymbol{x} - \boldsymbol{X}$.

The tangent map is linear such that $\mathrm{d}\boldsymbol{x} = \boldsymbol{F} \cdot \mathrm{d}\boldsymbol{X}$, where $\boldsymbol{F} := \mathrm{Grad}\,\boldsymbol{\varphi} \equiv \boldsymbol{I} + \mathrm{Grad}\,\boldsymbol{u}$ is called the deformation gradient and $\boldsymbol{I}$ is the second-order unit tensor. The mapping has to be one-to-one and must exclude self-penetration. Consequently, the Jacobian determinant $J = \det \boldsymbol{F} > 0$ has to be positive. We shall denote the (contravariant) push-forward transformation of rank-2 and rank-4 tensors $\mathbf{A}$ and $\mathcal{A}$ by a linear transformation χ defined as

$$\chi\left(\mathbf{A}\right)_{ij} = F_{iA} A_{AB} F_{jB} \tag{1}$$

$$\chi\left(\mathcal{A}\right)_{ijkl} = F_{iA} F_{jB} \mathcal{A}_{ABCD} F_{kC} F_{lD}. \tag{2}$$

Note that $\mathrm{d}\boldsymbol{x} = \chi(\mathrm{d}\boldsymbol{X})$.

---

[1]Here and below 'operator' denotes a finite-dimensional linear operator defined on the vector space $\mathbb{R}^N$, where N is the number of unknown degrees of freedom in the FE discretization. Although any linear mapping on a finite-dimensional space is representable by a matrix, we adopt the 'operator' term to avoid confusion between its matrix-free and matrix-based numerical implementation.

In what follows, we parameterize the material behavior using the right Cauchy-Green tensor $\boldsymbol{C} := \boldsymbol{F}^T \cdot \boldsymbol{F}$. We will also use the Green-Lagrange strain tensor $\boldsymbol{E} := \frac{1}{2}[\boldsymbol{C} - \boldsymbol{I}]$ and the left Cauchy-Green tensor $\boldsymbol{b} := \boldsymbol{F} \cdot \boldsymbol{F}^T$. Clearly $J = \sqrt{\det \boldsymbol{C}}$ and $\partial(\bullet)/\partial \boldsymbol{E} = 2\,\partial(\bullet)/\partial \boldsymbol{C}$. For conservative systems without body forces, the total potential energy functional $\mathcal{E}$ is introduced as

$$\mathcal{E} = \int_{\mathcal{B}_0} \psi(\boldsymbol{F})\,\mathrm{d}V - \int_{\partial\mathcal{B}_0^N} \overline{\boldsymbol{T}} \cdot \boldsymbol{u}\,\mathrm{d}S\,, \tag{3}$$

where $\overline{\boldsymbol{T}}$ is the prescribed loading at the Neumann part of the boundary $\partial\mathcal{B}_0^N$ in the referential configuration, ψ denotes the strain-energy density per unit reference volume and $\boldsymbol{u}$ should satisfy the prescribed Dirichlet boundary conditions $\boldsymbol{u} = \overline{\boldsymbol{u}}$ on $\partial\mathcal{B}_0^D := \partial\mathcal{B}_0 \setminus \partial\mathcal{B}_0^N \neq \emptyset$.

The principle of stationary potential energy at equilibrium requires that the directional derivative with respect to the displacement

$$\mathrm{D}_{\delta\boldsymbol{u}}\mathcal{E} := \frac{\mathrm{d}}{\mathrm{d}\epsilon}\mathcal{E}(\boldsymbol{u} + \epsilon\delta\boldsymbol{u})\Big|_{\epsilon=0} = 0 \tag{4}$$

vanishes for all admissible directions $\delta\boldsymbol{u}$ which satisfy homogeneous Dirichlet boundary conditions. This leads to the following scalar-valued non-linear equation

$$F(\boldsymbol{u}, \delta\boldsymbol{u}) = \int_{\mathcal{B}_0} \boldsymbol{P} : \mathrm{Grad}\,\delta\boldsymbol{u}\,\mathrm{d}V - \int_{\partial\mathcal{B}_0^N} \overline{\boldsymbol{T}} \cdot \delta\boldsymbol{u}\,\mathrm{d}S = 0\,, \tag{5}$$

where $\boldsymbol{P} := \partial\psi/\partial\boldsymbol{F}$ is the Piola stress tensor. The double contraction in the first term can be re-written in terms of the symmetric Kirchhoff stress tensor $\boldsymbol{\tau} := \boldsymbol{P} \cdot \boldsymbol{F}^T$ as

$$\boldsymbol{P} : \mathrm{Grad}\,\delta\boldsymbol{u} = \left[\boldsymbol{\tau} \cdot \boldsymbol{F}^{-T}\right] : \mathrm{Grad}\,\delta\boldsymbol{u} = \boldsymbol{\tau} : \left[\mathrm{Grad}\,\delta\boldsymbol{u} \cdot \boldsymbol{F}^{-1}\right] = \boldsymbol{\tau} : \mathrm{grad}\,\delta\boldsymbol{u}\,, \tag{6}$$

and therefore

$$F(\boldsymbol{u}, \delta\boldsymbol{u}) = \int_{\mathcal{B}_0} \boldsymbol{\tau} : \mathrm{grad}^s\,\delta\boldsymbol{u}\,\mathrm{d}V - \int_{\partial\mathcal{B}_0^N} \overline{\boldsymbol{T}} \cdot \delta\boldsymbol{u}\,\mathrm{d}S = 0\,, \tag{7}$$

where $\mathrm{grad}^s\,\delta\boldsymbol{u} := \frac{1}{2}\left[\mathrm{grad}\,\delta\boldsymbol{u} + (\mathrm{grad}\,\delta\boldsymbol{u})^T\right]$ is involved due to the symmetry of $\boldsymbol{\tau}$. Here, 'grad' denotes the gradient with respect to the current configuration, as opposed to the gradient in the referential setting 'Grad'. Note that $\boldsymbol{\tau}$ is the (contravariant) push-forward transformation of the Piola-Kirchhoff stress $\boldsymbol{S} := \partial\psi/\partial\boldsymbol{E} \equiv 2\partial\psi/\partial\boldsymbol{C}$, that is, $\boldsymbol{\tau} = \chi(\boldsymbol{S}) = \boldsymbol{F} \cdot \boldsymbol{S} \cdot \boldsymbol{F}^T$.

2.2 | Linearization

In order to solve (7) using Newton's method, a first order approximation around a given solution point (a field) $\overline{\boldsymbol{u}}$ is required such that

$$F(\overline{\boldsymbol{u}} + \Delta\boldsymbol{u}, \delta\boldsymbol{u}) \approx F(\overline{\boldsymbol{u}}, \delta\boldsymbol{u}) + \mathrm{D}_{\Delta\boldsymbol{u}}F(\overline{\boldsymbol{u}}, \delta\boldsymbol{u}), \tag{8}$$

where $\mathrm{D}_{\Delta\boldsymbol{u}}(\bullet)$ denotes the directional derivative in the direction $\Delta\boldsymbol{u}$. For conservative traction boundary conditions, the directional derivative is given by

$$\begin{aligned}
\mathrm{D}_{\Delta\boldsymbol{u}}F(\overline{\boldsymbol{u}}, \delta\boldsymbol{u}) = &\int_{\mathcal{B}_0} \mathrm{D}_{\Delta\boldsymbol{u}}\left(\boldsymbol{F} \cdot \boldsymbol{S} \cdot \boldsymbol{F}^T\right) : \overline{\mathrm{grad}^s}\,\delta\boldsymbol{u}\,\mathrm{d}V \\
&+ \int_{\mathcal{B}_0} \overline{\boldsymbol{\tau}} : \left[\mathrm{Grad}\,\delta\boldsymbol{u} \cdot \mathrm{D}_{\Delta\boldsymbol{u}}\boldsymbol{F}^{-1}\right]\mathrm{d}V\,.
\end{aligned} \tag{9}$$

Following [14], this can be simplified to[2]

$$
\mathrm{D}_{\Delta \boldsymbol{u}} F(\overline{\boldsymbol{u}}, \delta \boldsymbol{u}) = \int_{\mathcal{B}_0} \overline{\mathrm{grad}^s} \Delta \boldsymbol{u} \; : \; \boldsymbol{JC} \; : \; \overline{\mathrm{grad}^s} \, \delta \boldsymbol{u} \, \mathrm{d}V
$$
$$
+ \int_{\mathcal{B}_0} \overline{\mathrm{grad}} \delta \boldsymbol{u} \; : \; \left[\overline{\mathrm{grad} \Delta \boldsymbol{u} \cdot \overline{\boldsymbol{\tau}}} \right] \mathrm{d}V.
\tag{10}
$$

Here $\overline{(\bullet)}$ is used to denote quantities evaluated using the displacement field $\overline{\boldsymbol{u}}$, and the fourth-order material part of the spatial tangent stiffness tensor is the (contravariant) push-forward of the material part of the referential tangent stiffness tensor $\boldsymbol{JC} = \chi \left(4 \frac{d^2 \psi(\mathbf{C})}{d\mathbf{C} \otimes d\mathbf{C}} \right)$. The first term in (10) is related to the linearization of the Piola-Kirchhoff stress $\boldsymbol{S}$ and is therefore called material part of the directional derivative. The second term in (10) is called the geometric part of the directional derivative since it originates from the linearization of $\boldsymbol{F}$, $\boldsymbol{F}^T$ and $\boldsymbol{F}^{-1}$.

2.3 | Constitutive modelling

For the current study, we use the compressible Neo-Hookean model

$$
\psi(\mathbf{C}) = \frac{\mu}{2} \left[\mathrm{tr}\, \mathbf{C} - \mathrm{tr}\, \mathbf{I} - 2 \ln(J) \right] + \lambda \ln^2(J),
\tag{11}
$$

where μ and λ denote the shear modulus and Lamé parameter respectively. Derived using statistical thermodynamics applied to cross-linked polymers, the Neo-Hookean model [15,16] is commonly used to model rubber-like materials in the finite-strain regime. It can be shown that the first and second derivatives of the strain energy function are given by

$$
\frac{d\psi(\mathbf{C})}{d\mathbf{C}} = \frac{\mu}{2} \mathbf{I} - \frac{1}{2} \left[\mu - 2\lambda \ln(J) \right] \mathbf{C}^{-1},
\tag{12}
$$

$$
\frac{d^2 \psi(\mathbf{C})}{d\mathbf{C} \otimes d\mathbf{C}} = \frac{1}{2} \left[\mu - 2\lambda \ln(J) \right] \left[-\frac{d\mathbf{C}^{-1}}{d\mathbf{C}} \right] + \frac{\lambda}{2} \mathbf{C}^{-1} \otimes \mathbf{C}^{-1}.
\tag{13}
$$

The Kirchhoff stress and its associated fourth-order material part of the spatial tangent tensor are

$$
\boldsymbol{\tau} \equiv J\boldsymbol{\sigma} = \chi \left(2 \frac{d\psi(\mathbf{C})}{d\mathbf{C}} \right) = \mu \mathbf{b} - \left[\mu - 2\lambda \ln(J) \right] \mathbf{I}
\tag{14}
$$

and

$$
\boldsymbol{JC} = \chi \left(4 \frac{d^2 \psi(\mathbf{C})}{d\mathbf{C} \otimes d\mathbf{C}} \right) = 2 \left[\mu - 2\lambda \ln(J) \right] \boldsymbol{S} + 2\lambda \mathbf{I} \otimes \mathbf{I},
\tag{15}
$$

where $\boldsymbol{S}$ is the fourth-order symmetric identity tensor with $S_{ijkl} = \frac{1}{2} \left[\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk} \right]$. The action that $\boldsymbol{JC}$ performs when contracted with an arbitrary rank-2 symmetric tensor is therefore

$$
\boldsymbol{JC} : (\bullet) = 2 \left[\mu - 2\lambda \ln(J) \right] (\bullet)^s + 2\lambda \, \mathrm{tr}\,(\bullet) \mathbf{I}.
\tag{16}
$$

3 | FINITE ELEMENT DISCRETIZATION

We now introduce a FE triangulation $\mathcal{B}_0^h$ of $\mathcal{B}_0$ and the associated FE space of continuous piecewise elements with polynomial space of fixed degree p. The displacement fields are given in a vector space spanned by standard vector–valued FE basis functions $\boldsymbol{N}_i(\boldsymbol{x})$ (e.g. polynomials with local support on a patch of elements):

$$
\boldsymbol{u}^h =: \sum_{i \in \mathcal{I}} u_i \boldsymbol{N}_i(\boldsymbol{X}) \qquad \delta \boldsymbol{u}^h =: \sum_{i \in \mathcal{I}} \delta u_i \boldsymbol{N}_i(\boldsymbol{X}),
\tag{17}
$$

where the superscript h denotes that this representation is related to the FE mesh with size function $h(\boldsymbol{X})$ and $\mathcal{I}$ is the set of unknown degrees of freedom (DoF).

---

[2]To that end, $\mathrm{D}_{\Delta \boldsymbol{u}} \boldsymbol{E} = \frac{1}{2} \left[\mathrm{D}_{\Delta \boldsymbol{u}} \boldsymbol{F}^T \cdot \boldsymbol{F} + \boldsymbol{F}^T \cdot \mathrm{D}_{\Delta \boldsymbol{u}} \boldsymbol{F} \right]$, $\mathrm{D}_{\Delta \boldsymbol{u}} \boldsymbol{F} = \mathrm{Grad} \Delta \boldsymbol{u}$ and $\mathrm{D}_{\Delta \boldsymbol{u}} \boldsymbol{F}^{-1} = -\boldsymbol{F}^{-1} \cdot \mathrm{D}_{\Delta \boldsymbol{u}} \boldsymbol{F} \cdot \boldsymbol{F}^{-1}$ are employed together with $\mathrm{D}_{\Delta \boldsymbol{u}} \boldsymbol{S} = 2 \partial \boldsymbol{S} / \partial \boldsymbol{C} : \mathrm{D}_{\Delta \boldsymbol{u}} \boldsymbol{E}$.

The Newton–Raphson solution approach is adopted for the nonlinear problem considered here. Given the current trial solution field $\overline{\boldsymbol{u}^h}$, the correction $\Delta \boldsymbol{u}^h$ field is obtained as the solution to the following system of equations (see (7) and (10)):

$$\sum_{j \in \mathcal{I}} A_{ij} \Delta u_j = -F_i, \tag{18}$$

$$A_{ij} \equiv a(\boldsymbol{N}_i, \boldsymbol{N}_j) = \int_{B_0} \underbrace{\overline{\mathrm{grad}^s \boldsymbol{N}_i} : \boldsymbol{JC} : \overline{\mathrm{grad}^s}\, \boldsymbol{N}_j + \overline{\mathrm{grad}}\boldsymbol{N}_i : \left[\overline{\mathrm{grad}\boldsymbol{N}_j \cdot \overline{\boldsymbol{\tau}}}\right]}_{\tilde{a}(\boldsymbol{N}_i, \boldsymbol{N}_j)} \mathrm{d}V, \tag{19}$$

$$F_i = \int_{B_0} \overline{\boldsymbol{\tau}} : \mathrm{grad}^s\, \boldsymbol{N}_i \, \mathrm{d}V - \int_{\partial B_0^N} \overline{\boldsymbol{T}} \cdot \boldsymbol{N}_i \, \mathrm{d}S. \tag{20}$$

Here, $\boldsymbol{A}$ with elements A_{ij} is the discrete tangent operator, $\boldsymbol{F}$ with elements F_i is the discrete gradient of the potential energy and $\tilde{a}(\boldsymbol{N}_i, \boldsymbol{N}_j)$ is a representation for the integrand in the bilinear form $a(\boldsymbol{N}_i, \boldsymbol{N}_j)$ for the trial solution field $\overline{\boldsymbol{u}^h}$. Note that $\overline{\mathrm{grad}^s \boldsymbol{N}_i}$ and $\overline{\mathrm{grad}\boldsymbol{N}_i}$ are gradients of shape functions in the current configuration, whereas the integration is performed in the referential configuration.

4 | MATRIX-FREE OPERATOR EVALUATION

Classically, in order to solve (18) the matrix $\boldsymbol{A}$ and the residual force vector $\boldsymbol{F}$ corresponding to the discretization are assembled (that is, the non-zero matrix elements A_{ij} are individually computed and stored) and a direct or iterative solver is used to solve the linear system. In this case, the matrix-vector product $\boldsymbol{Ax}$ results from the product of the individual matrix elements with the elements in the vector, and is usually implemented with the aid of specialized linear algebra packages. On modern computer architectures however, loading sparse matrix data into the CPU registers is significantly slower than performing the associated arithmetic operations. For this reason, recent implementations often focus on so-called matrix-free approaches [2,10] where the matrix is not assembled but rather the result of its operation on a vector is directly calculated. The idea is to perform the operations within the solver on-the-fly rather than loading matrix elements from memory. For iterative solvers, this is possible since it is sufficient to compute matrix-vector products. The matrix-vector product $\boldsymbol{Ax}$ (i.e. the action of operator $\boldsymbol{A}$ on a vector $\boldsymbol{x}$) can be expressed by

$$\begin{aligned} (\boldsymbol{Ax})_i &= \sum_j a(\boldsymbol{N}_i, \boldsymbol{N}_j)x_j \\ &\approx \sum_K \sum_q \sum_j \tilde{a}(\boldsymbol{N}_i, \boldsymbol{N}_j)(\boldsymbol{\xi}_q)x_j w_q J_q^K \end{aligned} \tag{21}$$

where $\tilde{a}(\boldsymbol{N}_i, \boldsymbol{N}_j)(\boldsymbol{\xi}_q)$ is a representation for the evaluation of the bilinear form at one quadrature point $\boldsymbol{\xi}_q$, J_q^K is the determinant of the Jacobian of the mapping from the isoparametric element to the element K and w_q is the quadrature weight. For the sake of demonstration, let us assume that $\boldsymbol{A}$ represents a mass operator with scalar valued shape functions $\{N_i(\boldsymbol{x})\}$. In this case, it holds that

$$\tilde{a}(N_i, N_j)(\boldsymbol{\xi}_q) = N_i(\boldsymbol{\xi}_q)N_j(\boldsymbol{\xi}_q).$$

Further assuming that the number of quadrature points n in one-dimension matches the degree of the ansatz space plus one, $n = p + 1$, the evaluation of all of the shape functions (n^d) in all quadrature points (n^d) has complexity $\mathcal{O}(n^{2d})$ in d space dimensions. For the total matrix-vector product we get a cost of $\mathcal{O}(n^{2d})$ per element, the same as for a regular matrix-vector product where the entries are already precomputed. Assuming tensor product ansatz spaces and a tensor product quadrature rule, evaluation of this operations can be performed more efficiently using a technique called "sum factorization".

Let us illustrate this in 2D. Tensor-product quadrature points can be expressed as a combination of one dimensional quadrature points

$$\boldsymbol{\xi}_q = (\widetilde{\xi}_{q_1}, \widetilde{\xi}_{q_2}), \tag{22}$$

where for each quadrature points q we can associate a multi-index (q_1, q_2). Weights associated with the quadrature formula w_q can be expressed as a product of one-dimensional weights

$$w_q = \widetilde{w}_{q_1} \widetilde{w}_{q_2}. \tag{23}$$

6 |

(a) Nodes of cubic Lagrange element
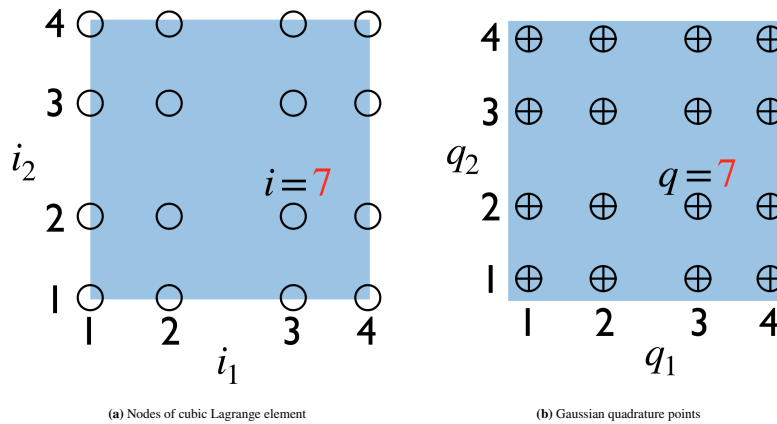
(b) Gaussian quadrature points

FIGURE 1 Illustration of tensorial indexing for nodes and quadrature points. In this sketch a node/quadrature point with the number 7 can be indexed with a multi-index $\{3, 2\}$.

Shape functions can also be expressed as product of one dimensional basis functions

$$N_i(\boldsymbol{\xi}_q) = \widetilde{N}_{i_1}(\widetilde{\xi}_{q_1})\widetilde{N}_{i_2}(\widetilde{\xi}_{q_2}) \tag{24}$$

where for each DoF index i we can associate a multi-index (i_1, i_2). See Figure 1 for an illustration. Therefore, the result of the application of the scalar-valued mass operator, represented as a vector y_i, on a single element K can be written as

$$y_i \equiv Y_{i_1 i_2} = \sum_{q_1}\sum_{q_2}\sum_{j_1}\sum_{j_2} \widetilde{N}_{i_1}(\widetilde{\xi}_{q_1})\widetilde{N}_{i_2}(\widetilde{\xi}_{q_2})\widetilde{N}_{j_1}(\widetilde{\xi}_{q_1})\widetilde{N}_{j_2}(\widetilde{\xi}_{q_2})X_{j_1 j_2}\widetilde{w}_{q_1}\widetilde{w}_{q_2}J^K_{q_1 q_2}$$

$$= \sum_{q_1}\widetilde{N}_{i_1}(\widetilde{\xi}_{q_1})\widetilde{w}_{q_1}\sum_{q_2}\widetilde{N}_{i_2}(\widetilde{\xi}_{q_2})\widetilde{w}_{q_2}J^K_{q_1 q_2}\left[\sum_{j_1}\widetilde{N}_{j_1}(\widetilde{\xi}_{q_1})\sum_{j_2}\widetilde{N}_{j_2}(\widetilde{\xi}_{q_2})X_{j_1 j_2}\right] \forall i_1\, i_2$$

Note that here $x_j \equiv X_{j_1 j_2}$; the former accesses the element source vector using a linear index j, whereas the latter uses the associated multi-index accesses $\{j_1 j_2\}$. These four loops all have the same structure. We either keep the shape function fixed and iterate over the quadrature points in one spatial direction (in the first two sums) or keep the quadrature point fixed and iterate over all the shape functions in one spatial direction (in the last two sums). Since each of these $2 \times d$ loops has a complexity of n and we perform them for n^d values in quadrature points or values in degrees of freedoms simultaneously, this results in an algorithm with an arithmetic complexity of $\mathcal{O}(n^{d+1})$. Hence, we can expect that such a matrix-free approach is faster than a regular matrix-based approach, especially for the 3D case, and where the discretization and evaluation employs higher degree basis functions with the corresponding quadrature rule. More information on matrix-free techniques can be found in [2,17].

For the performance of the matrix-free operator evaluation, it is crucial to find a good intermediate point between precomputing everything (caching) and evaluating all the quantities on-the-fly. In order to obtain the gradients with respect to the current configuration, which appear in the tangent operator (19) of the finite-strain elasticity problem derived with a Lagrangian formulation, we use the *MappingQEulerian* class from the `deal.II`[13] library. In addition to the standard mapping from the reference (isoparametric) element to the element in real space, this class computes the mapping to the current configuration given a displacement field. Due to the specifics of matrix-free operator evaluation implemented in `deal.II`, the integration is eventually performed over the current configuration. Therefore, we have to additionally divide by the Jacobian J of the deformation map at a given quadrature point to express the integral in the referential configuration.

Below, we propose three algorithms to implement the tangent operator of finite-strain elasticity (see (19)) using the matrix-free approach, each of which have a different level of abstraction, algorithmic complexity, and memory requirements. They are introduced in order of lowest memory footprint to largest:

Algorithm 1 caches a single scalar which involves the logarithm of the Jacobian – a computationally demanding operation compared to additions and multiplications. As evident from Section 2, we need to re-evaluate the Kirchhoff stress at each quadrature point in order to apply the tangent operator. This in turn requires evaluating the deformation gradient F and therefore gradients with respect to the referential configuration.

In order to avoid the repeated calculation of the Kirchhoff stress, Algorithm 2 caches its value for each element and quadrature point. In this case, we also avoid the need to evaluate gradients of the displacement field with respect to the referential

---

Algorithm 1: Matrix-free tangent operator: cache scalar quantities only

---

    Given : Source FE vector x, current FE solution $\overline{u}^h$, cached $c_1 := \mu - 2\lambda \log(J)$ for each cell and quadrature point

    Return: action of the FE tangent operator (19) on x

1   zero destination vector $y = 0$;

2   update ghost values for source vector x with MPI ;

3   **foreach** *element $K \in \Omega^h$* **do**

4   gather local vector values on this element $\overline{u}^h{}_K$, x_K ;

5   evaluate the following tensors at each quadrature point using sum factorization :

6 $\mathrm{Grad}\,\overline{u}^h{}_K$;   `// 2nd order`

7 $g := \overline{\mathrm{grad}x}_K$;   `// 2nd order`

8   **foreach** *quadrature point q on K* **do**

9   evaluate $F = I + \mathrm{Grad}\,\overline{u}^h$;   `// 2nd order`

10   evaluate $J = \det(F)$;   `// scalar`

11   evaluate $b = F \cdot F^T$;   `// 2nd order symmetric`

12   evaluate $\tau = \mu b - c_1 I$;   `// 2nd order symmetric`

13   evaluate $\mathcal{G}g := g \cdot \tau / J$;   `// 2nd order`

14   evaluate $g_s = (g + g^T)/2$;   `// 2nd order symmetric`

15   evaluate $Cg_s := \left[2c_1 g_s + 2\lambda \,\mathrm{tr}(g_s)I\right]/J$;   `// 2nd order symmetric`

16   queue $Cg_s + \mathcal{G}g$ for contraction $\overline{\mathrm{grad}N}_i : (Cg_s + \mathcal{G}g)$;

17   **end**

18   evaluate queued contractions using sum factorization ;

19   scatter results to the destination vector

20   **end**

21   compress remote integral contributions into y via MPI ;

---

configuration. This algorithm utilizes the chosen constitutive relationship in that the operation of JC remains directly expressed using (16). Therefore, the action of the material part of the fourth-order spatial tangent stiffness tensor JC on the second-order symmetric tensor $\overline{\mathrm{grad}^s x}$ is cheap to evaluate. To that end, we cache two scalars that depend on the Jacobian determinant J.

Finally, Algorithm 3 represents the most general case which does not assume any form of the material part of the fourth-order spatial tangent stiffness tensor. In this case, we cache the fourth-order symmetric tensor C and the Kirchhoff stress τ for each element and quadrature point, and perform the double contraction with the second-order symmetric tensor $\overline{\mathrm{grad}^s x}$ on-the-fly.

Note that the SIMD vectorization in deal.II[13] is applied at the finite element level, i.e. the matrix-free operator is applied simultaneously on several elements (called "blocks"). The main reason is that, apart from the point-wise computation of the stresses and corresponding tangent operators, the operations are typically the same on all elements, as opposed to the operations within an element[3]. For a given quadrature point, we simultaneously perform tensor evaluations and contractions on all elements[4]. For example, the deformation gradient $F = I + \mathrm{Grad}\,\overline{u}^h$ (second-order tensor) is evaluated at the specific quadrature point of all elements within the "block". This is achieved using templated number types within the *Tensor* class of the deal.II library. In particular, deal.II provides the generic templated class *VectorizedArray* that defines a unified interface to a vectorized data type and overloads basic arithmetic operations based on compiler intrinsics. The core of the sum factorization algorithms is provided by the *FEEvaluation* class, that supports all the necessary contractions such as $\overline{\mathrm{grad}N}_i : (Cg_s + \mathcal{G}g)$ to implement the three algorithms proposed above. Thanks to the object-oriented design of the deal.II library, the core of those matrix-free algorithms can be implemented in only a handful of lines. Distributed memory MPI parallelization of the matrix-free operators is done using the standard domain decomposition technique, where each MPI process is responsible for applying the operator only on the locally owned elements. For more information about the implementation details and data structures for

---

[3]In general, not all quadrature points in each element group are endowed with identical, non-dissipative constitutive laws that follow the same code path during evaluation or the kinetic quantities and tangents. This motivates the caching strategy employed in Algorithm 3, as the pre-caching of the chosen data ensures a context in which SIMD parallelization of subsequent operations is guaranteed. Algorithms 1 and 2 sacrifice some of this generality for decreased memory requirements.

[4]For the heterogeneous materials considered here, the elements are first grouped according to the material type (matrix or inclusion) and then the SIMD vectorization is applied for each group of elements. This functionality is provided by the deal.II library in the *MatrixFree* class.

8 |

Algorithm 2: Matrix-free tangent operator: cache second-order Kirchhoff stress $\boldsymbol{\tau}$ and thereby avoid the need to evaluate referential quantities like $\boldsymbol{F}$ at quadrature points when executed.

    Given : Source FE vector $\boldsymbol{x}$, cached $c_1 := 2\left[\mu - 2\lambda\log(J)\right]/J$, $c_2 := 2\lambda/J$ and $\boldsymbol{\tau}/J$ for each cell and quadrature point, evaluated based on $\overline{\boldsymbol{u}^h}$

    Return: action of the FE tangent operator (19) on $\boldsymbol{x}$

1 zero destination vector $\boldsymbol{y} = 0$;

2 update ghost values for source vector $\boldsymbol{x}$ with MPI;

3 **foreach** *element $K \in \Omega^h$* **do**

4   gather local vector values on this element $\boldsymbol{x}_K$;

5   evaluate the following tensor at each quadrature point using sum factorization :

6 $\boldsymbol{g} := \overline{\text{grad}}\boldsymbol{x}_K$;   `// 2nd order`

7   **foreach** *quadrature point q on K* **do**

8   evaluate $\mathcal{G}\boldsymbol{g} := \boldsymbol{g} \cdot \left[\boldsymbol{\tau}/J\right]$;   `// 2nd order`

9   evaluate $\boldsymbol{g}_s = (\boldsymbol{g} + \boldsymbol{g}^T)/2$;   `// 2nd order symmetric`

10   evaluate $\boldsymbol{C}\boldsymbol{g}_s := c_1\boldsymbol{g}_s + c_2\,\text{tr}(\boldsymbol{g}_s)\boldsymbol{I}$;   `// 2nd order symmetric`

11   queue $\boldsymbol{C}\boldsymbol{g}_s + \mathcal{G}\boldsymbol{g}$ for contraction $\overline{\text{grad}}N_i : (\boldsymbol{C}\boldsymbol{g}_s + \mathcal{G}\boldsymbol{g})$;

12   **end**

13   evaluate queued contractions using sum factorization ;

14   scatter results to the destination vector

15 **end**

16 compress remote integral contributions into $\boldsymbol{y}$ via MPI ;

Algorithm 3: Matrix-free tangent operator: cache material part of the fourth-order spatial tangent stiffness tensor $\boldsymbol{C}$ and Kirchhoff stress $\boldsymbol{\tau}$.

    Given : Source FE vector $\boldsymbol{x}$, cached $\boldsymbol{\tau}/J$ and $\boldsymbol{C}$ for each cell and quadrature point, evaluated based on $\overline{\boldsymbol{u}^h}$

    Return: action of the FE tangent operator (19) on $\boldsymbol{x}$

1 zero destination vector $\boldsymbol{y} = 0$;

2 update ghost values for source vector $\boldsymbol{x}$ with MPI ;

3 **foreach** *element $K \in \Omega^h$* **do**

4   gather local vector values on this element $\boldsymbol{x}_K$;

5   evaluate the following tensor at each quadrature point using sum factorization :

6 $\boldsymbol{g} := \overline{\text{grad}}\boldsymbol{x}_K$;   `// 2nd order`

7   **foreach** *quadrature point q on K* **do**

8   evaluate $\mathcal{G}\boldsymbol{g} := \boldsymbol{g} \cdot \left[\boldsymbol{\tau}/J\right]$;   `// 2nd order`

9   evaluate $\boldsymbol{g}_s = (\boldsymbol{g} + \boldsymbol{g}^T)/2$;   `// 2nd order symmetric`

10   evaluate $\boldsymbol{C}\boldsymbol{g}_s := \boldsymbol{C} : \boldsymbol{g}_s$;   `// 2nd order symmetric`

11   queue $\boldsymbol{C}\boldsymbol{g}_s + \mathcal{G}\boldsymbol{g}$ for contraction $\overline{\text{grad}}N_i : (\boldsymbol{C}\boldsymbol{g}_s + \mathcal{G}\boldsymbol{g})$;

12   **end**

13   evaluate queued contractions using sum factorization ;

14   scatter results to the destination vector

15 **end**

16 compress remote integral contributions into $\boldsymbol{y}$ via MPI ;

---

sum factorization approaches in `deal.II` (including hanging constraints, storage of the inverse Jacobian of the transformation from unit to real cell, MPI and distributed memory parallelization as well as SIMD vectorization) and performance tests, see[2].

    Table 1 summarizes the three algorithms in terms of their implied memory access for one operator evaluation. The vector access in Algorithm 1 is higher than in the other variants because both the source vector $\boldsymbol{x}$ and the current FE solution $\overline{\boldsymbol{u}^h}$ need

TABLE 1 Details of memory access for matrix-free evaluation with Algorithms 1–3.

	Vector access double / node	physics terms double / q-point	metric terms grad/Grad double / q-point
Algorithm 1	$2d$ read, d write	1	up to $d^2 + d^2 + 1$
Algorithm 2	d read, d write	$2 + \frac{d(d+1)}{2}$	up to $d^2 + 1$
Algorithm 3	d read, d write	$\frac{d(d+1)}{2} + \left(\frac{d(d+1)}{2}\right)^2$	up to $d^2 + 1$

to be accessed, each with the usual indirect addressing pertaining to continuous finite element fields[2]. The cached quantities listed in the algorithms, labeled "physics terms" in the table, are increasing in size with more caching, and are stored for each quadrature point. The evaluation of the gradients in the referential and current configuration also involves a data access via the so-called metric terms, the inverse of the Jacobian from the unit element to the referential or current configuration, respectively. Algorithm 1 involves both the Jacobian in the referential and the current configuration, whereas the other two algorithms only involve the spatial (Eulerian) terms. Finally, the implementation framework also holds the determinant of the Jacobian for each point. Following[2], some compression may be applied to the metric terms: On affine geometries, the Jacobian is the same at all quadrature points of a cell, and needs only be stored once. Therefore, the table lists those quantities as "up to" $d^2 + 1$ terms, with a memory access that depends on the mesh via a run-time decision in the code.

5 | DESCRIPTION OF THE UTILIZED NUMERICAL FRAMEWORK

The discretized system of linear equations formulated in Section 3, in conjunction with the constitutive model described in Section 2.3, has been implemented using[5] `deal.II` version 9.1[13], an open-source finite element library. This library offers a number of paradigms by which to parallelize a finite-element code, and as a member of the xSDK (Extreme-scale Scientific Software Development Kit) ecosystem[18] aims to support exascale computing. In this work we employ `deal.II`'s interface to the `p4est`[19] library to perform a standard domain decomposition, whereafter each MPI process "owns" only a subset of elements and the mesh is distributed across all MPI processes. Figure 2 illustrates a distributed MPI decomposition on the fine mesh
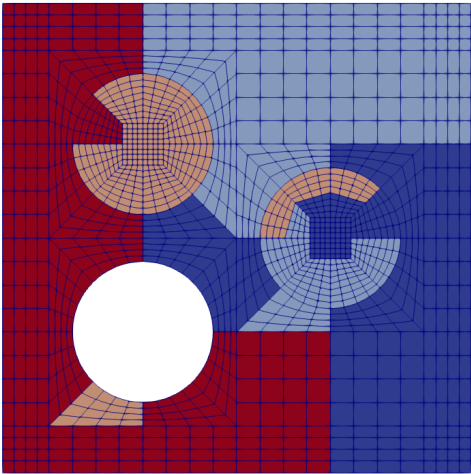

FIGURE 2 2D mesh after two global refinements distributed into three MPI processes. The color indicates the process owning the element.

level for the 2D problem that will be introduced later in Section 6.

---

[5]Including additional performance improvements for `Tensor` classes https://github.com/dealii/dealii/pull/8428.

In practice, this implies that parallelization of the assembly operation (for the matrix-based approach) and the linear solver, both of which are the focal points of investigation in this manuscript, has been achieved using MPI; various pre- and post-processing steps also leveraged parallelization. For the linear solver, `deal.II`'s implementation of the preconditioned conjugate gradient (CG) solver for symmetric positive-definite systems is consistently used.

The discrete tangent operator described by (19) can be implemented in one of two ways, namely through a classical matrix-based approach or, alternatively, a matrix-free approach. For the matrix-based approach adopted in this study, the Trilinos[20] library has been leveraged; the system tangent matrix is stored in a distributed *Epetra_FECrsMatrix* sparse data structure from the Epetra package[21] and the linear solver is preconditioned with an algebraic multigrid (AMG) preconditioner from the ML package[22]. This is the most highly performant matrix-based approach using the Trilinos framework that, to date, is available in `deal.II`.

The second implementation utilizes a matrix-free approach in conjunction with a geometric-multigrid (GMG) preconditioner. GMG preconditioners[23,24,25,3] can be proven to result in iteration counts that are independent of the number of mesh refinements by smoothing the residual on a hierarchy of meshes. For further information about geometric multigrid methods we refer the reader to[24,26,27]. In this work, we adopt the matrix-free level operators within the GMG. In general, given a heterogeneous non-linear material, the definition of transfer operators[6] (for restriction and prolongation) and level operators is not trivial. One approach is to start from an arbitrary heterogeneous material at the fine scale and employ homogenization theory[28,29,30,31] to design suitable transfer operators[32,33]. Such operators are, however, computationally very demanding. In this work we avoid these complications by assuming that the mesh at the coarsest level can accurately describe the heterogeneous finite-strain elastic material. In this case, the restriction and prolongation operations for patches of elements are always performed for the same material and thus admits usage of standard geometric transfer operators. We define the level operators as follows: Recall that the fine scale tangent operator (19) is obtained by linearization around the current displacement field $\overline{\boldsymbol{u}^h}$. We then restrict this displacement field to all multigrid levels $\{l\}$ and evaluate tangent operators using the matrix-free algorithms from Section 4. Essentially, level operators correspond to the linearization around the smoothed representation of the displacement field. Note that this differs from the classical approach where the (tangent) operator $\boldsymbol{A}^{l+1}$ on level $l+1$ is directly related to the (tangent) operator $\boldsymbol{A}^l$ on level l via $\boldsymbol{A}^{l+1} = \boldsymbol{I}_l^{l+1}\boldsymbol{A}^l\boldsymbol{I}_{l+1}^l$, where $\boldsymbol{I}_{l+1}^l$ and $\boldsymbol{I}_l^{l+1}$ are global prolongation (coarse-to-fine) and restriction (fine-to-coarse) operators, respectively.

The other parts of the GMG preconditioner such as level smoothers, matrix-free interlevel transfer operators, and the coarse level iterative solver are provided by the `deal.II` library; implementational details are discussed in[34]. The efficiency of the GMG preconditioner with the considered level operators will be assessed by numerical examples in the next section. We will, however, refrain from studying the machine performance aspects of the multigrid preconditioner as a whole as the focus of this paper is the matrix-free implementation of tangent operators of finite-strain elasticity both at the fine mesh level as well as the coarser multigrid levels. Finally, we note that a *p* version of multigrid[35] would also be conceivable but is not considered here.

6 | NUMERICAL EXAMPLES

In this section, we evaluate the proposed matrix-free algorithms for finite-strain hyperelastic materials as well as the geometric multigrid preconditioner on a benchmark problem from[32]. The coarse meshes for the 2D and 3D problems are illustrated in Figure 3 . The 2D material consists of a square (depicted in blue), a hole and two spherical inclusions (depicted in red). The size of the square domain is 10^{-3}mm. The matrix material is taken to have Poisson's ratio 0.3 and shear modulus $\mu = 0.4225 \times 10^6$ N/mm^2. The inclusion is taken to be 100 times stiffer. The 3D material is obtained by extrusion of the 2D geometry into the third dimension. Note that although a relatively coarse mesh is used at the coarsest multigrid level, the geometry of the inclusions is captured accurately by manifold descriptions of the boundaries and interfaces. Both domains are fully fixed along the bottom surface and a distributed load is applied at the top in the $(1, 0)$ or $(1, 1, 0)$ directions for the 2D and the 3D problems, respectively. The force density 12.5×10^3N/mm^2 or $12.5\sqrt{2} \times 10^3$N/mm^3 is applied in 5 steps for the 2D and the 3D problems, respectively. With respect to the nonlinear solver, the displacement tolerance of the ℓ_2 norm for the Newton update is taken to be 10^{-5} whereas the relative and absolute tolerances for the residual forces is 10^{-8}. With the chosen criteria it takes 4 Newton–Raphson iterations to converge within a load step for the 2D problem and 3 iterations for the 3D setup. The relative convergence

---

[6]Finite dimensional linear operators from a vector space associated with the fine-scale mesh to the coarse-scale mesh, and in the opposite direction.
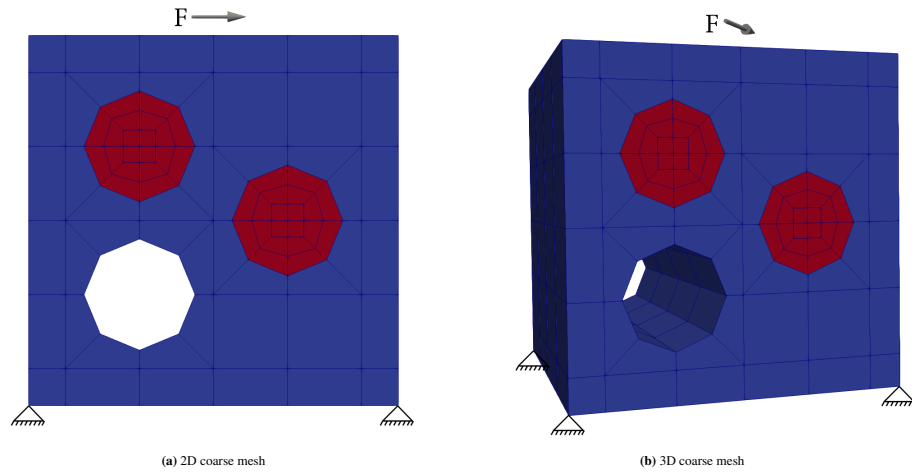
(a) 2D coarse mesh   (b) 3D coarse mesh

FIGURE 3 Discretization of the heterogeneous material at the coarsest mesh level and the prescribed boundary conditions.

criteria for the linear solver is 10^{-6}. Figure 4 shows deformed meshes at the final loading step with quadratic elements and two global mesh refinements.
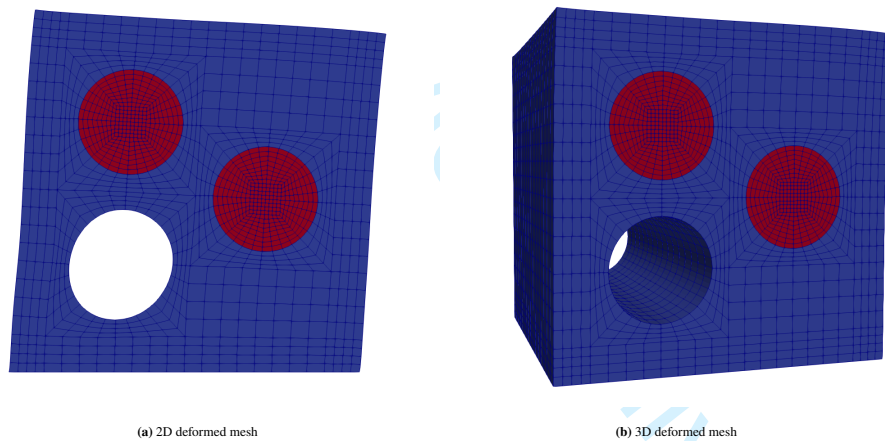

(a) 2D deformed mesh   (b) 3D deformed mesh

FIGURE 4 Deformed meshes at the final loading step with quadratic elements and two global mesh refinements.

Aligned with the approach taken in similar previous investigations, compare e.g. [10], we limit ourselves to comparisons made on a node level in this study. Examinations performed in this manner are most representative for the differences between the matrix-free and matrix-based approaches because problems of sufficient size are dominated by the local node-level performance with multigrid solvers. Multigrid solvers with small enough coarse-grid problems are well-known to be scalable even to the largest supercomputers[36,37] as the communication is mostly between nearest neighbors. The benchmark calculations are performed for 2D and 3D problems with various combinations of polynomial degrees and number of global refinements, as is stated in Tables 2 a and 2 b. We choose a problem size on the order of 10^6 DoFs for each test case as this is an upper limit due to the memory requirements of the matrix-based approach with a high polynomial degree. Even so, the problem size is large enough so that the sparse matrix (in excess of 1 Gb when using the lowest polynomial order finite element discretization) will not fit into the L3 cache (on the order of tens of Mb in size), and therefore we can observe that the matrix-based approach is memory bound. Furthermore, the data structures of the matrix-free approaches are also too large to fit completely into caches.

We conduct two performance studies using the described discretization, the first being for matrix-vector multiplication only, and the second for the preconditioned iterative linear solver. The results of the matrix-free approach with a geometric multigrid

12 |

TABLE 2 Parameters for the benchmark: p is the polynomial degree, q is the number of quadrature points in 1D, N_{gref} is the number of global mesh refinements, N_{el} is the number of elements and N_{DoF} is the number of DoFs.

(a) 2D

p	q	N_{gref}	N_{el}	N_{DoF}
1	2	7	1441792	2887680
2	3	6	360448	2887680
3	4	5	90112	1625088
4	5	5	90112	2887680
5	6	5	90112	4510720
6	7	4	22528	1625088
7	8	4	22528	2211328
8	9	4	22528	2887680

(b) 3D

p	q	N_{gref}	N_{el}	N_{DoF}
1	2	4	1441792	4442880
2	3	3	180224	4442880
3	4	2	22528	1891008
4	5	2	22528	4442880

preconditioner are compared to the matrix-based approach in conjunction with the algebraic multigrid (AMG) preconditioner using the package ML[22] of the Trilinos[20] library version 12.12.1. The aggregation threshold for the AMG preconditioner is taken as 10^{-4}. The geometric multigrid algorithm uses a V-cycle with pre- and post-smoothing using Chebyshev polynomials[38] of fifth degree, provided by the deal.II library via the *PreconditionChebyshev* class. The largest eigenvalue $\lambda_{\max}$ of the level operators is estimated from 30 iterations of the CG solver and the smoother is set to target the range $[0.06\lambda_{\max}, 1.2\lambda_{\max}]$. On the coarsest level, we use the Chebyshev preconditioner as a solver[38] to reduce the residual by three orders of magnitude. Since this smoother only uses diagonal elements of the operator, it is readily compatible with the matrix-free approach to leverage fast matrix-vector products, as opposed to most default smoothers which rely on an explicit matrix form.

Computations were performed on two systems: A dual-socket Intel Xeon Gold 6230 ("Cascade Lake"), released in 2019, with 20 cores per socket and 96 GB of DDR4-2933 memory (nominal bandwidth per socket: 141 GB/s). Turbo mode is enabled with a power limit of 135 watt per socket, which results in the processor running at 2.8 GHz for scalar code and 2.0 GHz for AVX-512-heavy code in the experiments. The GNU compiler version 9.1.0 with flags "-O3 -march=skylake-avx512" and OpenMPI version 4.0.1 were used, with the code run in pure MPI mode. The "Westmere" results were obtained on a single machine with eight Intel Xeon E7-8870 "Westmere" chips, released in 2011, (10 cores per chip + SMT) running at 2.4 GHz with 30 MB shared cache per chip. For the simulation, GCC version 8.1.0 with flags "-O3 -march=native" and OpenMPI version 3.0.0 were used. Unless noted otherwise, the simulations are performed with 20 MPI processes, i.e., one socket for Cascade Lake and two sockets for Westmere.

6.1 | Matrix-vector multiplication

Figures 5 and 6 show the results of the matrix-vector multiplication for the considered finite-strain hyperelastic benchmark problem on the two hardware systems. We are interested in the wall-clock time (average over 10 runs for each linear solver step) per DoF as a first metric. As expected, the matrix-vector multiplication becomes very expensive for higher polynomial degrees for sparse matrix-based approaches, as the matrix becomes more and more dense. In order to increase visibility, the figures also report the throughput of the matrix vector product, computed by the ratio of the number of degrees of freedom over the time for a matrix-vector product, in panels (a) and (d). The performance results for finite-strain elasticity operators considered in this study are qualitatively similar to those reported for the Laplace operator in[2]. Note that we consider only deformed meshes together with separate metric terms according to Table 1 . These deformed terms are the result of manifold descriptions of the boundaries and interfaces to capture the geometry as well as the linearization-related mapping required for evaluation of gradients with respect to the current configuration.

The matrix-free implementation is faster than the matrix-based counterpart already for a bi-quadratic Lagrangian basis in 2D and a tri-quadratic basis in 3D on both systems. The "MF tensor2" and "MF scalar" strategies outperform the sparse matrix already for tri-linear elements of polynomial degree $p = 1$ in 3D on Cascade Lake architecture. For example for the "tensor2" matrix-free implementation with $p = 4$, we record a 5× higher throughput than the matrix-based evaluation of $p = 1$. The advantage of the matrix-free implementation can be explained by the memory access per unknown shown in the panels (c) and (f) of the figures, as the time per unknown in panels (b) and (e) closely follows the memory access. Figures 5 b, 5 e, 6 b and 6 e clearly show the influence of the different caching strategies, described in Algorithms 1, 2 and 3. Two conclusions can be
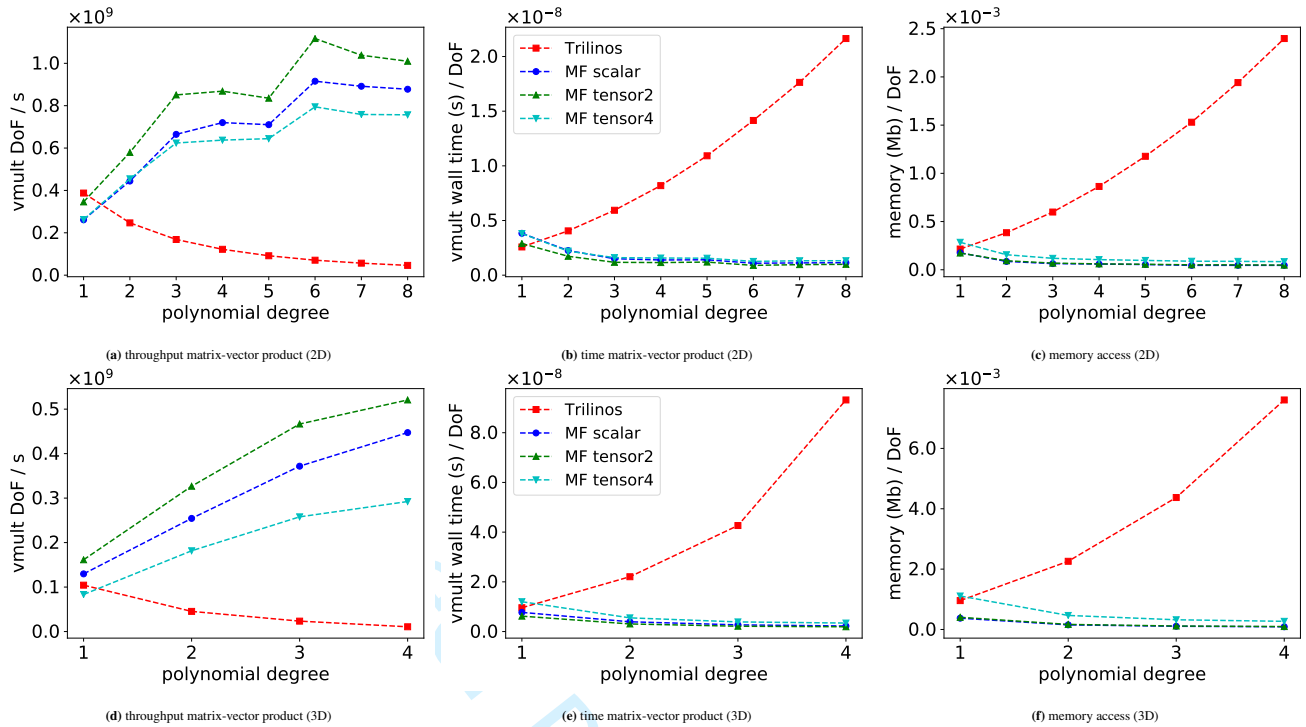
(a) throughput matrix-vector product (2D) **(b)** time matrix-vector product (2D) **(c)** memory access (2D)

(d) throughput matrix-vector product (3D) **(e)** time matrix-vector product (3D) **(f)** memory access (3D)

FIGURE 5 Cascade Lake, one socket with 20 cores, matrix-vector multiplication.

drawn from these results: Firstly, the algorithm where we only cache the second-order Kirchhoff tensor and utilize the chosen hyperelastic constitutive equation to efficiently implement the action of the material part of the fourth-order spatial tangent stiffness tensor on the second-order symmetric tensor ("tensor2") is the fastest approach in both 2D and 3D and on both systems. When compared to the scalar caching implementation, the "tensor2" variant loads slightly less data, because only the current configuration is needed for the metric terms, as compared to both the spatial and the referential configuration in the scalar caching case in order to evaluate the Kirchhoff stress τ. Besides the memory access, the scalar caching strategy also involves considerably more computations at quadrature points. On the newer Cascade Lake system with a Flop/Byte ratio of 9.1 for the theoretical peak values, the scalar caching strategy is closer to the "tensor2" variant. On the Westmere machine, the Flop/Byte ratio is 2.9, which means that much fewer computations can be sustained for the same memory transfer.

Secondly, the approach where the material part of the fourth-order spatial tangent stiffness tensor is cached ("tensor4") is slower than the "tensor2" approach, despite fewer arithmetic operations, on both systems. This shows that the memory access is the deciding metric for the "tensor4" strategy. Nonetheless, the "tensor4" strategy with $p \geq 2$ is considerably faster than matrix-based strategies: For $p = 4$ in 3D, the matrix-free variant is three times as fast as a sparse matrix-vector product on *linear elements* on Cascade Lake. This means we can apply this matrix-free variant to any finite-strain material model by caching the material part of the fourth-order spatial tangent stiffness tensor in addition to the Kirchhoff stress and expect this to be faster than the matrix-based implementation. It is therefore feasible to define highly performant generic operators that are independent of any applied constitutive laws, as long as the resulting tangent operator is expressed via the format in (19).

As outlined in the introduction, it is not only the performance of the matrix-free vector multiplication, but also the memory requirement to store a sparse matrix which is the driving force behind matrix-free methods. Figures 5 c, 5 f, 6 c and 6 f demonstrate that even with caching one fourth-order symmetric tensor and one second-order tensor at each quadrature point ("tensor4"), the matrix-free approach takes much less memory than its matrix-based counterpart. In those graphs, we report the overall memory needed to cache additional quantities required for the tangent operator according to Table 1 as well as all metadata required by the `deal.II` *MatrixFree* object.

Finally, we analyze the node-level performance of the two approaches using the roofline performance model[39]. The achieved memory throughput (MBytes/s) and the number of floating point operations executed per second (MFLOP/s) are measured using the `MEM_DP` group of the LIKWID[40] tool, version 4.2.1. The measurements are done on a single socket of the Cascade

(a) throughput matrix-vector product (2D)

(b) time matrix-vector product (2D)

(c) memory access (2D)

(d) throughput matrix-vector product (3D)

(e) time matrix-vector product (3D)
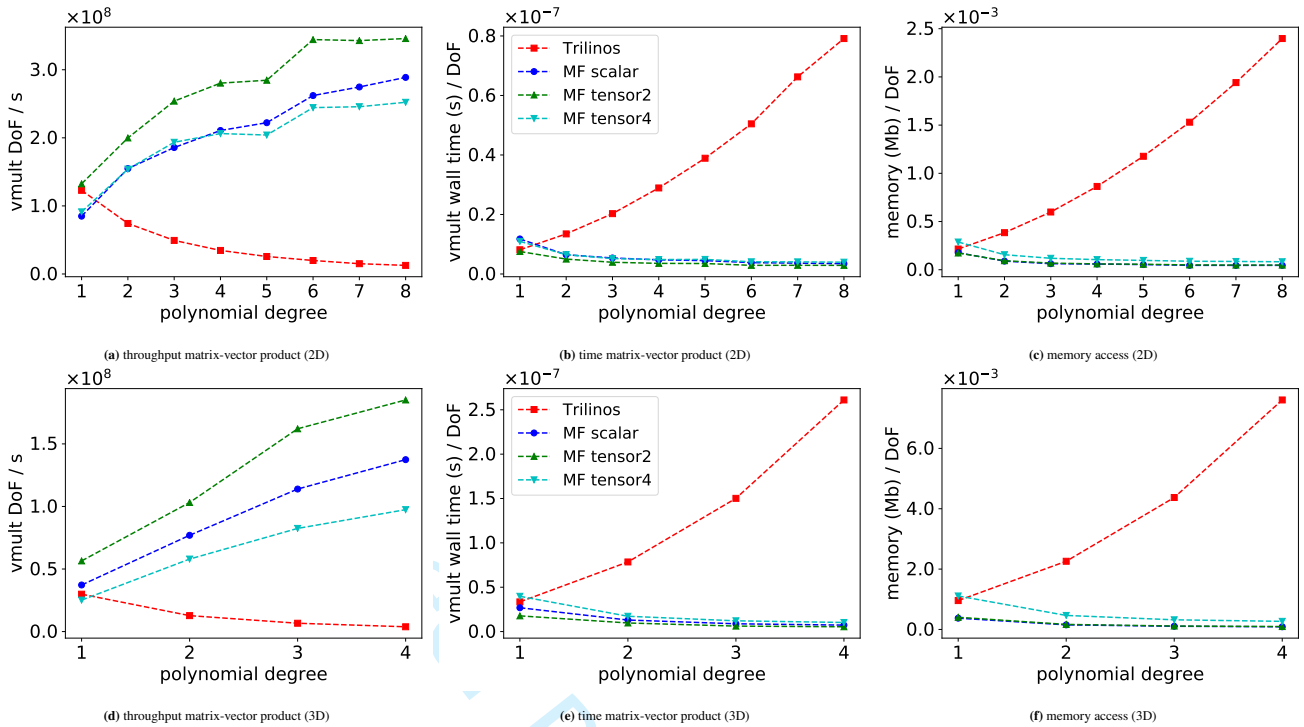
(f) memory access (3D)

FIGURE 6 Westmere cluster, 20 cores, matrix-vector multiplication

Lake system by pinning all 20 MPI processes to this socket, whose arithmetic peak performance at 2.0 GHz is 1280 GFlop/s and measured memory bandwidth for pure read tasks is 120 GB/s as well as 90 GB/s for mixed data access of two loads per one store.

The results of the roofline analysis in Figure 7 show that the matrix-based approach is fully memory-bandwidth bound with a very low arithmetic intensity of 0.16 and a performance of 17.36 GFlop/s (1.4% of the arithmetic peak) averaged over the 2D results. The matrix-free implementations of the finite-strain elastic tangent operator (on average) lead to a much better performance of 86.9 GFlop/s in 2D, as well as about an order of magnitude higher computational intensity, but still stays in the memory-bound regime. The achieved performance is about 6.8% of the peak arithmetic performance. This result is similar to the deformed mesh results of [10, Figure 18].

The roofline model also explains the difference in performance between the "tensor4" caching strategy (Algorithm 3) and the "tensor2" caching strategy (Algorithm 2): Both algorithms essentially run at the memory limit of around 100 GB/s, but the former involves a considerably higher memory transfer according to Table 1. The "scalar" caching strategy (Algorithm 1) demonstrates the highest computational intensity, but remains close to the memory bandwidth limit with more than 80 GB/s except for the $p = 2$ case. The higher arithmetic load is not surprising as at each quadrature point we need to re-evaluate the Kirchhoff stress τ. Given that in this case we need to evaluate gradients with respect to both the referential and the current configuration as listed in Table 1, no advantage of the scalar caching can be deduced despite the higher arithmetic performance.

In order to better understand the behavior of the matrix-free implementations, we exemplarily collect a breakdown of the operator evaluation into various stages inside the element loop of Algorithm 3: (i) 'read / write' denotes reading and writing from/to a global vector (lines 4 and 14) with indirect addressing of gather/scatter type; (ii) 'sum factorization' denotes application of sum factorization techniques (lines 6 and 13); (iii) 'quadrature loop' denotes operations performed at each quadrature (lines 7 – 12); (iv) 'zero vector' denotes setting destination vector to zero (line 1); (v) 'MPI' denotes MPI communication (lines 2 and 16). Given that the measurements are within the cell loop and there is an extensive overlap between the memory transfer (due to hardware prefetching) and arithmetic operations, the data of Figure 8 is derived from measuring the cost of 'sum factorization' by comparing the timings of part (i) and (ii) together against part (i) only. From Figure 8 a and 8 b, it becomes clear that sum factorization is cheap enough to be almost completely hidden behind the memory transfer. The majority of time is spent within the quadrature loop for all variants, with an increasing proportion for increasing degrees. Note also that higher degrees achieve a higher throughput in terms of DoF/s in Figures 5 and 6, as the memory access of quadrature point data scales as $(p+1)^d/p^d$ compared to the number of unknowns.
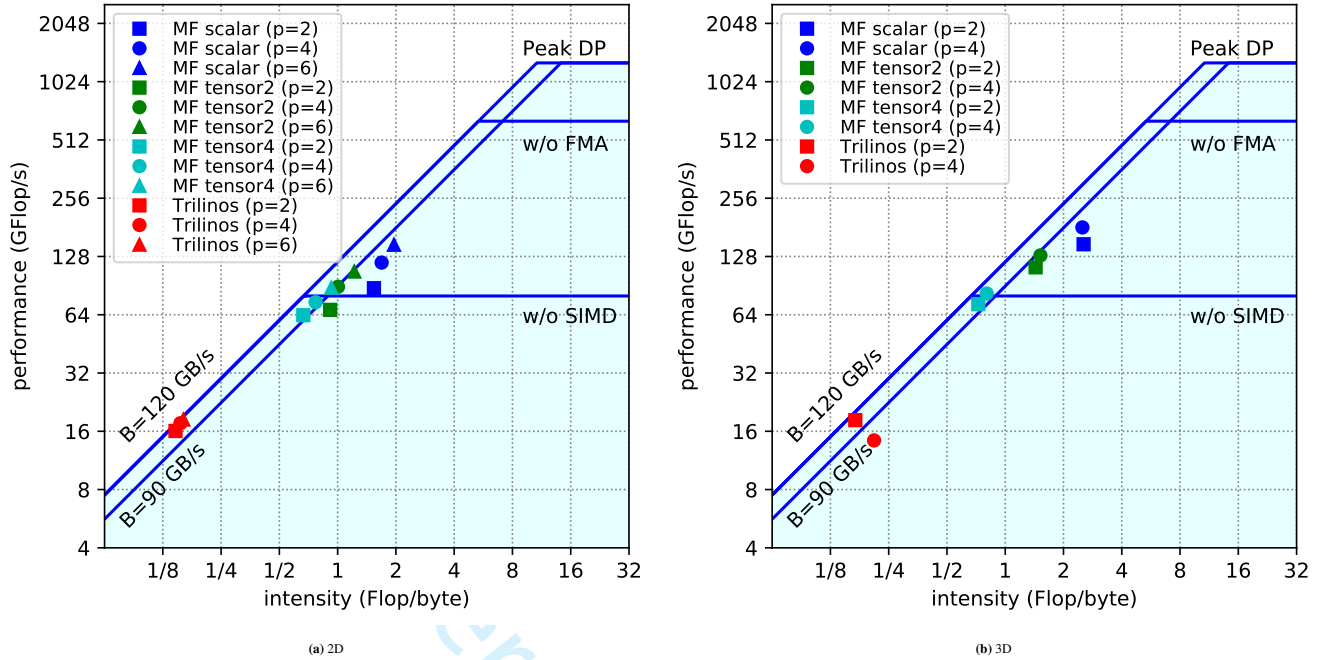
(a) 2D

(b) 3D

FIGURE 7 Roofline model for a selection of polynomial degrees on Cascade Lake. 'Peak DP' denotes the peak double precision performance (P), 'w/o FMA' represents a ceiling without Fused Multiply-Add ($P/2$), finally 'w/o SIMD' is a ceiling where both FMA and SIMD vectorization are discarded ($P/16$).
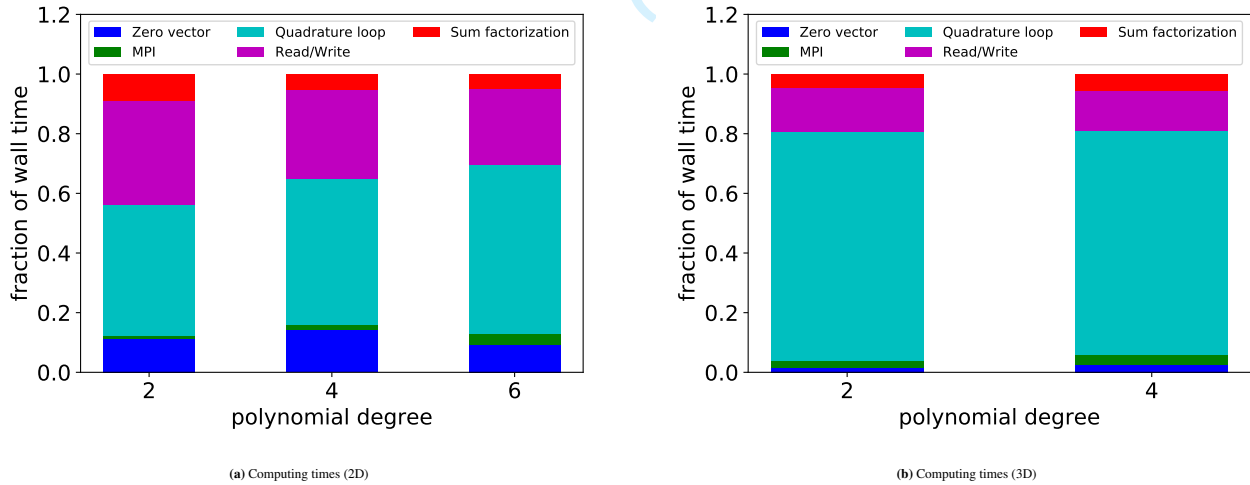

(a) Computing times (2D)

(b) Computing times (3D)

FIGURE 8 Breakdown of computing times and roofline analysis of various steps of Algorithm 3.

Table 3 and Table 4 show the speed-up obtained due to explicit SIMD vectorization intrinsics and due to the MPI parallelization. When run in serial, explicit SIMD vectorization via intrinsics provides a speedup of up to 3× over the auto-vectorization case. This is less than the theoretical factor of 8, but in line with previous experiments presented in Figure 6 of[41] with a speedup of around 1.5−2.5× with 8-wide vectorization. The reason for sub-optimal speedup is mostly related to the memory access in the gather/scatter operations as well as the limited memory bandwidth. When going to the parallel case with 20 cores, the less-than-optimal speedup can be attributed to two different effects. For the auto-vectorized case, the different (turbo) clock frequencies

TABLE 3 Wall-clock time in seconds and performance in GFlops of Algorithms 2 and 3 in 2D for various combinations of polynomial degrees, vectorization and parallelization on Cascade Lake.

	auto-vect serial 3.8 GHz		auto-vect MPI 20 cores 2.4 GHz			SIMD 3.7 GHz			SIMD+MPI 20 cores 2.0 GHz		
p	time	GFlop/s	time	GFlop/s	speedup	time	GFlop/s	speedup	time	GFlop/s	speedup
					Algorithm 2						
2	0.68	2.18	$4.81 \cdot 10^{-2}$	31.27	14.03	0.22	6.62	3.09	$2.19 \cdot 10^{-2}$	67.82	30.77
4	0.36	3.61	$2.99 \cdot 10^{-2}$	43.38	11.95	0.12	11.01	3.09	$1.44 \cdot 10^{-2}$	89.25	24.86
6	0.18	4.4	$1.49 \cdot 10^{-2}$	52.74	11.9	$5.93 \cdot 10^{-2}$	13	2.99	$7.32 \cdot 10^{-3}$	106.97	24.21
					Algorithm 3						
2	0.63	2.62	$4.58 \cdot 10^{-2}$	36.15	13.65	0.23	6.93	2.73	$2.53 \cdot 10^{-2}$	63.66	24.7
4	0.37	3.85	$3.11 \cdot 10^{-2}$	45.71	11.74	0.14	9.57	2.55	$1.86 \cdot 10^{-2}$	74.28	19.63
6	0.18	4.56	$1.57 \cdot 10^{-2}$	54.1	11.69	$7.39 \cdot 10^{-2}$	11.1	2.49	$9.46 \cdot 10^{-3}$	88.23	19.42

TABLE 4 Wall-clock time in seconds and performance in GFlops of Algorithms 2 and 3 in 3D for various combinations of polynomial degrees, vectorization and parallelization on Cascade Lake.

	auto-vect serial 3.8 GHz		auto-vect MPI 20 cores 2.4 GHz			SIMD 3.7 GHz			SIMD+MPI 20 cores 2.0 GHz		
p	time	GFlop/s	time	GFlop/s	speedup	time	GFlop/s	speedup	time	GFlop/s	speedup
					Algorithm 2						
2	0.33	4.34	$2.81 \cdot 10^{-2}$	51.75	11.75	0.12	11.72	2.7	$1.32 \cdot 10^{-2}$	112.47	25.11
4	0.19	5.46	$1.59 \cdot 10^{-2}$	64.77	11.76	$6.95 \cdot 10^{-2}$	14.71	2.69	$8.03 \cdot 10^{-3}$	129.66	23.27
					Algorithm 3						
2	0.38	4.5	$3.47 \cdot 10^{-2}$	51.9	11.06	0.22	7.74	1.72	$2.43 \cdot 10^{-2}$	72.56	15.81
4	0.22	5.39	$1.9 \cdot 10^{-2}$	63.11	11.62	0.13	9.33	1.73	$1.46 \cdot 10^{-2}$	82.39	15.06

permit a speedup of a factor 12.6 at most, close to our results[7]. For the SIMD+MPI case, the main reason is the memory bandwidth: As can be seen from Figures 7 a and 7 b, both algorithms are eventually memory-bandwidth limited. Since Algorithm 3 has a lower arithmetic intensity, the speedup over the serial, auto-vectorized case is less than that of Algorithm 2.

To summarize this section, we can attribute the speed-up of the matrix-free finite-strain tangent operators for higher-order elements to two factors. The first one is the higher algorithmic intensity and lower memory access as compared to the matrix-based implementation, see Figure 7 . Since both variants are in the memory-limited regime, the matrix-free implementation with its lower memory access is beneficial. The second benefit is the reduction in the number of floating point operations per DoF for higher order elements thanks to the sum factorization technique, as discussed in Section 4 and Section 2.4 of[2]. Among the three suggested implementations, the Algorithm 3 is the most flexible approach, whereas Algorithm 2 is faster due to a higher arithmetic intensity.

6.2 | Preconditioned iterative solver

Next, we evaluate the performance of the proposed geometric multigrid preconditioner. Our main goal is to examine whether or not the adopted level operators lead to an efficient preconditioner from the linear algebra perspective. To that end, we consider the average number of CG iterations throughout the entire simulation, i.e., averaged over each Newton–Raphson iteration and each loading step. Figures 9 a, 9 b, 10 a and 10 b show a noticeable increase in the average number of CG iterations for increasing polynomial degrees for the GMG preconditioner. The black-box AMG preconditioner demonstrates the same trend in terms of the number of CG iterations, but requires many more iterations for convergence. Consequently, we can conclude that (i) the selected smoother is suitable for the present finite strain elasticity problem, and that (ii) although the adopted level

---

[7]Note that we intentionally run with Turbo mode on to permit the processor to come close to the power limit for all settings and present more realistic performance limits.
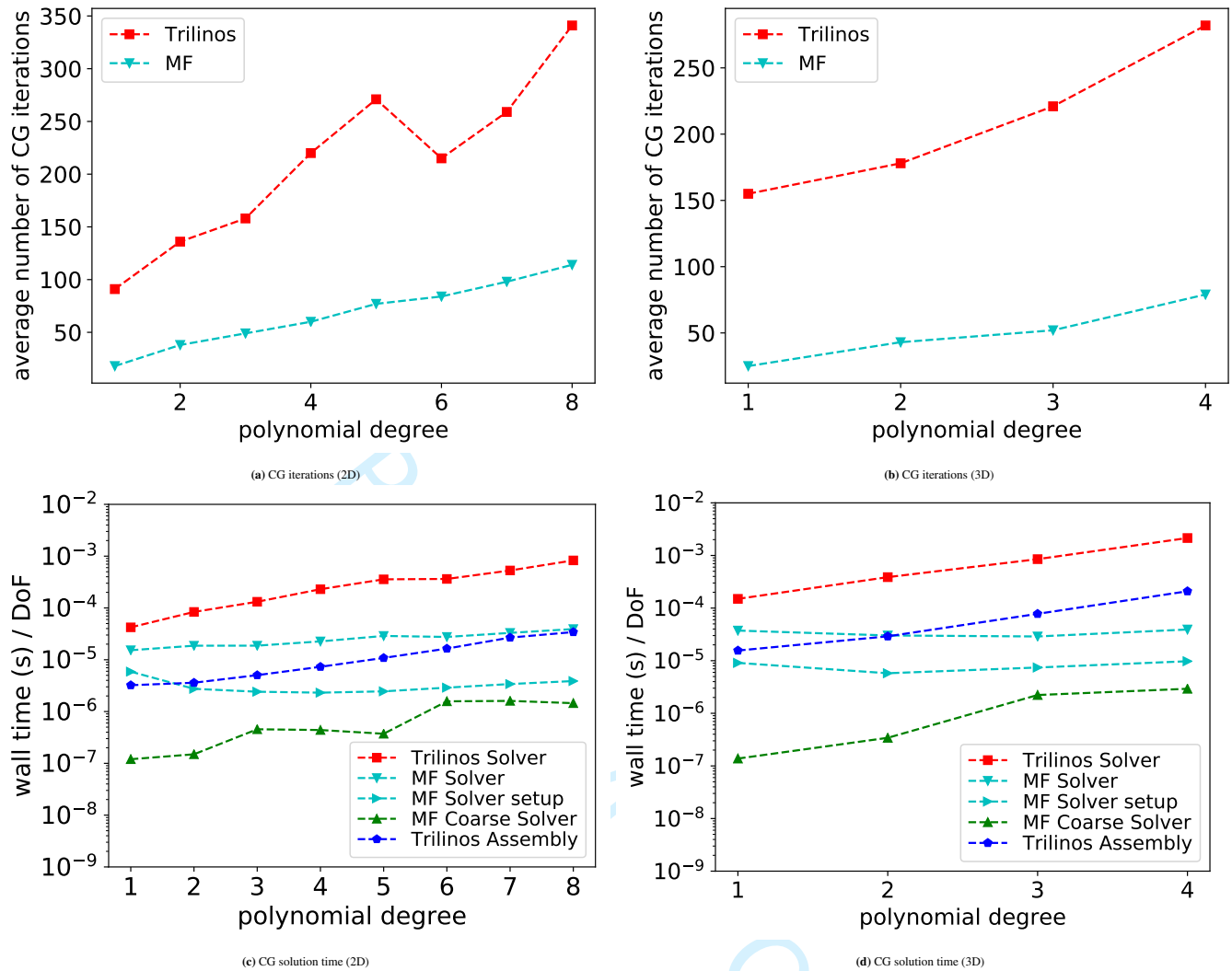
(a) CG iterations (2D)

(b) CG iterations (3D)

(c) CG solution time (2D)

(d) CG solution time (3D)

FIGURE 9 Cascade Lake, one socket with 20 cores, iterative solver.

operators are not built using the triple product $\boldsymbol{A}^{l+1} = \boldsymbol{I}_l^{l+1} \boldsymbol{A}^l \boldsymbol{I}_{l+1}^l$, they still result in a multigrid preconditioner which requires fewer CG iterations as compared to the algebraic multigrid baseline.

Figures 9 and 10 also report the solution times of the preconditioned iterative solvers. Our goal is to compare the matrix-free solver with a GMG preconditioner against a matrix-based approach with a well-established implementation of an AMG preconditioner. We note specifically that the comparison is restricted to the choice of level operators used with the pre-existing GMG framework, and not the framework itself. Figures 9 c, 9 d, 10 c and 10 d confirm that the efficiency of the proposed GMG preconditioner translates into faster solution times for all polynomial degrees. Most importantly, the wall time per DoF for the matrix-free solver with GMG preconditioner stays almost constant both for 2D and 3D problems. In comparison, the wall time of the matrix-based iterative solver grows rapidly with increasing polynomial degree. Clearly this is related both to the performance of the preconditioner from the linear algebra perspective as well as the matrix-vector products, studied in the previous section. Based on these results, we conclude that the matrix-free approach represents a very competitive solution strategy for engineering problems with compressible hyperelastic finite strain material models.

An extension of this study from the node level to the cluster level is beyond the scope of this work. However, we have performed a preliminary study of the weak scalability of the implementation. Table 5 demonstrate a good weak scaling up to 320 cores, i.e., 8 nodes of dual-socket Cascade Lake, of the wall time per matrix-vector product and per iteration in the CG solver for the 2D problem with $p = 4$ and the 3D problem with a tri-quadratic basis. This confirms that the matrix-free and multigrid frameworks
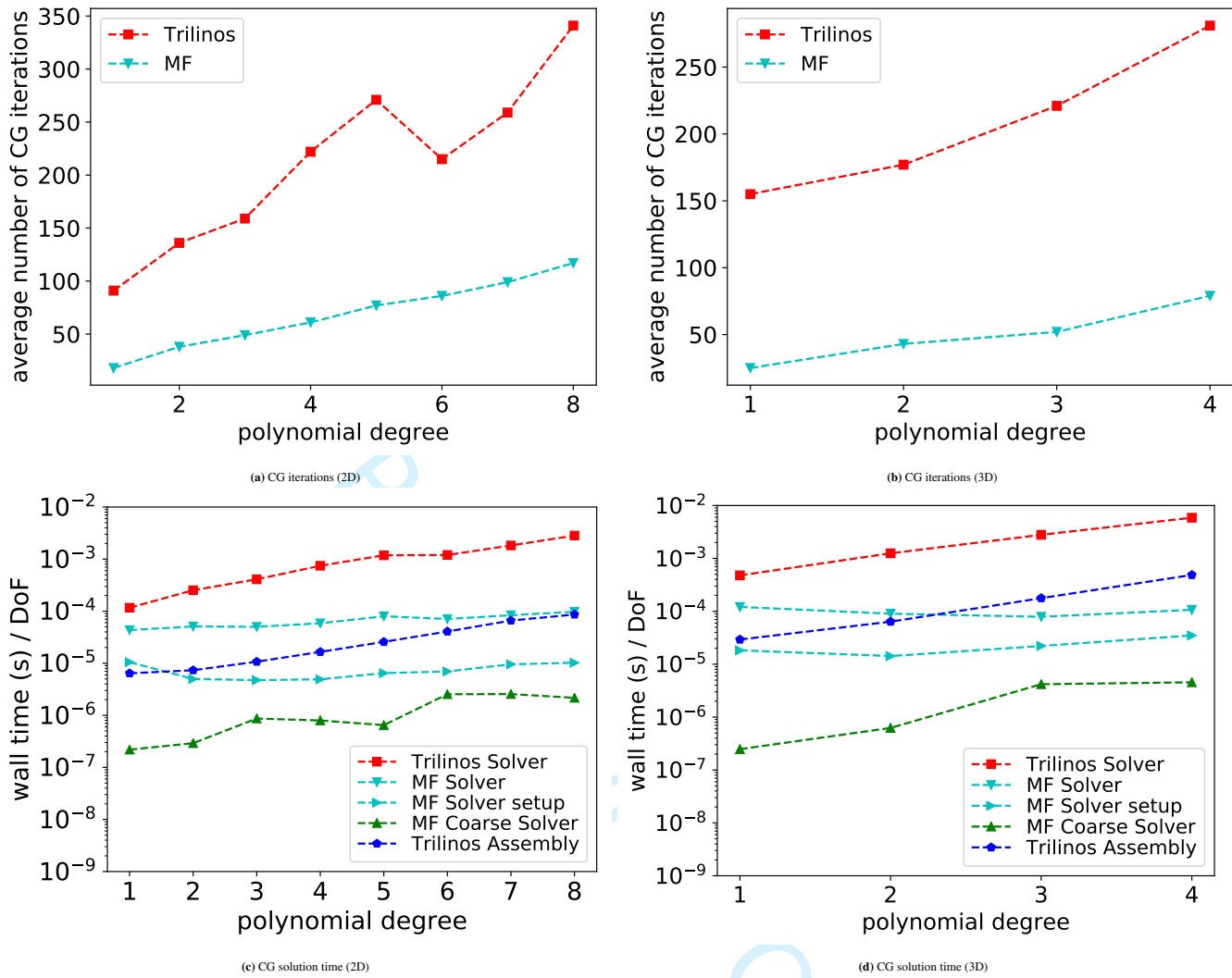
(a) CG iterations (2D)

(b) CG iterations (3D)

(c) CG solution time (2D)

(d) CG solution time (3D)

FIGURE 10 Westmere cluster, 20 cores.

implemented in the `deal.II` library can provide a competitive solution for cluster-sized problems as well, also compare [4, Figure 7] for scaling results for up to 147,456 CPU cores and 34.4 billion degrees of freedom for an incompressible flow problem.

TABLE 5 Weak scaling of Algorithm 2.

(a) 2D and quartic polynomial basis

cores	N_{DoF}	N_{gref}	vmult [s]	CG [s / iteration]
20	2,887,680	5	$3.39 \cdot 10^{-3}$	$4.56 \cdot 10^{-2}$
80	11,542,528	6	$3.54 \cdot 10^{-3}$	$4.78 \cdot 10^{-2}$
320	46,153,728	7	$3.79 \cdot 10^{-3}$	$5.2 \cdot 10^{-2}$

(b) 3D and quadratic polynomial basis

cores	N_{DoF}	N_{gref}	vmult [s]	CG [s / iteration]
40	4,442,880	3	$7.07 \cdot 10^{-3}$	$6.43 \cdot 10^{-2}$
320	35,071,488	4	$7.53 \cdot 10^{-3}$	$7.14 \cdot 10^{-2}$

7 | SUMMARY AND CONCLUSIONS

In this contribution, we proposed and numerically investigated three different matrix-free implementations of tangent operators for finite-strain elasticity with heterogeneous materials. To the best of our knowledge, this is the first work that evaluates matrix-free sum factorization techniques on the partial differential equations arising in this case. Among the three examined algorithms, the implementation that caches the second-order Kirchhoff stress tensor and evaluates terms in the current configuration (Algorithm 2) is faster than caching only a scalar at quadrature points (Algorithm 1) or caching the material part of the fourth-order spatial tangent stiffness tensor together with the second-order Kirchhoff stress (Algorithm 3). Algorithm 2 is recommended when linearization of the adopted material model allows to efficiently implement the action of the material part of the fourth-order spatial tangent stiffness tensor on a second-order symmetric tensor by two scalar quantities. Algorithm 3 does not utilize the specific form of the material part of the stiffness tensor. In this case the matrix-free implementation of the tangent operator is somewhat slower but more flexible because it can be applied to any constitutive model which operates with the material part of the fourth-order spatial tangent stiffness tensor at the quadrature points. This implementation is still considerably faster than matrix-based strategies for polynomial degrees higher than one.

The roofline model indicates that the matrix-free finite-strain elasticity tangent operators have an order magnitude higher algorithmic intensity. Despite the higher arithmetic intensity, the matrix-free implementations are still memory bandwidth-limited and in particular by the data required at quadrature points in terms of fourth-order and second-order symmetric tensors.

We also propose a method by which to construct level tangent operators and employ them to define a geometric multigrid preconditioner with standard geometric transfer operations between each level. The GMG has been applied to a heterogeneous material assuming that the coarsest level can provide an adequate discretization of the heterogeneity. The numerical studies indicate that the proposed preconditioner leads to much fewer iterations of the iterative solver as compared to the algebraic multigrid preconditioner. The multigrid matrix-free preconditioner also leads to a solution approach that is faster than the matrix-based AMG for all polynomial degrees studied here. Most importantly, the wall time per degree of freedom for solving the problem is close to being constant for polynomial degrees between one and four. On the other hand, the matrix-based implementation with AMG becomes prohibitively expensive for higher-order bases. We conclude that the matrix-free implementations of tangent operators for finite-strain elasticity together with the geometric multigrid preconditioner is a very competitive solution strategy, as compared to traditional matrix-based approach. Our future work will be focused on adopting the proposed matrix-free multigrid solution approach to FE^2 homogenization as well studying its behavior on the cluster level.

How to cite this article: D. Davydov, J.-P. Pelteret, D. Arndt, M. Kronbichler and P. Steinmann (2019), A matrix-free approach for finite-strain hyperelastic problems using geometric multigrid, *International Journal for Numerical Methods in Engineering*, *2019;00:1–23*.

APPENDIX

It is a common practice to write the bilinear form completely in the referential configuration. For the constitutive equation (11) considered in this study, this results in the following bilinear form

$$A_{ij} \equiv a(\boldsymbol{N}_i, \boldsymbol{N}_j) = \int_{\mathcal{B}_0} \mathrm{grad}\, \boldsymbol{N}_i \,:\, \mathrm{D}_{\boldsymbol{F}}\boldsymbol{P} \,:\, \mathrm{grad}\, \boldsymbol{N}_j \mathrm{d}V \tag{1}$$

$$\mathrm{D}_{\boldsymbol{F}}\boldsymbol{P} := 2\lambda \boldsymbol{F}^{-T} \otimes \boldsymbol{F}^{-T} + [\mu - 2\lambda \ln(J)]\, \boldsymbol{F}^{-T} \underline{\otimes} \boldsymbol{F}^{-1} + \mu \boldsymbol{I} \overline{\otimes} \boldsymbol{I} \tag{2}$$

where the upper and lower dyadic products of pairs of second-order tensors are respectively given by

$$\boldsymbol{A}\overline{\otimes}\boldsymbol{B} = A_{ik} B_{jl} \boldsymbol{e}_i \otimes \boldsymbol{e}_j \otimes \boldsymbol{e}_k \otimes \boldsymbol{e}_l$$
$$\boldsymbol{A}\underline{\otimes}\boldsymbol{B} = A_{il} B_{jk} \boldsymbol{e}_i \otimes \boldsymbol{e}_j \otimes \boldsymbol{e}_k \otimes \boldsymbol{e}_l \,.$$

This formulation can also be straight forwardly adopted within the matrix-free framework of the `deal.II` library. To that end a variation of the Algorithm 3 can be used where the gradients are evaluated with respect to the referential configuration and only a fourth-order tangent tensor $\mathrm{D}_{\boldsymbol{F}}\boldsymbol{P}$ is cached per quadrature point.

In the interest of brevity, no results are reported for this approach, however we do support it in our code. As compared to Algorithm 3, we need to cache more data, which results in both higher memory requirement per DoF as well as a lower arithmetic intensity of the operator evaluation. On both architectures we recorded a slightly larger wall time per DoF as compared to the Algorithm 3, however still much faster than the matrix-based approach. From the LIKWID measurements we observed that this algorithm also results in lower performance on Cascade Lake. The advantage of this algorithm is that it does not require evaluation of gradients in the current configuration, which leads to a slightly lower setup time as compared to the other three algorithms that rely on `MappingQEulerian` class of the `deal.II` library. Overall we consider this approach to be a valid alternative to the Algorithm 3 that can be be applied to any constitutive model which operates with the fourth-order referential tangent stiffness tensor on the quadrature point level.

A scalar variant of the algorithm in referential coordinates, related to Algorithm 1, can also be devised. In this setting, all quantities are evaluated in referential coordinates, necessitating only a single set of metric terms according to Table 1 . For the derivatives of the current solution an additional multiplication by $\boldsymbol{F}^{-T}$ for both the solution as well as the test function is necessary, with $\boldsymbol{F}^{-T}$ evaluated on the fly after line 10 in Algorithm 1. Despite the additional evaluations, this leads to a faster calculation than Algorithm 1 on Cascade Lake, running with 162 GFlop/s and 68 GB/s for $p = 6$ in 2D or 225 GFlop/s and 59 GB/s for $p = 4$ in 3D. However, execution is still slower than Algorithm 2, despite the lower memory access. This is because the bottleneck is now mostly within the core, although below the arithmetic throughput limit. This is mainly caused by the cost of indirect addressing in the vector access as well as division for inverting $\boldsymbol{F}$ on the fly.

References

1. Brown Jed. Efficient nonlinear solvers for nodal high-order finite elements in 3D. *Journal of Scientific Computing.* 2010;45(1-3):48–63.

2. Kronbichler Martin, Kormann Katharina. A generic interface for parallel cell-based finite element operator application. *Computers & Fluids.* 2012;63:135–147.

3. May Dave A., Brown Jed, Le Pourhiet Laetitia. A scalable, matrix-free multigrid preconditioner for finite element discretizations of heterogeneous Stokes flow. *Computer methods in applied mechanics and engineering.* 2015;290:496–523.

4. Krank Benjamin, Fehn Niklas, Wall Wolfgang A., Kronbichler Martin. A high-order semi-explicit discontinuous Galerkin solver for 3D incompressible flow with application to DNS and LES of turbulent channel flow. *Journal of Computational Physics.* 2017;348:634–659.

5. Gmeiner Björn, Huber Markus, John Lorenz, Rüde Ulrich, Wohlmuth Barbara. A quantitative performance study for Stokes solvers at the extreme scale. *Journal of Computational Science.* 2016;17:509–521.

6. Abdelfattah Ahmad, Baboulin Marc, Dobrev Veselin, et al. High-performance tensor contractions for GPUs. *Procedia Computer Science*. 2016;80:108–118.

7. Kronbichler Martin, Ljungkvist Karl. Multigrid for matrix-free high-order finite element computations on graphics processors. *ACM Transactions on Parallel Computing*. 2019;6(1):2:1–2:32.

8. Bauer Simon, Drzisga Daniel, Mohr Marcus, Rüde Ulrich, Waluga Christian, Wohlmuth Barbara. A Stencil Scaling Approach for Accelerating Matrix-Free Finite Element Implementations. *SIAM J. Sci. Comput.*. 2018;40(6):C748–C778.

9. Cantwell Chris D., Sherwin Spencer J., Kirby Robert M., Kelly Paul H. J.. From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements. *Comput. Fluids*. 2011;43:23–28.

10. Kronbichler Martin, Kormann Katharina. Fast matrix-free evaluation of discontinuous Galerkin finite element operators. *ACM Transactions on Mathematical Software*. 2019;45(3):29:1–29:40.

11. Müthing Steffen, Piatkowski Marian, Bastian Peter. High-performance Implementation of Matrix-free High-order Discontinuous Galerkin Methods. *arXiv preprint arXiv:1711.10885*. 2017;.

12. Fischer Paul, Rathnayake Thilina, Dutta Som, et al. Running Faster in HPC Applications. *submitted*. 2019;. http://ceed.exascaleproject.org.

13. Arndt Daniel, Bangerth Wolfgang, Clevenger Thomas C., et al. The deal.II Library, Version 9.1. *Journal of Numerical Mathematics*. 2019;.

14. Wriggers Peter. *Nonlinear finite element methods*. Springer Science & Business Media; 2008.

15. Treloar Leslie R. G.. *The Physics of Rubber Elasticity*. USA: Oxford University Press; 1975.

16. Treloar Leslie R. G., Hopkins H. Geoffrey, Rivlin Ronald S., et al. The Mechanics of Rubber Elasticity. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*. 1976;351(1666):301–330.

17. Vos Peter E. J., Sherwin Spencer J., Kirby Robert M.. From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low-and high-order discretisations. *Journal of Computational Physics*. 2010;229(13):5161–5181.

18. Bartlett Roscoe, Demeshko Irina, Gamblin Todd, et al. xSDK Foundations: Toward an Extreme-scale Scientific Software Development Kit. *Supercomputing Frontiers and Innovations*. 2017;4(1).

19. Burstedde Carsten, Wilcox Lucas C., Ghattas Omar. p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*. 2011;33(3):1103-1133.

20. Heroux Michael A., Bartlett Roscoe A., Howle Vicki E., et al. An overview of the Trilinos project. *ACM Trans. Math. Softw.*. 2005;31(3):397–423.

21. Heroux Michael A.. *Epetra performance optimization guide*. SAND2005-1668: Sandia National Laboratories; 2005.

22. Gee Michael W., Siefert Christopher M., Hu Jonathan J., Tuminaro Ray S., Sala Marzio G.. *ML 5.0 Smoothed Aggregation User's Guide*. SAND2006-2649: Sandia National Laboratories; 2006.

23. Bramble James H., Pasciak Joseph E., Xu Jinchao. Parallel multilevel preconditioners. *Mathematics of Computation*. 1990;55(191):1–22.

24. Briggs William L., Henson Van Emden, McCormick Steve F.. *A Multigrid Tutorial: Second Edition*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics; 2000.

25. Janssen Bärbel, Kanschat Guido. Adaptive Multilevel Methods with Local Smoothing for H^1- and H^{curl}-Conforming High Order Finite Element Methods. *SIAM Journal on Scientific Computing*. 2011;33(4):2095–2114.

26. Hackbusch Wolfgang. *Multi-grid methods and applications*. Springer Series in Computational MathematicsSpringer-Verlag Berlin Heidelberg; 1985.

22 |

27. Wesseling Pieter. *An introduction to multigrid methods*. Pure and applied mathematicsJohn Wiley & Sons; 1992.

28. Suquet Pierre. Elements of homogenization for inelastic solid mechanics. In: Sanchez-Palencia Enrique, Zaoui Andre, eds. *Homogenization techniques for composite media*, Lecture Notes in Physics, vol. 272: Springer-Verlag Berlin Heidelberg 1987 (pp. 193–278).

29. Hill Rodney. On constitutive macro-variables for heterogeneous solids at finite strain. *Proc. R. Soc. Lond. A*. 1972;326(1565):131–147.

30. Hashin Zvi. Analysis of composite materials–a survey. *Journal of Applied Mechanics*. 1983;50(3):481–505.

31. Castaneda Pedro Ponte, Suquet Pierre. Nonlinear composites. In: Advances in Applied Mechanics, vol. 34: Elsevier 1997 (pp. 171–302).

32. Miehe Christian, Bayreuther Claus G.. On multiscale FE analyses of heterogeneous structures: from homogenization to multigrid solvers. *International Journal for Numerical Methods in Engineering*. 2007;71(10):1135–1180.

33. Kaczmarczyk Lukasz, Pearce Chris J., Bicanic Nenand, Souza Neto Eduardo. Numerical multiscale solution strategy for fracturing heterogeneous materials. *Computer Methods in Applied Mechanics and Engineering*. 2010;199(17-20):1100–1113.

34. Clevenger Thomas C., Heister Timo, Kanschat Guido, Kronbichler Martin. A Flexible, Parallel, Adaptive Geometric Multigrid method for FEM. *CoRR*. 2019;abs/1904.03317.

35. Rønquist Einar M., Patera Anthony T.. Spectral element multigrid. I. Formulation and numerical results. *J. Sci. Comput.*. 1987;2(4):389–406.

36. Gholami Amir, Malhotra Dhairya, Sundar Hari, Biros George. FFT, FMM, or Multigrid? A comparative Study of State-Of-the-Art Poisson Solvers for Uniform and Nonuniform Grids in the Unit Cube. *SIAM J. Sci. Comput.*. 2016;38(3):C280–C306.

37. Ibeid Huda, Olson Luke, Gropp William. FFT, FMM, and Multigrid on the Road to Exascale: performance challenges and opportunities. *arXiv preprint arXiv:1810.11883*. 2018;.

38. Varga Richard S.. *Matrix iterative analysis*. Berlin: Springer; 2nd ed.2009.

39. Williams Samuel, Waterman Andrew, Patterson David. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*. 2009;52(4):65–76.

40. Treibig Jan, Hager Georg, Wellein Gerhard. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In: ; 2010; San Diego CA.

41. Kronbichler Martin, Kormann Katharina, Pasichnyk Igor, Allalen Momme. Fast matrix-free discontinuous Galerkin kernels on modern computer architectures. In: Kunkel J. M., Yokota R., Balaji P., Keyes D. E., eds. *International Supercomputing Conference 2017, LNCS 10266*, :237–255; 2017.