A generic interface for parallel cell-based finite element operator application

Martin Kronbichler* Katharina Kormann*

Abstract

We present a memory-efficient and parallel framework for finite element operator application implemented in the generic open-source library `deal.II`. Instead of assembling a sparse matrix and using it for matrix-vector products, the operation is applied by cell-wise quadrature. The evaluation of shape functions is implemented with a sum-factorization approach. Our implementation is parallelized on three levels to exploit modern supercomputer architecture in an optimal way: MPI over remote nodes, thread parallelization with dynamic task scheduling within the nodes, and explicit vectorization for utilizing processors' vector units. Special data structures are designed for high performance and to keep the memory requirements to a minimum. The framework handles adaptively refined meshes and systems of partial differential equations. We provide performance tests for both linear and nonlinear PDEs which show that our cell-based implementation is faster than sparse matrix-vector products for polynomial order two and higher on hexahedral elements and yields ten times higher Gflops rates.

Key words. Finite/spectral element method, matrix-free method, sum-factorization, hybrid parallelization

1 Introduction

Finite element problems are often solved by evaluating discrete differential operators like in iterative linear solvers [52] or time stepping schemes. This makes the operator evaluation the central and usually most time-consuming component in finite element codes. In linear problems, these operations are matrix-vector products. Nonlinear problems are often tackled by standard assembly of residuals and linearization to obtain coefficient matrices (re-assembled from one iteration to the next). Many generic finite element libraries like `deal.II` [5, 6], DiffPack [17, 39], DUNE [10, 11], FEniCS/Dolfin [40, 41], Getfem++ [50], libMesh [35], OOFEM [48], or PLTMG [8] separate finite element computations from linear algebra and rely on sparse matrix-vector (SpMV) kernels in iterative solvers, either by direct implementation or interfaces through linear algebra packages like PETSc [2, 3] or Trilinos [29, 30]. In this article, we want to challenge the view of strictly separating linear algebra from finite element assembly routines.

*Division of Scientific Computing, Department of Information Technology, Uppsala University, Box 337, 751 05 Uppsala, Sweden (martin.kronbichler@it.uu.se, katharina.kormann@it.uu.se).

We present a framework that exploits the special structure of the finite element operation as the differential operator is applied. Instead of assembling a global sparse matrix, we only store the unit cell shape function information, the enumeration of degrees of freedom, and the transformation from unit to real cell. For most configurations, this approach reduces the storage requirements considerably, at a low increase or even reduction of arithmetic operations compared to sparse matrices. A reduced memory requirement of the operator representation promises improved wall times through higher Gflop rates (billion arithmetic operations per second) because SpMVs are usually limited by memory bandwidth rather than arithmetic throughput [47, Chapt. 7]. Even though attempts have been made to tune SpMV kernels [56, 57] or to allow only problems with structured non-zero pattern [55], Gflops rates rarely exceed 2%–20% of peak arithmetic throughput.

Besides poor performance characteristics of traditional matrix-vector products, the number of nonzero elements per row in the matrix for a $(p-1)$th order finite element in d dimensions is proportional to p^d, rendering high order methods increasingly expensive. If we split the application of the FE operator into a function evaluation and integration step described by unit cell shape functions and derivatives, the shape information can be applied for one dimension at a time for basis functions that are derived from a tensor product. This is the case for quadrilateral and hexahedral elements we consider in this article, even though most of the ideas can also be applied to tetrahedral elements. This restructuring reduces the computational complexity to d^2p operations per degree of freedom and is usually referred to as sum-factorization in the literature, as proposed for matrix assembly in [43].

It is obvious that sum-factorization reduces the amount of computational work the larger d and p. Also, the effect of the aforementioned memory-efficiency becomes more relevant for higher dimensions and order. Hence, our implementation of a cell-based finite element operator in a generic code makes it possible to use high-order finite elements much more efficiently and provides a more flexible alternative to high-order finite differences [28] since adaptive mesh refinement and variable coefficients can be included straight-forwardly. Despite higher operation counts for low and medium order elements, we can report speedup compared to a sparse matrix vector product already for second order elements in both two and three dimensions, even when ignoring the additional cost of sparse matrix assembly that arises in practice. The reason for this is the improved arithmetic intensity, as we reach up to 70% of arithmetic peak performance. Moreover, our approach to selectively cache precomputed mapping information allows evaluating nonlinear finite element operators equally efficiently as linear ones. This is in contrast to standard codes where vector assembly is (much) more expensive than matrix-vector products.

The sum-factorization approach is a special form of a cell-based strategy. Early implementations of finite element methods often stored stiffness matrices as a collection of cell matrices and indices, a so-called element-by-element storage scheme, see e.g. [31, 22]. However, those approaches rather increase memory consumption and are therefore rarely efficient. Recently, cell-based strategies without explicit matrix storage have been analyzed for GPU programming [36, 37], and used in application-specific software like SPECFEM 3D [9]. However, these approaches have been limited to certain applications and do not target general finite element tools that provide a wider spectrum of functionality like adaptivity. Cantwell et al. [20] have com-

pared global sparse matrices, local matrices (an element-by-element approach), and sum-factorization for the Helmholtz problem in the special software Nektar++. However, their implementation uses BLAS routines for evaluating the components in sum-factorization, which implies function call overheads and relatively poor cache usage for low to moderate polynomial order. Instead, we choose to create specially adapted data types and kernels (that are tightly integrated into the broad computation facilities of `deal.II` for the ease of use). Also, the proposed implementation can handle constraints to enable adaptive mesh refinement with hanging nodes in a very efficient way. The implementation is tailored to exploit parallelism, including vectorization, shared and distributed memory computations on thousands of processors. The parallelization strategies build upon the special structure of cell-based operations.

The major contribution of this article is the discussion of an efficient implementation of the building blocks for finite element operator application (instead of assembly of matrices) and the consideration of memory consumption and data structures that are the key for high performance on current computer architectures. In particular, our implementation is apt for adaptively refined meshes and higher order methods on hexahedral elements. The outline of the article is as follows. In the next section, we present the cell-based approach. We also consider a special type of element based on the Gauss–Lobatto quadrature rule that have particularly nice properties for spectral elements. Sec. 3 presents our adapted data structures and Sec. 4 the parallelization strategies. In Sec. 5, performance tests for a sample problem are collected and discussed. Sec. 6 considers more advanced uses of the operator for linear and nonlinear problems. Finally, Sec. 7 concludes the article.

2 Cell-based implementation of FE operations

With our framework based on direct application of finite element operators, we can treat both linear and non-linear finite element problems. In this section, we demonstrate its idea.

Let us consider a finite element Galerkin approximation of a (possibly non-linear) operator A that takes a vector u as input and computes the integral of the operation over test functions ϕ_i, $i = 1,\ldots,n$, giving an output vector v. The operation can be rewritten as a sum of n_{cells} cell-based operations. This gives the general structure of the operation,

$$v = A(u) = \sum_{k=1}^{n_{\text{cells}}} C^T P_k^T A_k(P_k C u). \tag{1}$$

By P_k, we denote the matrix that defines the location of cell-related degrees of freedom in the global vector and C takes care of hanging node constraints that are necessary to maintain C^0 continuity on adaptively refined meshes (cf. Sec. 3.1 for details). Finally, A_k denotes the representation of operator A on cell k. For linear PDEs, the operation $A(u)$ corresponds to a matrix-vector product. In this case, $C^T \left(\sum_k P_k^T A_k P_k\right) C$ describes the construction of a matrix A by element assembly and final application of constraints, cf. [7].

If we take into account that the cells are partitioned among several MPI processes and not all degrees of freedom relevant to a given subdomain are owned by the respec-

tive processor (see Sec. 4.1), the steps for one operator application are as summarized in Algorithm 2.1.

Algorithm 2.1 *(Prototype finite element operator application)*

1. `import_ghost`*: Import vector values from other MPI processes that are needed for computations on locally owned cells associated with the present MPI process.*
2. *loop over locally owned cells (thread-parallelized on each MPI node)*
   (a) *Extract local vector values on cell:* $u_k = (P_k C)u$.
   (b) *Evaluate local operation* $v_k = A_k(u_k)$ *by quadrature approach.*
   (c) *Add the local contributions into the global result vector,* $v \leftarrow v + (P_k C)^T v_k$.
3. `compress`*: Exchange of information computed on locally owned cells for degrees of freedom owned by another MPI process.*

Step (2b) above can be implemented by explicitly forming and storing an array of the local matrices A_k for linear problems. For avoiding the storage of all matrix data, alternatives are to

- Compute $A_k(u)$ by quadrature on cell k through evaluating the FE function u^h and/or its derivatives on all quadrature points and testing by all test functions related to the cell.
- Compute matrix representation A_k and then $A_k u_k$. This is only efficient for linear operators and simple geometries that are described by constant Jacobian transformations, as used e.g. in FEniCS [40, 41].

Because of its efficiency and generality, we discuss the first variant.

2.1 Local quadrature approach

As a prototype operation, let us consider the differential operator

$$-\nabla \cdot K(\mathbf{x})\nabla(\cdot), \tag{2}$$

which corresponds to a Laplacian with variable anisotropic coefficient. $K(\mathbf{x})$ is a symmetric $d \times d$ matrix that couples the various components of the d-dimensional gradient. The corresponding FE operator evaluates the weak form

$$(\nabla\phi_j, K\nabla u^h)_\Omega, \quad j = 1,\dots,n, \tag{3}$$

where $u^h(\mathbf{x}) = \sum_{i=1}^n \phi_i(\mathbf{x})u^{(i)}$ denotes the global FE interpolation associated with the input vector u and $\{\phi_j,\, j = 1,\dots,n\}$ is the set of all basis functions in the weak form. Let us further denote by $\{\hat{\phi}_j(\hat{\mathbf{x}}),\, j = 1,\dots,p^d\}$ the unit cell basis functions and for a quadrature point $\mathbf{x}_q$ we denote by $\hat{\mathbf{x}}_q$ the corresponding point on the unit cell. Here $(p-1)$ is the degree of the finite element. As opposed to matrix approaches, we explicitly evaluate the gradient of the local FE interpolation $u_k^h(\mathbf{x}_q) = \sum_{i=1}^{p^d} \hat{\phi}_i(\hat{\mathbf{x}}_q)u_k^{(i)}$ on quadrature points $\mathbf{x}_q$,

$$\nabla u^h(\mathbf{x}_q) = \sum_{i=1}^{p^d} J_k^{-1}(\hat{\mathbf{x}}_q)\hat{\nabla}\hat{\phi}_i(\hat{\mathbf{x}}_q)u_k^{(i)} = J_k^{-1}(\hat{\mathbf{x}}_q)\sum_{i=1}^{p^d} \hat{\nabla}\hat{\phi}_i(\hat{\mathbf{x}}_q)u_k^{(i)}, \tag{4}$$

where ∇ denotes the gradient in real coordinates, $\hat{\nabla}$ the gradient on the unit cell, and $J_k^{-1}(\hat{\mathbf{x}}_q)$ is the inverse Jacobian of the transformation from the unit to the real cell. Note that our implementation evaluates unit-cell gradients first (summation), and applies the geometry in a second step. The computation of $\sum_i \hat{\nabla}\hat{\phi}_i(\hat{\mathbf{x}}_q)u_k^{(i)}$ on all quadrature points can be implemented efficiently using sum-factorization, see Sec. 2.2 below.

Each component i of the vector $v_k = A_k u_k$ corresponds to an integral, which is evaluated through quadrature:

$$\begin{aligned} v_k^{(i)} &= \sum_q \left(\nabla\phi_i(\mathbf{x}_q)\cdot K(\mathbf{x}_q)\nabla u^h(\mathbf{x}_q)\right) w_q|\det J_k(\hat{\mathbf{x}}_q)|, \\ &= \sum_q \left(J_k^{-T}(\hat{\mathbf{x}}_q)\hat{\nabla}\hat{\phi}_i(\hat{\mathbf{x}}_q)\cdot K(\mathbf{x}_q)\nabla u^h(\mathbf{x}_q)\right) w_q|\det J_k(\hat{\mathbf{x}}_q)|, \end{aligned} \tag{5}$$

where w_q denotes the quadrature weight and $|\det J_k(\hat{\mathbf{x}}_q)|$ is the Jacobian determinant. The multiplication by $J_k^{-T}(\hat{\mathbf{x}}_q)$ and $w_q|\det J_k(\hat{\mathbf{x}}_q)|$ is the same for each component $v_k^{(i)}$, so it can be factored out and applied before the summation over the quadrature points. The computation $v_k = A_k u_k$ is summarized in Algorithm 2.2.

Algorithm 2.2 *(Local computation $v_k = A_k u_k$ for Laplacian)*

(i) Compute gradients on unit cell for all quadrature points, i.e., sum in Eq. (4).

(ii) On each quadrature point:

- *Apply Jacobian transformation $J_k^{-1}(\hat{\mathbf{x}}_q)$ to get gradient in real space $\nabla u^h(\mathbf{x}_q)$ according to Eq.* (4).
- *Multiply by the variable coefficient matrix $K(\mathbf{x}_q)$.*
- *Multiply each component of $K(\mathbf{x}_q)\nabla u^h(\mathbf{x}_q)$ by $w_q|\det J_k(\hat{\mathbf{x}}_q)|$.*
- *Apply transpose Jacobian transformation $J_k^{-T}(\hat{\mathbf{x}}_q)$ to prepare for gradient application according to Eq.* (5).

(iii) Multiply by unit-cell gradient of shape functions and sum over all quadrature points. Accumulate value and each gradient component according to Eq. (5).

We emphasize that the implementation of this generic algorithm (except sum-factorization) is straight-forward in any finite element package, as the algorithm only makes use of the basic building blocks in finite element codes. However, an efficient operator computation necessitates different design choices compared to codes specialized at assembly. The most important choices are:

- Split the gradient computation into a unit-cell part and the application of the Jacobian transformation (fewer operations) instead of holding gradients in real space.
- Unit-cell operation is the same on all cells, so several cells (but not too many, in order for the local results to fit into caches) can be computed at once (matrix-vector product $\rightarrow$ matrix-matrix product, better computational properties).
- For tensor-product elements, one can apply sum-factorization that has lower complexity than straight-forward evaluation as nested loops over all quadrature points and local basis functions.

2.2 Sum-factorization

In case quadrature points and shape functions are constructed from tensor product of one-dimensional objects, sum-factorization can be used to efficiently evaluate all function values and derivatives at all quadrature points [45, 43]. The tensor-product form is straight-forward for quadrilateral/hexahedral elements, but can also be applied to tetrahedral elements, even though at a somewhat higher cost [34, Ch. 4].

On the unit cell, the evaluation of the first component of the gradient of a finite element function u^h based on a tensor-product basis can be written as

$$\frac{\partial}{\partial x_1} u^h(x_1,\ldots,x_d) = \sum_{i_d=1}^{p} \varphi_{i_d}(x_d) \ldots \sum_{i_2=1}^{p} \varphi_{i_2}(x_1) \sum_{i_1=1}^{p} \varphi'_{i_1}(x_1) u^{(i_1,\ldots,i_d)}, \tag{6}$$

where φ_* are the one-dimensional basis function and $(i_1,\ldots,i_d)$ is a multi-index accounting for the construction principle from a tensor product. For simplicity of notation, we skip the ^ symbol in this section that is used to denote unit-cell operations. The basic idea of the sum-factorization approach is to evaluate one sum at a time for all quadrature points in the respective coordinate direction, to store the intermediate results, and then to continue with the next direction. This can be interpreted as applying the basis functions and/or their derivative in one direction at a time. This evaluation corresponds to a matrix-matrix product with matrix dimensions $p \times p$ (all 1D basis functions evaluated at all 1D quadrature points) and $p \times p^{d-1}$ (input values $u^{(i_1,\ldots,i_d)}$, intermediate results), see also [18, 34, 20].

When computing components of the gradient, we can also combine parts of the computations for the individual components. For the derivative in x_2- and x_3-direction in 3D, the innermost sum in x_1-direction is exactly the same, and the temporary result after the first summation can be reused. In general, only $\frac{d(d+3)-2}{2}$ directional evaluations are necessary instead of the full d^2 when computing each gradient component separately. The cost per cell for the gradient evaluation is then $p^{d+1} \cdot (d(d+3)-2)$ arithmetic operations (additions and multiplications). Directional evaluations can also be combined when evaluating values u^h and gradients of u^h simultaneously.

Sum-factorization can also be applied to the backward transformation, i.e., the summation over all quadrature points for all basis functions according to (5).

2.3 Special case: Gauss–Lobatto elements

So far we have considered an arbitrary quadrilateral element with p^d basis functions and q^d quadrature points. Next, we present the special case of so-called Gauss–Lobatto elements (cf. [23]), which are Lagrangian elements based on the support points $\{r_1,\ldots,r_p\}$ of the pth order Gauss–Lobatto quadrature rule, a common element for higher order (spectral element) approaches [34]. The integrals are evaluated on the same points with the underlying Gauss–Lobatto quadrature formula. As a consequence of this construction principle, it holds that

$$\varphi_i(r_j) = \delta_{i,j}. \tag{7}$$

Due to this fact the mass matrix becomes diagonal, and thus trivially invertible. This is very appealing when finite elements are used to solve time-dependent partial differential equations with explicit time integrators since then no system of equation is to

be solved. Since the Gauss–Lobatto points cluster close to the cell boundaries, they also have better conditioning for higher order elements than Lagrangian elements with equidistant points (cf. [34, Chap. 2] and Sec. 6.2 below).

Relation (7) simplifies the computations in our tensor-product based gradient evaluation since the evaluation of the values at the quadrature points is an identity operator. At a point, $(r_{i_1},\dots,r_{i_d}) \in \{r_1,\dots,r_p\}^d$, it thus holds

$$u_k^h(r_{i_1},\dots,r_{i_d}) = u_k^{(i_1,\dots,i_d)}, \tag{8}$$

and

$$\frac{\partial}{\partial x_m} u_k^h(r_{i_1},\dots,r_{i_d}) = \sum_{j=1}^{p} u_k^{(i_1,\dots,i_{m-1},j,i_{m+1},\dots,i_d)} \varphi_j'(r_{i_m}), \quad m = 1,\dots,d. \tag{9}$$

Thus, the gradient on the whole cell can be computed with only $2d \cdot p^{d+1}$ arithmetic operations.

2.4 Discussion of computational complexity

We conclude this section by discussing the computational complexity of one matrix-vector product with the tensor-product based implementation compared to both a standard implementation of the finite element method and to finite differences. For this comparison, we focus on the discretization of the Laplacian with constant coefficient $K \equiv 1$ in equation (2), and take into account the fact that nodes located at cell boundaries are shared by several cells and hence touched more often.

Let us consider the most common case where the number of degrees of freedom (DoFs) equals the number of quadrature points on each cell. For the Laplacian, steps (2b)(i) and (2b)(iii) have the same costs. Then we have $2p^{d+1}\,(d(d+3)-2)$ arithmetic operations per cell or

$$2\frac{(d\cdot(d+3)-2)\cdot p}{(1-1/p)^d}$$

operations per DoF on average. Here, we disregard effects at the domain boundary and use that we have one cell per $(p-1)^d$ DoFs in the interior of the domain. Besides these operations on the unit cell, the application of the Jacobian transformations and quadrature weights involves additional operations, namely $4d^2+2d$ operations per quadrature point for general cells, and $4d+1$ for Cartesian cells.

For a standard implementation with a global (sparse) matrix the number of operations is given by twice the number of nonzero entries per row. The number of entries varies between p^d for interior points and $(2p-1)^d$ at the cell corners. Since the leading order term for the sparse matrix implementation is p^d while the one of the tensor product implementation is $d^2 p$, it depends on the combination of p and d which implementation needs less operations. Clearly, sum-factorization is more efficient the higher the order and the higher the dimension, cf. also the results in [20]. Fig. 1 compares the complexity in two and three dimensions for various element degrees.

For the Gauss–Lobatto case, the number of operations per DoF is given by $\frac{4d\cdot p}{(1-1/p)^d}$ so that the complexity is reduced by a factor d compared to the general case. The cost in the Gauss–Lobatto case is essentially twice the cost of a finite difference method on an equidistant grid for constant coefficients.

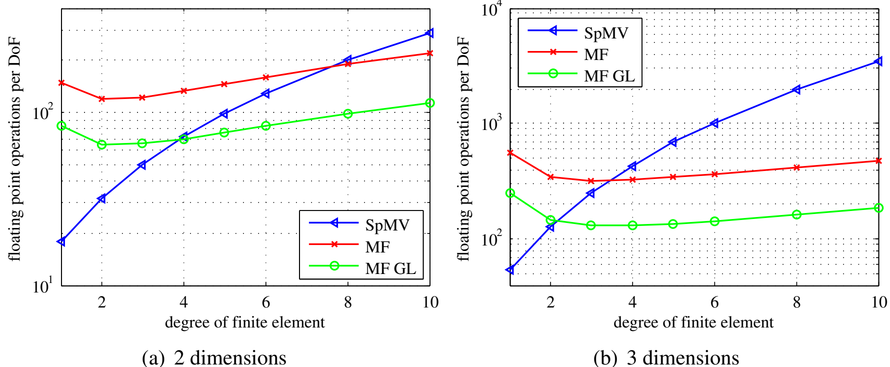


Figure 1: Comparison of number of floating point operations per DoF for sparse matrix and the proposed cell-based matrix-vector product. For the latter, both the general case with full Gauss quadrature (MF) and the Gauss–Lobatto case with inexact quadrature (MF GL) are included. Note the logarithmic scaling of the ordinate axis.

Note that the quadrature based implementation is very flexible since a variable (anisotropic) coefficient can easily be included (at the expense of $2d^2$ operations per quadrature point). This coefficient does not need to be fixed when assembling the matrix which is very useful for time-dependent or nonlinear problems where the coefficient changes from one operator application to the next. Any nonlinear function of the input variable can be assembled straightforwardly. Likewise, for systems of equations where different components couple, the coupling is introduced by a few operations on the quadrature points. The leading order costs do not change, as opposed to global matrix approaches where the coupling introduces additional matrix entries, see Sec. 6.3 on the Stokes equation below.

3 Implementation and data structures

In this section, we discuss efficient data structures for the finite element operator application described in Algorithm 2.1, together with the necessary geometry data for evaluation of derivatives according to Algorithm 2.2. The basic algorithm requires access to the following data:

(i) An optimized form of the very sparse matrix $P_k C$ that describes the local-to-global relations including constraints.

(ii) For each quadrature point, the inverse Jacobian $J^{-1}(\hat{\mathbf{x}}_q)$ and the Jacobian determinant.

(iii) The evaluation of the basis functions at all (one-dimensional) quadrature points, $\varphi_i(x_q)$ and their derivatives, on the unit cell.

The data structures for the indices (i) and the mapping information (ii) are in general different from cell to cell and therefore dominate the memory consumption, as opposed to the unit cell shape functions. Below, we devote one subsection to each of these data structures.

The setup of these structures requires only a subset of the operations that need to be done in a typical FE assembly routine. For a scalar problem, the setup costs are between five and twenty operator applications, depending on the element order and the mesh type (data for Cartesian meshes can be computed faster). For the scalar test problem considered in Sec. 5.1, the setup time is 0.8 seconds for $\mathcal{Q}_2$ elements, while one matrix-vector product costs about 0.09 seconds. For comparison, the assembly of a sparse matrix with the fastest Trilinos 10.6 initialization routine takes around 12 seconds (counting only the time of sparsity graph allocation and inserting the entries into the matrix). The time advantage compared to global matrices is particularly significant for time-dependent and non-linear problems, where global matrices need to be rebuilt from one iteration to the next.

Since we focus on computational aspects of the implementation, the data structures are designed to be memory-efficient and with low overhead from loops and branches. For a fixed algorithm, lower memory consumption implies a higher arithmetic intensity, which better suits computer architectures in use today and expected in the near future. This is a clear difference to any type of explicit storage of matrix entries (global sparse matrix, array of local element matrices), where arithmetic intensity is typically very low [47, Chapt. 7].

3.1 Storage of indices and constraints

The matrices P_k describing the relation between local and global degrees of freedom (DoFs) are special sparse matrices that contain exactly one entry equal to 1 per row. Therefore, the usual storage scheme for P_k is a list of indices for each DoF on the cells [34]. This list enables straight-forward implementation of elemental decomposition of the solution, $u_k = P_k u$ in Algorithm 2.1, and assembly of the local contribution to the global vector, $v \leftarrow v + P_k^T v_k$, by a loop over all local indices per cell.

In addition to indices, the description of finite element solutions often involves constraints on components of a global vector u, expressed in terms of adjacent degrees of freedom and constants. Constraints arise at hanging nodes between cells of different refinement level. For the theoretical justification of hanging node constraints, we refer to [51, 21, 53], and the algorithms and data structures used in `deal.II` are described in [7]. Moreover, many application problems constrain some degrees of freedom on the boundary, e.g. by Dirichlet type of boundary conditions. The general form for constraints on a degree of freedom x_i is

$$x_i = \sum_{j=0}^{N-1} c_{ij} x_j + b_i, \tag{10}$$

with constraint matrix $C = (c_{ij})_{i,j=0,\dots,N-1}$. In this equation, N denotes the total number of degrees of freedom (counted from zero). Hanging node constraints are typically homogeneous, e.g. $x_i = \frac{1}{2}x_{i-1} + \frac{1}{2}x_{i+1}$ for linear finite elements in two space dimensions, and Dirichlet boundary conditions are of the form $x_i = b_i$ with no indirections to other degrees of freedom. In the notation (10), the constraint matrix C represents an identity operation for unconstrained degrees of freedom x_i, namely $c_{ij} = \delta_{ij}$ and $b_i = 0$.

Constrained components are eliminated from vectors and matrices when solving linear systems because these do not contribute to the discrete solution space. Hence,

for consistent FE operator application during iterative processes these entries need to be filled with the respective data (10) before evaluating integrals based on local degrees of freedom. Likewise, integral contributions arising from testing with local DoFs are to be added to the entries a DoF is constrained to.

3.1.1 Strategies for applying constraints

There are two possibilities to implement constraints in Algorithm 2.1:

(i) Compute products $t_1 = Cu$ and $v = C^T t_2$ explicitly and apply cell operations ($u_k = P_k t_1,\ v_k = A_k(u_k)$, and $t_2 \leftarrow t_2 + P_k^T v_k$) using temporary vectors. This ensures minimal number of operations for applying constraints, but the constraint application is of low arithmetic intensity (sparse matrix-vector product) and hence limited by memory bandwidth on most computer systems.

(ii) Avoid temporaries by applying C while reading out global indices for computation on cell k, i.e., compute $(P_k C)u$ instead of $P_k(Cu)$. By computing $(P_k C)u$ for a few cells at a time, this variant interleaves memory intensive operations with parts of the algorithm that are computationally intensive (sum-factorization part). This promises considerably improved performance, even though $(P_k C)$ duplicates constraints on degrees of freedom with higher valence (vertices in 2D, vertices and edges in 3D).

Typically not more than about 30% of DoFs are constrained in hp-adaptive computations [7] and often less than 10% in the h-adaptive case. Hence, it is not efficient to store $(P_k C)$ in a generic sparse matrix format, as most rows consist of a single unit entry like P_k, introducing substantial computational overhead in form of loops of length 1. We found that a naive implementation of $(P_k C)u$ and $v \leftarrow v + (P_k C)^T v_k$ could easily dominate the total cost of operator application (up to 70–90% of costs) when all other parts in Algorithm (2.1) were done efficiently. Therefore, we propose an improved storage scheme for $(P_k C)$ below that is tailored to the elemental decomposition and global assembly operations. Our implementation also enables straightforward combination with process-local index spaces in MPI parallelization, which is more challenging for variant (i) because C and P_k access different ghost elements in general.

3.1.2 Efficient combination of constraints and local-to-global indexing

Our approach to store and apply $(P_k C)$ for each cell is to hold an array `indices_local_global` that stores the global indices for all the degrees of freedom that contribute to the local degrees of freedom. If a DoF is constrained, we replace it by the list of DoFs it is constrained to. In order to identify the local degrees of freedom with an expansion into constraint entries, we use an additional array `constraint_indicator` that stores the distance from one constrained DoF to the next, in conjunction with the actual weights c_{ij} for the constraints. With this data structure, the local information can be extracted with one loop that runs between the entries in `constraint_indicator` for regular entries, and expands the constraint entries else. This gives more regular data access and allows for loop unrolling between the constraints.

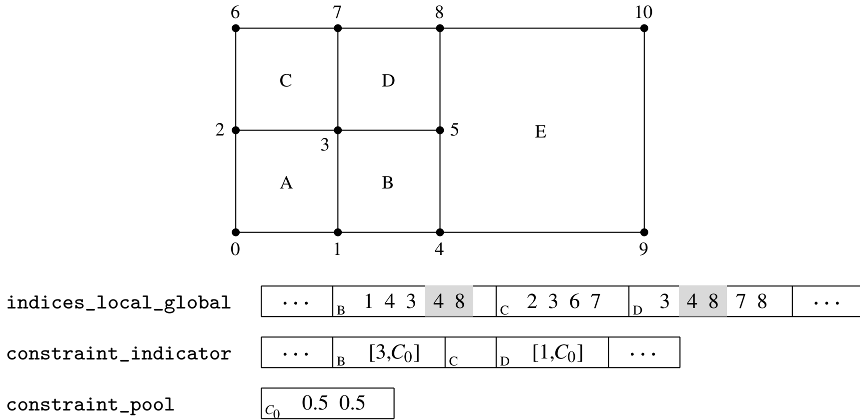


Figure 2: Sketch of the storage concept for degrees of freedom including direct access to indices related to constraints. In this example, the global degree of freedom 5 is a hanging node and replaced by $\{4,8\}$ (shaded).

Fig. 2 depicts the storage concept on a simple two-dimensional example with Q_1 elements. The degree of freedom 5 is a hanging node and gets constrained to the degrees of freedom 4 and 8 by the relation $x_5 = 0.5x_4 + 0.5x_8$, see also [7]. On cell B, the degree of freedom 5 is the last local index. As explained above, in `indices_local_global` we replace the global number 5 by the constraint expansion, i.e., the numbers $\{4,8\}$. The information for the position is encoded in `constraint_indicator`, where the distance from the start to the constrained index 3 is stored together with a pointer to the constraint weights in `constraint_pool`.

Taking all cells together, the three data fields `indices_local_global`, `constraint_indicator`, and `constraint_pool` are arrays of arrays. We store each of them row-wise in a contiguous range of memory. Since the length of the rows is in general not the same on different cells, we store an index with the start of individual rows, very similar to the storage scheme in compressed row storage for sparse matrices used in state-of-the-art linear algebra packages like Trilinos Epetra/Tpetra [30]. This yields linear memory access for both the array of indices and constraint indicators as we loop over cells.

From the data in Fig. 2 we see that the same constraint weight appears twice. For a larger mesh with Q_1 elements, all the hanging nodes have weight $\{0.5, 0.5\}$. Therefore, we store the weights only once. To detect identical weights in the general case, we implemented a simple hashing algorithm that associates an array of floating point values with an integer. This approach of finding similar weights from a sparse matrix representation of constraints is necessary because constraints often contain indirections to other constrained entries, which need to be resolved globally first [7]. This hashing approach could also be applied to design an improved constraint matrix C where not all floating point values for c_{ij} are stored, but only the ones that are mutually different.

The approach to store only *different* constraint weights motivates the name "constraint pool". We emphasize that the number of entries in the pool does usually not depend on the size of the problem. It depends merely on the space dimension and the

polynomial degree. For the 2D case with Q_1 elements and only hanging node constraints, there is only one single weight (the aforementioned combination $\{0.5, 0.5\}$). When both hanging nodes and Dirichlet boundary conditions are used, there are up to three weights, namely $\{0.5, 0.5\}, \{\}, \{0.5\}$, the second representing a constraint of the form $x_i = 0$ and the third for hanging nodes that are close to the boundary. On a 3D example with Q_4 elements and 2.3 million unknowns, our algorithm detects 50 different weight combinations out of 440,000 constraints, consuming only 5 kilobytes of data.

3.1.3 Inhomogeneous constraints

In problems where linear systems are to be solved, an additional difficulty arises when inhomogeneous constraints (e.g. Dirichlet boundary conditions) are present in the solution space. When working with sparse matrices and a right hand side vector, the standard procedure is to apply Gauss elimination on affected rows and columns. The elimination of columns in the matrix gives rise to some additional entries in the right hand side, coming from constrained degrees of freedom. However, for matrix-free implementations, matrix entries are never explicitly computed. Thus, it is not at all obvious how to perform this elimination efficiently. Therefore, we propose to set up the PDE problem such that the solution is computed with homogeneous data, and inhomogeneous data adds a forcing to the right hand side,

$$Au_{\text{hom}} = f - Ag_{\text{inhom}},$$

where $u = u_{\text{hom}} + g_{\text{inhom}}$ is the solution to the inhomogeneous problem, u_{hom} a solution to homogeneous data and g_{inhom} represents the inhomogeneous (boundary) data. The contribution Ag_{inhom} can be evaluated efficiently with our operator, too.

3.2 Jacobian transformation

The second building block for finite element operator application is the access to the transformation from unit to real cell. In our implementation, we choose to directly store the inverse Jacobian $J^{-1}(\hat{\mathbf{x}}_q)$ in an array for all quadrature points on all cells. For most problems, there is no significant negative impact on performance because the computationally intensive evaluation of the unit cell gradient $\sum_j \hat{\nabla}\phi_j(\hat{\mathbf{x}}_q)u_j$ precedes the use of J^{-1}. Hence, data prefetching can hide the latency of the access to the large Jacobian data.

Another option would be to compute the Jacobian on the fly, as is usual for assembly routines in many finite element libraries, including `deal.II`. The Jacobian J is evaluated from the shape gradients based on mapping support points (vertices for usual bi-/trilinear $\mathcal{Q}_1$ mappings), followed by inversion to get J^{-1}. We consider this less efficient because the inversion of J involves divisions which are more expensive than additions and multiplications on most computer architectures. Moreover, the support point information must be read in an irregular fashion similar to reading from solution vectors, which increases costs. Finally, the approach of storing the inverse Jacobians directly is completely generic and does not need to distinguish between the type of mapping (bi-/trilinear mapping, higher order parametric mappings, deforming

meshes). The mapping information is evaluated once in a setup phase, which gives considerable gains if the operator is applied several times in an iterative method.

The numerical approximation of integrals with the transformation rule also involves the weighting by the determinant of the Jacobian times the quadrature weight, $|\det J(\hat{\mathbf{x}}_q)|\, w_q$. We choose to also store this value in order to avoid its computation on the fly (division when computed from J^{-1}). We also provide the option to cache precomputed locations of the quadrature points in real space, which are necessary to evaluate variable coefficients that depend on the spatial coordinates, and second derivatives through storage of gradients of inverse Jacobians.

Compression of mapping data. The mapping data contributes by far the most to the total memory consumption of our implementation framework. For linear Lagrange elements and scalar problems, this is actually more data than needed for storing a sparse matrix, see also Fig. 6. For some special cases, the full generality of storing an individual Jacobian for each quadrature point on each cell can be avoided. Consider e.g. the Cartesian mesh in Fig. 2, where the inverse Jacobian is diagonal and constant within each cell, i.e.,

$$J^{-1}(\hat{\mathbf{x}}_q) = \operatorname{diag}\left(\frac{1}{h}, \frac{1}{h}\right).$$

Moreover, cells A, B, C, and D involve the same h. Another example are linear transformations where the Jacobian is constant throughout a cell. For meshes with triangles/tetrahedra, this is a very common case, whereas it is more rare for quadrilaterals (e.g. on a parallelepiped geometry). We check for similarities between the transformation data from one cell to another, besides checking for Cartesian and linear mappings. Similar weights on adjacent cells are then only saved once. We found this a good compromise between simple data structures and possible reductions in memory consumption. Compared to storing and applying the full transformations for all quadrature points of a cell, the cost for checking for these three possibilities is negligible. On Cartesian meshes, the memory consumption of the geometry information is dominated by one integer that encodes the cell type (0: Cartesian, 1: linear, 2: general), and the storage location of the data, whereas the actual data is only some hundreds of bytes long also on adaptively refined meshes.

Second derivatives. For H^1 conforming PDE problems with second derivatives, Hessians needs to be evaluated in the case when residuals of solutions are computed, like e.g. in error estimators [12, 24] and stabilized finite element methods [15, 54]. Taking the gradient of expression (4) and applying the product rule, we obtain

$$\nabla^2 u^h(\mathbf{x}_q) = J_k^{-1}(\hat{\mathbf{x}}_q)\left(\sum_j \hat{\nabla}^2 \phi_j(\hat{\mathbf{x}}_q) u_j\right) J_k^{-T}(\hat{\mathbf{x}}_q) + \nabla J_k^{-1}(\hat{\mathbf{x}}_q) \sum_j \hat{\nabla}\phi_j(\hat{\mathbf{x}}_q) u_j. \tag{11}$$

Hence, the Hessian is computed by applying the inverse Jacobian twice on the Hessian of u^h on the unit cell plus the gradient of the inverse Jacobian times the unit cell gradient. The gradient of the inverse Jacobian is given by

$$\nabla J_k^{-1}(\hat{\mathbf{x}}_q) = J_k^{-1}(\hat{\mathbf{x}}_q)\left(J_k^{-1}(\hat{\mathbf{x}}_q)\frac{\partial^2 \mathbf{F}_k(\hat{\mathbf{x}}_q)}{\partial^2 \hat{\mathbf{x}}}\right) J_k^{-1}(\hat{\mathbf{x}}_q),$$

where $\mathbf{F}_k$ denotes the transformation from unit to real cell. We also cache this data if problems with second derivatives are solved. On cells with linear or Cartesian transformations, the Jacobian is constant throughout the cell, so that the derivative $\nabla J^{-1}(\hat{\mathbf{x}}_q)$ is zero.

3.3 Computational kernels for cell operations

To perform a local integration, one has to perform Algorithm 2.2 for each cell. These cell-based operations are implemented in a class we call `FEEvaluation` where we provide three types of operations:

- Reading from vectors, assembling into global vectors (access through the index and constraint data).
- Compute shape values, gradients, or Hessians on the unit cell as required by the PDE problem using sum-factorization. The polynomial degree and the number of quadrature points are made templates and operate on object-local stack variables. This ensures thread-safety and negligible initialization costs as well as it minimizes loop overheads since the loop bounds in sum factorization are known at compile time and can be unrolled.
- Application of Jacobian transformation, multiplication with variable coefficients or other non-linear functions of the evaluated FE functions u^h. The class `FEEvaluation` provides read and write access to the internal data fields where values, gradients, and Hessians are stored. Users can access these fields to specify their particular equation.

4 Parallelization

Our implementation exploits parallelism on three levels, adapted to current computer architectures. On a large-scale level, we tailor our algorithms to be applicable to large clusters with thousands or more cores using the message passing interface (MPI) [44]. Moreover, we enable parallelism in shared memory, and finally use vector units available in most of today's processors.

4.1 MPI parallelization

On a distributed memory system, the domain is decomposed into subdomains of equal size, which are assigned to the individual processors in the system. In `deal.II`, each processor holds a common coarse mesh and refines only on those parts of the triangulation that belong to the local subdomain. This implementation is based on the `p4est` package [19] and the interface to `deal.II` described in [4]. When performing an element-based operation, data associated to DoFs on the subdomain boundaries need to be made available to all processors involved. Likewise, values accumulated for DoFs at subdomain boundaries need to be sent to the processor owning the DoF. In addition, hanging nodes located at or near subdomain boundaries create additional dependencies to data on other processors.

In order to handle the MPI communication, we have designed a special vector class, called `distributed::Vector`, that is tailored to the access patterns of our

implementation. To initialize a vector, one specifies the degrees of freedom that are owned by the processor (`locally_owned_dofs`) and the ones on subdomain boundaries where read or write access is needed (`ghost_dofs`). The `import_ghost` and `compress` functions implement the data exchange using non-blocking MPI send and receive commands. Since we expect the same vector to be used for several operator applications, we use persistent MPI communication requests that preserve the MPI channels from one call to the next [44, sec. 3.9]. This reduces the overhead of data exchange to a minimum. Note that for the `import_ghost` command the sender does not necessarily need to wait for the operation to complete, only the receiving processor needs to make sure it has obtained all data before accessing ghosts. However, we observe a considerably improved performance when also the sender waits.

Similarly to distributed sparse matrix storage in the PETSc [2, 3] and Trilinos [29, 30] linear algebra packages, we transform indices into process-local index space during the setup described in Sec. 3.1, as opposed to the `deal.II` DoFHandler and constraint matrix which store the global number of degrees of freedom. Ghost indices are appended at the end of the locally owned array, enumerated starting from the number of locally owned degrees of freedom. The localized indices provide direct access to vector entries with one single CPU instruction for all locally relevant degrees of freedom. Moreover, local degrees of freedom can be enumerated by 32-bit unsigned integers, whereas global indices require 64-bit indices once the problem size exceeds 4 billion unknowns. Our implementation keeps track of the global indices and the target processes for ghost indices and the list of locally owned DoFs that are ghosts on other processors and are about to be received/sent. This information is identified in the initialization step of the vector and referred to as "partitioner" of the vector. In order to avoid to build and store the partitioning information for several vectors based on the same layout, we share this information.

In the initialization of the operator data each processor orders its cells such that the cells that operate on DoFs that require communication are lumped together. We will refer to these cells as boundary cells and to the rest as inner cells in the following. During a DoF operation, one first touches parts of the inner cells, followed by the boundary cells, and finally the rest of the inner cells. In this way, one can overlap both communication steps (`import_ghots` and `compress`) with computations.

Note that an alternative would be to use PETSc or Trilinos vectors to handle the communications. However, these vector types are more general and come therefore with a larger overhead for ghost handling as compared to the vector type described above that is tailored for our purposes.

4.2 Shared-memory parallelization

The operation that computes the local result of the cell-based operator is independent from one cell to another. However, the operation that adds the data to a destination vector is not, because several cells share degrees of freedom along commonly owned vertices, edges, and faces. There are several possibilities to account for this issue in shared memory codes and to obtain thread-safety, cf. the discussion in [46]. We have chosen a strategy to subdivide cells into subdomains with precomputed neighbor relations. Then, we use Intel Threading Building Blocks [49] to schedule the tasks dynamically in a way that no neighboring cells are worked on simultaneously. We define

neighboring cells as those that share a common degree of freedom, either directly or through a constraint in the dependency graph of $(P_k C)$. Popular methods of this kind are based on coloring of the cells in such a way that cells of the same color are not adjacent, cf. e.g. [14, 31, 22, 25, 13, 37].

A drawback with coloring strategies are points of synchronization for all computational threads after each color as well as suboptimal use of cache memories. The authors have therefore proposed an alternative strategy [38] that works on two levels: First, the domain is partitioned in such a way that cells belonging to partition k do at most overlap with cells from partition $k-1$, k, and $k+1$. In this way, all odd (even) partitions are independent. To avoid an explicit synchronization point when switching from odd to even partitions and to split the work into smaller portions, we add a second level of partitioning: Within each partition, we subdivide the cells once more following the same strategy as on the first level. Then, the task for each partition on level one spawns tasks according to the partitioning on level two, and the Threading Building Blocks task scheduler can keep all computational threads busy with tasks.

In order to keep task scheduling costs low and improve the memory access pattern, we do not consider individual cells when we build the partitions but cluster a number of neighboring cells to blocks of cells. Since we want to combine the shared-memory parallelism with the MPI parallelization, we define all boundary cells as part of the first partition and schedule the corresponding task such that the partition is handled towards the middle of the computations. In this way, communication can be overlapped with computations on other partitions.

4.3 Vectorization

Most computer systems today offer a third level of parallelism which does not fit into the two categories above, namely vector operations within the CPUs in a single instruction multiple data (SIMD) fashion.

In most generic finite element packages, vectorization is not explicitly addressed and merely left as a task to an optimizing compiler. However, it is our experience that many compilers already fail at vectorizing matrix-matrix multiplications,[1] not to mention the more complicated data flow present in finite element operator applications with many short loops. Moreover, it is usually not efficient to rely on external vectorized packages like level-3 BLAS. The operations on cell level are too fine-grained for most practically relevant finite elements, so that function call overheads are prohibitive. At the same time, the finite element framework is ideally suited to the SIMD concept, since the operations are typically the same on all cells. Therefore, our implementation makes vectorization explicit by vector-type classes with overloading of basic arithmetic operations based on compiler intrinsics [32, 33]. For 128-bit SSE registers, this translates to 2 cells for double data types and 4 cells for single precision floats, and 4 or 8 for 256-bit AVX registers.

The elemental decomposition and global assembly steps in Algorithm 2.1 directly collect the data from n_{vec} cells and pack it into one vector data field for each local degree of freedom. The complete work-flow at cell level, i.e., step (2b) in Algorithm

[1] We tested matrix-matrix multiplication (matrix dimensions compile-time constants) with the GNU C++ compiler (version 4.5.0) and the Intel compiler (version 12.0.2) on x86-64 Linux at different optimization levels. The code was at best marginally faster than non-vectorized versions, sometimes considerably slower.

2.1, is performed using vectorized types instead of single data fields. Unit cell data is stored as n_{vec} same copies in each array entry, and mapping information on quadrature points is stored for several cells at once. When different cell types are involved within the vector components, the most general is applied.

5 Performance tests

All the experiments in this section have been performed with the GNU C++ compiler (version 4.5.2) on Linux x86-64, compiled with flags `-O2 -funroll-loops`. Sparse matrix-vector products use the kernel provided by the Trilinos version 10.6.4 [30] compiled at `-O3`. We perform benchmarks on two systems based on different processor architectures:

Nehalem-EP Intel Xeon E5520 (Nehalem-EP) at 2.26 GHz (turbo up to 2.53 GHz), 2 sockets with 4 cores each. Theoretical peak performance double precision: 72.3 Gflops, sustained memory bandwidth according to STREAM [42] benchmark: 24 GB/s. DDR Infiniband interconnect.

Harpertown Intel Xeon E5430 (Harpertown) at 2.66 GHz, 2 sockets with 4 cores each. Theoretical peak performance double precision: 85.1 Gflops, sustained memory bandwidth according to STREAM: 5.8 GB/s.

The Nehalem-EP system represents a system with state-of-the-art memory interfaces, whereas the Harpertown system has rather limited memory bandwidth. This difference will be reflected in the performance of memory-intensive operations like sparse matrix-vector products. However, we expect the ratio between arithmetic and memory performance on future systems to be more like the Harpertown system.

5.1 Matrix-vector products for Laplace problem

As a first example, we consider the finite element operation with the following operator:

$$Lu = -\Delta u, \tag{12}$$

which corresponds to a matrix-vector product with the matrix A being the stiffness matrix arising from discretizing the negative Laplacian. We have tested the performance by taking the wall-clock time needed for one matrix-vector product (minimum out of 40 runs). Fig. 3 shows the wall time for a 2D problem for various polynomial orders of the finite element. The problem size is held constant at 3.7 million degrees of freedom and there are only constraints from homogeneous boundary conditions, but no hanging nodes. Fig. 4 shows the same experiment for a 3D example with 1.8 million degrees of freedom on two different computer systems. We use a Gauss quadrature formula with as many points as local degrees of freedom, giving exact integration on Cartesian cells.

Comparing the performance results with the number of arithmetic operations displayed in Fig. 1, one can see a qualitative similarity. The computing time for our implementation is almost the same for polynomial orders two through six, which is due to the efficient tensor-product form. For the sparse matrices whose complexity behaves as p^d, on the other hand, the computing time is quickly increasing. The

experiments show that the performance of our cell-based operator outperforms the standard sparse matrix kernel for the Laplacian of the problem for any polynomial order but one. This is in contrast to the theoretical complexity of the sparse matrix that is better than sum-factorization up to degree eight in 2D and four in 3D, respectively, see Fig. 1. The reason for this difference are the different Gflops rates as reported in Table 1, firstly because the performance of the sparse matrix vector product is limited by the memory bandwidth and secondly that we have vectorized the cell operations in our implementation. Memory bandwidth limits become especially obvious when MPI parallelization is used on the eight cores of the computing node. In this case, parallel speedup is considerably improved by our implementation, especially on the Harpertown system where the access to main memory is slower. The reduced memory consumption of the cell-based strategy is illustrated in Fig. 6. Comparing the obtained Gflop rates with the peak performance, we see that we can reach up to 70% of peak arithmetic performance already at low order, while the sparse matrix does not exceed 15% of peak in serial and 7% in parallel (at 100% of memory throughput) on Nehalem-EP.

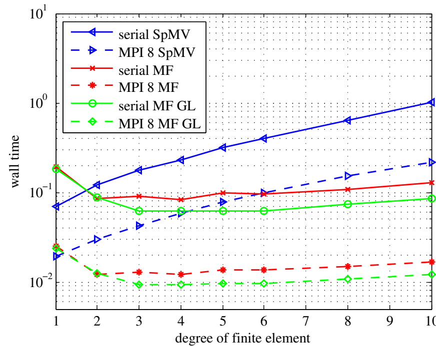


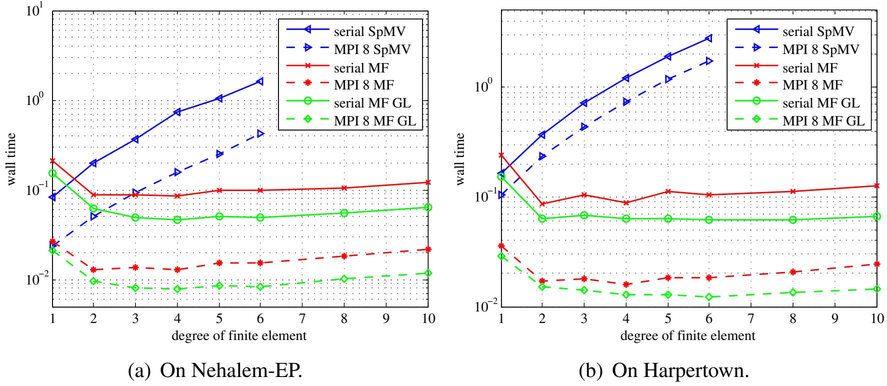
Figure 3: Wall-clock times per matrix-vector product for different polynomial orders with 2D Laplacian with 3.7 million degrees of freedom on Nehalem-EP.

In Fig. 5, we compare the performance in Gflops for different parallelizations as a function of the problem size. It shows that thread parallelization is less efficient for small problem sizes compared to the MPI-only parallelization, but the peak performance with more than about 100,000 degrees of freedom is quite similar.

We have also done experiments with single precision. Single precision is about 90 percent faster. This shows that the vectorization is efficient also for vector lengths larger than two, which we also have verified on a system with AVX.

5.1.1 Non-Cartesian meshes

We next consider a different geometry, namely a three dimensional ball with adaptive refinements applied, where the cells are non-Cartesian and higher order boundary mappings are applied, so that no simple compression can be applied when storing transformation data, see Section 3.2. Besides the increase in memory consumption, this also increases the number of operations on the cells slightly. This makes our sum-factorization operator application more expensive compared to meshes consist-



(a) On Nehalem-EP. (b) On Harpertown.

Figure 4: Wall-clock times per matrix-vector product for different polynomial orders with 3D Laplacian with 1.8 million degrees of freedom on two computer systems.

Table 1: Wall-clock times in seconds for one matrix-vector product on 3D Laplacian on a Cartesian mesh (Nehalem-EP). The problem size is held constant at 1.8 million degrees of freedom, compare with Fig. 4(a). Shared memory and MPI parallelization are compared.

	serial		8 cores (MPI)			8 cores (shared-mem.)		
Pol. order	time	Gflops	time	Gflops	speedup	time	Gflops	speedup
finite element operator application								
1	0.21	4.3	0.027	33	7.8	0.028	33	7.4
2	0.088	6.4	0.013	44	6.9	0.013	43	6.5
3	0.089	5.9	0.013	39	6.6	0.013	40	6.8
4	0.086	6.3	0.013	42	6.7	0.012	45	7.2
5	0.10	5.7	0.015	37	6.5	0.014	41	7.1
6	0.099	6.2	0.015	40	6.5	0.014	46	7.4
8	0.10	6.8	0.018	39	5.7	0.014	50	7.3
10	0.12	6.7	0.022	37	5.6	0.017	49	7.3
sparse matrix (CRS)								
1	0.083	1.1	0.024	4.0	3.5			
2	0.20	1.1	0.050	4.4	3.9			
3	0.37	1.2	0.094	4.6	3.9			
4	0.73	1.0	0.16	4.8	4.6			
5	1.1	1.1	0.25	4.8	4.2			
6	1.6	1.1	0.42	4.2	3.9			

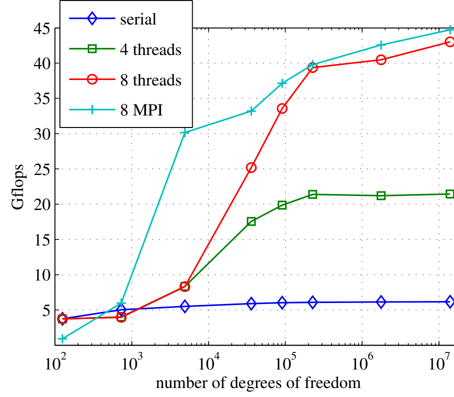


Figure 5: Performance over problem size with $\mathcal{Q}_2$ elements on a Cartesian mesh for serial computation, shared-memory parallelization, and MPI parallelization (Nehalem-EP).

Table 2: Laplace problem on a 3D ball (non-Cartesian cells, higher order boundary mapping, adaptive refinement). Wall-clock times per million degrees of freedom for one matrix-vector product in serial are given and flops for polynomial orders one to six with proposed operator and sparse matrix for serial runs. Test performed on Nehalem-EP.

	FE operator application		sparse matrix (CRS)	
Pol. order	memory [bytes/DoF]	time [s/10^6 DoFs]	memory [bytes/DoF]	time [s/10^6 DoFs]
1	670	0.19	320	0.050
2	280	0.090	740	0.11
3	190	0.078	1400	0.21
4	160	0.078	2400	0.34
5	150	0.080	3900	0.58
6	130	0.074	—	—
8	120	0.080	—	—

ing of Cartesian cells, especially for lower order approximations. However, looking at the numbers collected in Table 2, we can see that our operator again outperforms the sparse matrix, both in terms of memory requirements and computing time, for any finite element degree but linear (cf. also Fig. 6 for memory consumption). Note that the times in Table 2 are expressed in terms of million degrees of freedom in order to compensate for different problem sizes for different polynomial orders.

5.1.2 Breakdown of total computation time

Fig. 7 breaks down the times spent in different components of the operator application for $\mathcal{Q}_2$ and $\mathcal{Q}_8$ elements (normalized to the time per million degrees of freedom). We break down the time into the read and write operation (steps 2a and 2c in Algorithm 2.1), the application of unit cell gradients by sum-factorization (steps (i) and (iii) in Algorithm 2.2) and the application of the Jacobian transformation and the Laplace operation on quadrature points (step (ii) in Algorithm 2.2). For $\mathcal{Q}_2$ elements, the time for applying Jacobian transformation and operation is considerably larger for the ball

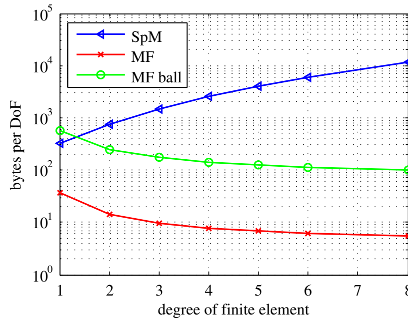


Figure 6: Memory requirements per DoF for a three dimensional example. The memory requirements for the sparse matrix (SpM) do no depend on the geometry, while for the matrix-free implementation we distinguish the case of Cartesian cells (MF) and the case with different Jacobians on each quadrature point for a ball geometry (MF ball).

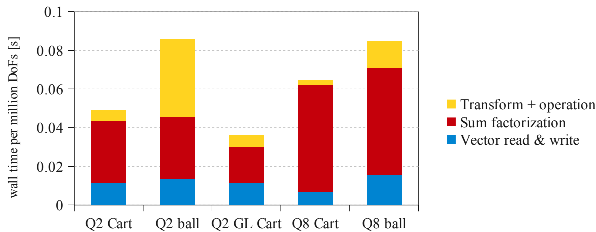


Figure 7: Breakdown of computing times for Cartesian mesh examples (no constraints) and ball meshes with adaptive refinements and boundary constraints. Element orders 2 and 8 are considered. For $\mathcal{Q}_2$ elements, we also include the times for a Gauss–Lobatto quadrature rule instead of full quadrature based on a Gauss formula.

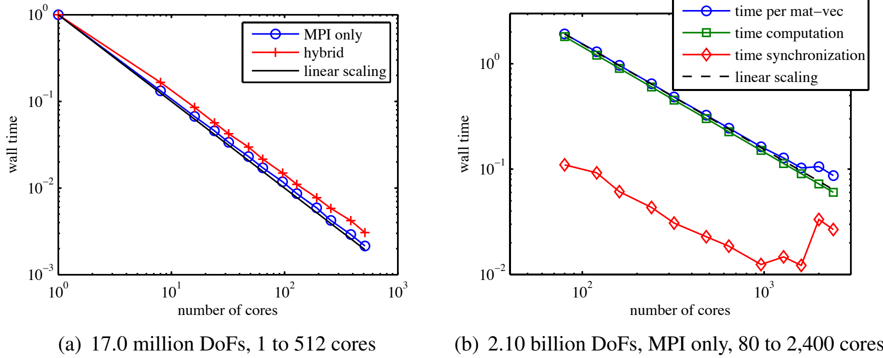
mesh because of high memory intensity, compared to a Cartesian mesh. When going to higher order, the time spent in the unit cell gradients increases, as it is the only component whose complexity depends on the element order. We also observe that the time for reading and writing into the vector is larger on the adaptively refined ball mesh where hanging node constraints need to be resolved. Nonetheless, the numbers show that no obvious bottleneck exists in our implementation, supporting the high Gflops rates shown above.

5.1.3 Influence of compiler optimizations

Finally, we consider the impact of the compiler on the performance on the test case in Table 1 with $\mathcal{Q}_2$ elements. Below we report Glops rates for the GCC compiler g++, version 4.5.0, and the Intel compiler `icpc`, version XE 12, with different compiler flags.

`g++`:	`-O1` 4.2 Gflops, `-O2` 4.7 Gflops, `-O3` 6.1 Gflops.
`g++ -funroll-loops`:	`-O1` 5.3 Gflops, `-O2` 6.1 Gflops, `-O3` 6.1 Gflops.
`icpc`:	`-O1` 3.5 Gflops, `-O2` 4.1 Gflops, `-O3` 4.1 Gflops, `-fast` 4.4 Gflops.

We see that the flag `-funroll-loops` has a considerable impact on performance at low levels of compiler optimization. Also, the Intel compiler performs considerably



(a) 17.0 million DoFs, 1 to 512 cores

(b) 2.10 billion DoFs, MPI only, 80 to 2,400 cores

Figure 8: Strong scaling of one matrix-vector product for finite element operator application on a cluster of Nehalem-EP.

worse than the GCC compiler, which is because of non-optimal code generation from SSE intrinsics.

5.2 Large-scale parallelism

Next we consider the scalability of the finite element operator application to large computer clusters. The results of strong scaling experiments are given in Fig. 8. According to the model outlined in section 4.1, the application only involves communication with nearest neighbors, which should give perfect scaling to very large numbers of processors.

In Fig. 8(a), we compare the performance of plain MPI parallelization with hybrid parallelization, i.e., MPI parallelization over different nodes of the cluster and thread parallelization within the nodes. The problem consists of 17.0 million degrees of freedom based on $\mathcal{Q}_2$ elements. We see that both variants show very good scaling behavior from 8 to 512 cores. At 512 cores, there are about 33,000 degrees of freedom per core. There is a slight advantage for the MPI-only parallelization, which is mainly due to the better performance of MPI within one node as presented in Table 1. This suggests that for our implementation there is no benefit from using hybrid parallelization on current systems. However, this might change for future systems where more heterogeneity within the nodes is to be expected and a static subdivision of the workload as done in MPI is suboptimal.

In Fig. 8(b), we display the time for one matrix-vector product on 80 to 2,400 MPI processes for a problem with 2.1 billion degrees of freedom (26 million to 870,000 DoFs per process). We separately collect computation times and synchronization times. The synchronization time collects the time spent in calls to MPI send and MPI wait functions, which initiate and finalize the data exchange, and also takes into account when the computational work on different processors does not take exactly the same amount of time. We see that the computation time scales perfectly and also the total time for matrix-vector product scales very well. This excellent scaling behavior is expected to translate to user cases with massive parallelism where matrix-vector products constitute the dominant part of the computation.

6 Application problems

In this section, we demonstrate the usability of our operator for more advanced problems, namely a nonlinear wave equation (Sine–Gordon problem), a multigrid solver, and Stokes/Navier–Stokes operators. The implementation of the former two examples is publicly available as tutorial programs `step-48` and `step-37` of the `deal.II` library.

6.1 Nonlinear wave equation

We consider the Sine–Gordon equation,

$$u_{tt} = \Delta u - \sin(u), \tag{13}$$

complemented by homogeneous Neumann boundary conditions and u and u_t specified at the initial time. We choose leap frog as time discretization using the second-order formulation of the partial differential equation. In weak form, the scheme reads

$$\left(v, u^{n+1}\right) = \left(v, 2u^n - u^{n-1} - \Delta t^2 \sin(u^n)\right) - \left(\nabla v, \Delta t^2 \nabla u^n\right), \tag{14}$$

where v denotes a test function. We choose to make use of the explicit time stepping with Gauss–Lobatto based elements that yield a diagonal mass matrix M, which gives the following update formula for the node values U^{n+1}:

$$U^{n+1} = M^{-1} L(U^n, U^{n-1}), \tag{15}$$

where the action of the (nonlinear) operator $L(U^n, U^{n-1})$ corresponds to the right hand side in (14). This test represents a typical situation in practical applications of the finite element method.

Table 3 shows computation times for one time step with the above implementation for two and three space dimensions and different element orders. Comparing first just the linear part, i.e., (14) without the sine term, we see that the performance of the finite element operator application is considerably higher than with sparse matrices. In the table, we also show the numbers for an implementation with a standard `deal.II` assembly loop that is optimized for matrices and not just vector assembly. Compared to that implementation, our operator is more than an order of magnitude faster, for 3D problems with $\mathcal{Q}_4$ elements almost two orders of magnitude.

Regarding the performance for the complete Sine–Gordon problem (14), we compare the proposed finite element operator application and a standard assembly routine. From Table 3, we see that the difference between the two variants is less than in the linear case, which is because about 60% of the computation time is spent in evaluation of the sine for the FE operator application (all other operations are memory accesses, multiplications, and additions).

6.2 Multigrid solver

To demonstrate that state-of-the-art linear solvers like multigrid highly benefit from fast matrix-vector products, we consider the solution of the three-dimensional Poisson problem

$$-\nabla \cdot a(\mathbf{x}) \nabla u = 1, \quad \text{on } [0,1]^3 \tag{16}$$

Table 3: Wall-clock time for wave equation ((14) without the sine term) and the full Sine-Gordon equation (14). *MF* denotes the finite element operator application, *SpMV* a matrix–vector product with a sparse matrix, and *dealii* a usual assembly loop in `deal.II` . There are 65,536 cells for the 2D tests and 32,768 cells for the 3D tests. Test performed on Nehalem-EP with one thread.

	wave equation			Sine-Gordon	
	MF	SpMV	dealii	MF	dealii
2D, $\mathcal{Q}_2$	0.0106	0.00971	0.109	0.0243	0.124
2D, $\mathcal{Q}_4$	0.0328	0.0706	0.528	0.0714	0.502
3D, $\mathcal{Q}_2$	0.0151	0.0320	0.331	0.0376	0.364
3D, $\mathcal{Q}_4$	0.0918	0.844	6.83	0.194	6.95

where $a(\mathbf{x}) = \frac{1}{0.1+2\|\mathbf{x}\|^2}$ is a smoothly varying coefficient, and homogeneous Dirichlet boundary conditions are prescribed. We choose a conjugate gradient solver accelerated by a geometric multigrid preconditioner based on a polynomial Chebyshev smoother [1]. This smoother only requires the action of the differential operator (which we implement with our finite element operator instead of sparse matrices) and knowledge of the diagonal elements in the operator which are cheap to obtain. Relaxation-based smoothers like Gauss–Seidel or ILU, on the other hand, require explicit knowledge of all matrix elements. For a mildly varying coefficient as in this example, the Chebyshev smoother gives optimal complexity. Note that more complicated problems require stronger multigrid smoothers [27] or algebraic multigrid approaches [26], where fast matrix-vector products can be used on the finest level. We estimate the largest eigenvalue $\tilde{\lambda}_{\max}$ for the Chebyshev smoother on each level by computing 10 iterations with a conjugate gradient algorithm, set the smoothing to $[0.12\tilde{\lambda}_{\max}, 1.2\tilde{\lambda}_{\max}]$, and choose a degree of 6 (i.e., 6 matrix-vector products per level and iteration). The CG solver is run until the residual is reduced by 10^{-12}.

In Table 4 we show computation times for finite elements of polynomial degrees 2, 4, and 6. We observe a speedup of a factor two for second order polynomials and more than a factor three for higher orders compared to an implementation based on sparse matrices. Note that for the $\mathcal{Q}_6$ case, about 80% of the computational time for the FE operator implementation is spent in restriction and prolongation operations, which are based on (non-optimal) sparse matrices. The gains can be expected to be even larger when also level transfer operations are implemented similarly to our FE operator. We note that the higher-order elements are based on non-equidistant Gauss–Lobatto–Legendre support points. In this case, the number of iterations needed is very small (4–6) and independent of the problem size. However, if one uses equidistant points instead, the performance becomes worse with increasing order due to ill-conditioning of the interpolation [34]. For solving the problem with $\mathcal{Q}_4$ elements, 22 iterations are required and with $\mathcal{Q}_6$ elements, more than 2 000 iterations are needed.

6.3 Performance of Stokes and Navier–Stokes operations

As demonstrated in section 6.1, the framework for finite element operator application is suited not only for matrix-vector products, but also nonlinear operations and systems of PDEs. Here we show timings for the incompressible Stokes and Navier–Stokes equations in fluid dynamics. We consider the operation described by the following

Table 4: Number of CG iterations for multigrid-preconditioned Poisson solve and wall-clock time in seconds. *MF* denotes the matrix-vector products with the finite element operator application, *SpMV* matrix–vector products with a sparse matrices. Test performed on Nehalem-EP with eight threads.

	4 levels (512 cells)			6 levels (32 768 cells)			
	#its	MF time	SpMV time	#its	MF time	SpMV time	MF time per million DoF
$\mathscr{Q}_2$	4	0.0169	0.0117	5	0.4135	0.9753	1.51
$\mathscr{Q}_4$	6	0.191	0.361	5	6.56	22.6	3.06
$\mathscr{Q}_6$	6	1.241	3.51	6	70.3	290	9.78

Table 5: Wall-clock time for 20 applications of the Stokes and Navier–Stokes operator (17). *FE Op* denotes the finite element operator application and *SpMV* a sparse matrix-vector product. Test performed on Nehalem-EP with one and eight threads (FE operator with shared memory parallelization).

	FE operator		SpMV		Speedup
	time	Gflops	time	Gflops	FE op over SpMV
Stokes, serial	0.293	5.25	1.30	1.04	4.44
Stokes, 8 threads	0.0487	31.5	0.486	2.77	9.98
N–S, serial	0.320	5.18			
N–S, 8 threads	0.0524	31.6			

weak form

$$\left(\mathbf{v}, \rho\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}\right)\right) + \left(\nabla\mathbf{v}, \mu(\nabla\mathbf{u} + \nabla\mathbf{u}^T) - p\mathbf{I}\right) + (q, \nabla\cdot\mathbf{u}). \tag{17}$$

Here, $\mathbf{u}$ denotes the vector-valued fluid velocity and p the fluid pressure, $\mathbf{v}$ and q test functions on the velocity and pressure space, respectively, and ρ and μ are the fluid's density and dynamic viscosity. For the Stokes equations, the first term is omitted and the operation described by (17) represents one matrix-vector product. The matrix-vector product is the operation that consumes the largest part of computing time in solving linear systems of equations with iterative solvers. For the Navier–Stokes equations, the operation corresponds to evaluating the nonlinear residual to a given solution $(\mathbf{u}, p)$ during time-stepping, where the terms from time discretization at older time levels are omitted. We perform a test with the so-called Taylor–Hood finite element pair $(\mathscr{Q}_2, \mathscr{Q}_1)$ which selects tri-quadratic elements for each velocity component and tri-linear elements for the pressure. The mesh represents a shell geometry in three space dimensions and consists of 6,144 cells with 163,704 degrees of freedom (velocity and pressure together).

Table 5 compares the performance of the finite element operator application with a usual sparse matrix-vector product for the Stokes problem. The symmetric gradient in (17) introduces a coupling between the different components, which results in a denser matrix, and is necessary if the viscosity is non-constant. In the cell-based operator, this coupling is merely introduced at quadrature points when defining the Stokes operation, which is very cheap in comparison. The operator application is more than four times faster than a SpMV already for the comparably low polynomial order of 2 for velocity in the serial case and ten times faster in the parallel case. This speedup can be expected to have big impact in many vector-valued problem with practically rele-

vant polynomial orders where matrix-vector products constitute the main cost. From Table 5 we also note that the difference in computing time between evaluating the Stokes and Navier–Stokes operation is small. This shows that more complicated operations do not have significant impact on the performance of the cell-based implementation, as long as this involves only some combinations on quadrature points and not too many additional function values or gradients.

7 Discussion and conclusions

We have presented a cell-based implementation of finite element operations that is parallelized on three levels. Compared to state-of-the-art implementations based on sparse matrices, we have demonstrated tremendously improved Gflops rates and better performance for all polynomial degrees but one. There are various reasons for this effect, namely usage of the tensor-product form of basis functions, regular data structures on element level, vectorization of operations on cell level, and a reduced memory traffic since there is no need to load the elements of a matrix but the unit-cell shape information is reused. The latter point particularly improves parallel scalability within one computing node. Especially for high-order methods in multiple dimensions and on Cartesian grids, our implementation remarkably reduces the memory requirements which also allows to increase the problem size that can be computed on a single node. Almost perfect strong scalability has been demonstrated also for thousands of processors. In order to achieve such high Gflop rates, we have implemented data structures tailored to a cell-based FE operator application.

In this paper, we did not explicitly target GPU-based programming, but the data structures outlined here fit well into the concept of GPU-programming, with elements from shared-memory parallelization (see Sec. 4.2) and vectorization (see Sec. 4.3) as basic building blocks. We expect that future generic finite element packages will build around building blocks like operator application that is targeted to the specific hardware in use.

The method presented here is not efficient for problems with linear elements and constant coefficients because the quadrature loop is expanded in every case and hence the operator is applied in two steps: first interpolation and then integration. Taking also into account that all nodes for linear continuous elements are vertex nodes with high valence (approximately 2^d), the number of operations per matrix-vector product is considerably higher than for sparse matrices. Nevertheless, in the two step formulation presented here one can exploit the tensor product structure of the basis functions to keep the number of operations small. This is particularly useful for Gauss–Lobatto elements where shape values on quadrature points form an identity operation. We give evidence that our implementation performs much better than sparse matrices already for second order elements.

In addition, the decomposition of the local matrix application in substeps with sums over quadrature points are advantageous for more complicated problems with time-dependent coefficients or non-linear terms where a pre-assembling of a sparse matrix is not possible. As demonstrated in Sec. 6.1, such a problem can be implemented — based on pre-assembled unit-cell shape information — in a few lines of code with very little increase in computational costs compared to linear/constant-

coefficient problems. Moreover, the framework presented here might make nonlinear solvers like nonlinear GMRES [16] for finite element discretizations more attractive. Our implementation is available as part of the `deal.II` package and two tutorial programs serve as a starting point for user-defined problems.

To sum up, our formulation of the finite element operation combines the simplicity and efficiency of stencil-based finite difference methods with the flexibility of finite elements when it comes to mesh transformation, adaptive mesh refinement, and variable coefficients.

8 Acknowledgments

The authors thank W. Bangerth, Texas A&M University, for valuable discussions and comments on the manuscript. Also, discussions with M. Gustafsson and E. Rudberg, Uppsala University, are acknowledged. The authors were supported by the Graduate School in Mathematics and Computing (FMB). The computations were performed on resources provided by SNIC through Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX) under projects p2003013, p2005005, and p2010002.

References

[1] M. Adams, M. Brezina, J. Hu, and R. Tuminaro. Parallel multigrid smoothing: polynomial versus Gauss–Seidel. *J. Comput. Phys.*, 188:593–610, 2003.

[2] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.1, Argonne National Laboratory, 2010.

[3] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2011. `http://www.mcs.anl.gov/petsc`.

[4] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler. Algorithms and data structures for massively parallel generic finite element codes. *ACM Trans. Math. Softw.*, 38(2), 2011.

[5] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007.

[6] W. Bangerth and G. Kanschat. `deal.II` *Differential Equations Analysis Library, Technical Reference*. `http://www.dealii.org`.

[7] W. Bangerth and O. Kayser-Herold. Data structures and requirements for *hp* finite element software. *ACM Trans. Math. Softw.*, 36(1):4/1–4/31, 2009.

[8] R. E. Bank. *PLTMG: A software package for solving elliptic partial differential equations*. SIAM, Philadelphia, 1998. Users' guide 8.0.

[9] P. Basini, C. Blitz, E. Bozdağ, E. Casarotti, M. Chen, H. N. Gharti, V. Hjörleifsdóttir, S. Kientz, D. Komatitsch, J. Labarta, N. L. Goff, P. L. Loher, Q. Lui, Y. Luo, A. Massi, F. Magnoni, R. Martin, D. McRitchie, M. Meschede, D. Michéa, T. Nissen-Meyer, D. Peter, B. Savage, B. Schuberth, A. Sieminski, L. Strand, C. Tape, J. Tromp, and H. Zhu. SPECFEM 3D user manual. Technical report, Computational Infrastructure for Geodynamics, Princeton University, University of PAU, CNRS, and INRIA, 2011.

[10] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. Part I: Abstract framework. *Computing*, 82(2–3):103–119, 2008.

[11] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. Part II: Implementation and tests in DUNE. *Computing*, 82(2–3):121–138, 2008.

[12] R. Becker and R. Rannacher. An optimal control approach to a posteriori error estimation in finite element methods. *Acta Numerica*, 10:1–102, 2001.

[13] M. Benantar and J. E. Flaherty. A six-color procedure for the parallel solution of elliptic systems using the finite quadtree structure. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, pages 230–236, 1990.

[14] P. Berger, P. Brouaye, and J. C. Syre. A mesh coloring method for efficient MIMD processing in finite element problems. In *Proceedings of the International Conference on Parallel Processing, ICPP'82*, pages 41–46, Bellaire, Michigan, USA, 1982. IEEE Computer Society.

[15] M. Braack, E. Burman, V. John, and G. Lube. Stabilized finite element methods for the generalized Oseen problem. *Comput. Meth. Appl. Mech. Engrg.*, 196:853–866, 2007.

[16] P. N. Brown and Y. Saad. Hybrid Krylov methods for nonlinear systems of equations. *SIAM J. Sci. Comput.*, 11:450–481, 1990.

[17] A. M. Bruaset and H. P. Langtangen. A comprehensive set of tools for solving partial differential equations; DiffPack. In M. Dæhlen and A. Tveito, editors, *Numerical Methods and Software Tools in Industrial Mathematics*, pages 61–90. Birkhäuser, Boston, 1997.

[18] P. E. Buis and W. R. Dyksen. Efficient vector and parallel manipulation of tensor products. *ACM Trans. Math. Softw.*, 22:18–23, 1996.

[19] C. Burstedde, L. C. Wilcox, and O. Ghattas. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM J. Sci. Comput.*, 33(3):1103–1133, 2011.

[20] C. D. Cantwell, S. J. Sherwin, R. M. Kirby, and P. H. J. Kelly. From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements. *Comput. Fluids*, 43:23–28, 2011.

[21] G. F. Carey. *Computational Grids: Generation, Adaptation and Solution Strategies*. Taylor & Francis, 1997.

[22] G. F. Carey, E. Barragy, R. McLay, and M. Sharma. Element-by-element vector and parallel computations. *Commun. Appl. Numer. Meth.*, 4:299–307, 1988.

[23] G. C. Cohen. *Higher-Order Numerical Methods for Transient Wave Equations*. Springer Verlag, Berlin, 2002.

[24] K. Eriksson, C. Johnson, and A. Logg. Adaptive computational methods for parabolic problems. In *Encyclopedia of Computational Mechanics*. John Wiley & Sons, Ltd, 2004.

[25] C. Farhat and L. Crivelli. A general approach to nonlinear finite-element computations on shared-memory multiprocessors. *Comput. Meth. Appl. Mech. Engrg.*, 72(2):153–171, 1989.

[26] M. W. Gee, J. J. Hu, and R. S. Tuminaro. A new smoothed aggregation multigrid method for anisotropic problems. *Numer. Linear Algebra Appl.*, 16:19–37, 2009.

[27] D. Göddeke and R. Strzodka. Mixed precision GPU-multigrid solvers with strong smoothers. In J. Kurzak, D. A. Bader, and J. J. Dongarra, editors, *Scientific Computing with Multicore and Accelerators*, chapter 7. CRC Press, 2010.

[28] B. Gustafsson. *High Order Difference Methods for Time Dependent PDE*, volume 38 of *Springer Series in Computational Mathematics*. Springer, 2008.

[29] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B.

Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31:397–423, 2005.

[30] M. A. Heroux et al. Trilinos Web page, 2011. `http://trilinos.sandia.gov`.

[31] T. J. R. Hughes, R. M. Ferencz, and J. O. Hallquist. Large-scale vectorized implicit calculations in solid mechanics on a cray X-MP/48 utilizing EBE preconditioned conjugate gradients. *Comput. Meth. Appl. Mech. Engrg.*, 61(2):215–248, 1987.

[32] Intel Corporation. *Intel C++ Compiler for Linux Intrinsics Reference*, 2006. Ref. no. 312482-001US, `http://software.intel.com/file/6373`.

[33] Intel Corporation. *Intel Advanced Vector Extensions Programming Reference*, December 2010. Ref. no. 319433-009, `http://software.intel.com/file/33301`.

[34] G. E. Karniadakis and S. J. Sherwin. *Spectral/hp element methods for computational fluid dynamics*. Oxford University Press, 2nd edition, 2005.

[35] B. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. `libMesh`: A C++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22(3–4):237–254, 2006.

[36] A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *J. Comput. Phys.*, 228(21):7863–7882, 2009.

[37] D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michéa. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. *J. Comput. Phys.*, 229:7692–7714, 2010.

[38] K. Kormann and M. Kronbichler. Parallel finite element operator application: Graph partitioning and coloring. Accepted for publication in the proceedings of the 7th IEEE International Conference on e-Science., 2011.

[39] H. P. Langtangen. *Computational Partial Differential Equations: Numerical Methods and Diffpack Programming*. Texts in Computational Science and Engineering. Springer Verlag, 2003.

[40] A. Logg. Automating the finite element method. *Arch. Comput. Meth. Eng.*, 14(2):93–138, 2007.

[41] A. Logg and G. N. Wells. DOLFIN: Automated finite element computing. *ACM Trans. Math. Softw.*, 37(2):1–28, 2010.

[42] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers, 1991–2007. A continually updated technical report. `http://www.cs.virginia.edu/stream`.

[43] J. M. Melenk, K. Gerdes, and C. Schwab. Fully discrete h,p-finite elements: fast quadrature. *Comput. Meth. Appl. Mech. Engrg.*, 190:4339–4364, 2001.

[44] Message Passing Interface Forum. MPI: A message-passing interface standard (version 2.2). Technical report, `http://www.mpi-forum.org`, 2009.

[45] S. A. Orszag. Spectral methods for problems in complex geometries. *J. Comput. Phys.*, 37:70–92, 1980.

[46] O. Pantalé. Parallelization of an object-oriented FEM dynamics code: influence of the strategies on the speedup. *Adv. Engrg. Softw.*, 36:361–373, 2005.

[47] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann, Burlington, 4th edition, 2009.

[48] B. Patzák and Z. Bittnar. Design of object oriented finite element code. *Adv. Eng. Software*, 32(10–11):759–767, 2001.

[49] J. Reinders. *Intel Threading Building Blocks*. O'Reilly Media, Sebastopol, CA, 2007.

[50] Y. Renard and J. Pommier. Getfem++. Technical report, INSA Toulouse, available from `http://download.gna.org/getfem/html/homepage/`, 2006.

[51] W. C. Rheinboldt and C. K. Mesztenyi. On a data structure for adaptive finite element mesh refinements. *ACM Trans. Math. Softw.*, 6:166–187, 1980.

[52] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, second edition, 2003.

[53] P. Šolín, J. Červený, and I. Doležel. Arbitrary-level hanging nodes and automatic adaptivity in the hp-FEM. *Math. Comput. Sim.*, 77:117–132, 2008.

[54] T. E. Tezduyar. Finite elements in fluids: Stabilized formulation and moving boundaries and interfaces. *Comput. Fluids*, 36:191–206, 2007.

[55] S. Turek, D. Göddeke, C. Becker, S. H. M. Buijssen, and H. Wobker. FEAST – realisation of hardware-oriented numerics for HPC simulations with finite elements. In *Concurrency and Computation: Practice and Experience*, volume 22 (6), pages 2247–2265, 2010. Special Issue Proceedings of ISC 2008.

[56] R. Vuduc, J. W. Demmel, and K. A. Yelick. The optimized sparse kernel interface (OSKI) library. Technical report, Berkeley Benchmarking and Optimization Project, University of California, Berkeley, `http://bebop.cs.berkeley.edu/oski`, 2007.

[57] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. W. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. SC2007: High performance computing, networking, and storage conference*, pages 10–16, 2007.