Computers & Fluids 63 (2012) 135–147

Contents lists available at SciVerse ScienceDirect

Computers & Fluids

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

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

Martin Kronbichler [*], Katharina Kormann

Division of Scientific Computing, Department of Information Technology, Uppsala University, Box 337, 75105 Uppsala, Sweden

ARTICLE INFO

Article history:
Received 6 October 2011
Received in revised form 4 April 2012
Accepted 10 April 2012
Available online 21 April 2012

Keywords:
Finite/spectral element method
Matrix-free method
Sum-factorization
Hybrid parallelization

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.

1. Introduction

Finite element problems are often solved by evaluating discrete differential operators like in iterative linear solvers [1] or time stepping schemes. This makes the operator evaluation—in particular matrix–vector products for the linear case—the central and usually most time-consuming component in finite element codes. Nonlinear problems are usually handled by 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 [2,3], DiffPack [4,5], DUNE [6,7], FEniCS/Dolfin [8,9], Getfem++ [10], libMesh [11], OOFEM [12], or PLTMG [13] separate finite element computations from linear algebra and rely on sparse matrix–vector (SpMV) kernels in iterative solvers, either by direct implementation or interfaces to specialized linear algebra packages like PETSc [14,15] or Trilinos [16,17]. In this article, we want to challenge the view of strictly separating linear algebra from finite element assembly routines.

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 [18, chapter 7]. Even though attempts have been made to tune SpMV kernels [19,20] or to allow only problems with structured non-zero pattern [21], Gflops rates rarely exceed 2–20% of peak arithmetic throughput.

Besides poor performance characteristics of sparse 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 restructuring reduces the computational complexity to d^2p operations per degree of freedom and is usually referred to as sum-factorization in the literature [22].

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. [24,25]. However, those approaches rather increase memory consumption

* Corresponding author. Tel.: +46 184712990.
E-mail addresses: kronbichler.martin@gmail.com (M. Kronbichler), katharina.kormann@it.uu.se (K. Kormann).

0045-7930/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.compfluid.2012.04.012

because of data duplication for nodes of higher valence and are therefore rarely efficient. Recently, cell-based strategies without explicit matrix storage have been considered for GPU programming [26,27], and used in application-specific software like SPECFEM 3D [28]. 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 mesh adaptation. Cantwell et al. [29] have compared 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 the `deal.II` library for the ease of use). The discussion of efficient algorithms with particular focus on memory efficiency for obtaining high computational performance is the major contribution of this article, including explicitly low and moderate order finite elements. Our framework includes constraints to enable adaptive mesh refinement with hanging nodes. The implementation is tailored to exploit parallelism, including vectorization, shared and distributed memory computations on thousands of processors. This approach is also beneficial for nonlinear PDEs and time-dependent coefficients where the assembly of a constant matrix is not possible in the first place.

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 has particularly nice properties. Section 3 presents adapted data structures and Section 4 the parallelization strategies. In Section 5, performance tests for a sample problem are collected and discussed. Section 6 considers more advanced uses of the operator application for linear and nonlinear problems. Finally, Section 7 concludes the article.

2. Cell-based implementation of FE operations

Let us consider a finite element Galerkin approximation of a (possibly nonlinear) operator A that takes a vector u as input and computes the integrals of the operator multiplied by trial functions ϕ_i, $i = 1, \ldots, n$, giving an output vector v. The operation can be expressed 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. Section 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 that 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. [30].

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

Algorithm 2.1

```
(Prototype finite element operator application)
1. update_ghost_values: 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 efficient
       quadrature.
   (c) Add the local contributions into the global result vec-
       tor, $v \leftarrow v + (P_k C)^T v_k$.
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: Evaluate the FE function u^h and/or its derivatives on all quadrature points and test by all trial functions related to the cell.
- Compute matrix representation A_k on the fly 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 [8,9].

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

2.1. Local quadrature approach

As a prototype operation, let us consider the Laplacian with variable coefficient,

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

where $K(\mathbf{x})$ is a symmetric $d \times d$ matrix. The corresponding FE operator evaluates the weak form

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

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 $\{\phi_j, j = 1, \ldots, n\}$ is the set of all basis functions tested with in the weak form. Let us further denote by $\{\hat\phi_j(\hat{\mathbf{x}}), j = 1, \ldots, p^d\}$ 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 $\mathbf{x}_q$,

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

where ∇ denotes the gradient in real coordinates, $\hat\nabla$ the gradient on the unit cell, and $J_k^{-T}(\hat{\mathbf{x}}_q)$ is the inverse transposed 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.

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

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