Computer Physics Communications 217 (2017) 99–116

Contents lists available at ScienceDirect

Computer Physics Communications

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

IGA-ADS: Isogeometric analysis FEM using ADS solver☆

Marcin M. Łoś [a,*], Maciej Woźniak [a], Maciej Paszyński [a], Andrew Lenharth [b], Muhamm Amber Hassaan [b], Keshav Pingali [b]

[a] AGH University of Science and Technology, Kraków, Poland
[b] The University of Texas at Austin, TX, USA

ARTICLE INFO

Article history:
Received 5 September 2016
Received in revised form 18 February 2017
Accepted 24 February 2017
Available online 10 April 2017

Keywords:
Finite element method
Isogeometric analysis
Shared memory

ABSTRACT

In this paper we present a fast explicit solver for solution of non-stationary problems using L^2 projections with isogeometric finite element method. The solver has been implemented within GALOIS framework. It enables parallel multi-core simulations of different time-dependent problems, in 1D, 2D, or 3D. We have prepared the solver framework in a way that enables direct implementation of the selected PDE and corresponding boundary conditions. In this paper we describe the installation, implementation of exemplary three PDEs, and execution of the simulations on multi-core Linux cluster nodes. We consider three case studies, including heat transfer, linear elasticity, as well as non-linear flow in heterogeneous media. The presented package generates output suitable for interfacing with Gnuplot and ParaView visualization software. The exemplary simulations show near perfect scalability on Gilbert shared-memory node with four Intel® Xeon® CPU E7-4860 processors, each possessing 10 physical cores (for a total of 40 cores).

Program summary

Program Title: IGA-ADS

Program Files doi: http://dx.doi.org/10.17632/pbpsyzyvfy.1

Licensing provisions: MIT license (MIT)

Programming language: C++

Nature of problem: Solving non-stationary problems in 1D, 2D and 3D

Solution method: L^2 projections with isogeometric finite element method

Additional comments including Restrictions and Unusual features: Due to nature of the ADS solver, using explicit Euler scheme is necessary. This imposes some restrictions on time step size required to maintain stability.

1. Introduction

The Alternating Directions Implicit (ADI) method was originally introduced in [1–3] to solve parabolic and hyperbolic partial differential equations using finite difference methods. The method has been recently generalized for the fast computation of L^2 projections with isogeometric finite element method (IGA-FEM), for regular geometries [4,5]. For non-regular geometries, the method still can be used as a preconditioner for iterative solvers [6].

The isogeometric finite element method originally introduced by [7] utilizes the B-spline basis functions for approximation of the solution fields. This guarantees higher continuity of the resulting solution fields. Our version of the solver works over the regular 1D, 2D or 3D cube shape domain. The isogeometric ADI method decomposes a 2D problem into two 1D problems with multiple right-hand sides, whose number corresponds to the number of B-splines basis functions utilized over the rows and columns of the mesh. Similarly, a 3D

☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author.
E-mail address: los@agh.edu.pl (M.M. Łoś).

http://dx.doi.org/10.1016/j.cpc.2017.02.023
0010-4655/

problem is decomposed into three 3D problems with multiple right-hand sides, and their number corresponds to the number of B-splines along x, y, or z directions, respectively.

The method is applicable as long as the mass matrix of the problem possesses a Kronecker product structure. This imposes certain restrictions on the problems being solved. In particular, arbitrary Dirichlet and zero Neumann boundary conditions are supported, but non-zero Neumann BC in general change the matrix structure and render the method inapplicable. Furthermore, for mixed boundary conditions the separation of boundary into Dirichlet and Neumann parts is not fully arbitrary—each side of the cube must be fully contained in one or the other.

The sequential version of our solver has been already used for solution of heat transfer [5], linear elasticity [5], non-linear flows in heterogeneous media [8], as well as for the simulations of the tumor growth [9].

In this work we present a parallel implementation of the isogeometric L^2 projection solver targeting a shared-memory Linux cluster node. The solver has been implemented using the GALOIS framework [10]. From our numerical experiments it implies that the solver exhibits near perfect parallel scalability on 40 cores Linux cluster node. Namely, the numerical experiments have been performed on the Gilbert shared-memory node with four Intel® Xeon® CPU E7-4860 processors, each possessing 10 physical cores (for a total of 40 cores).

The structure of the paper is the following. We start in Section 2 with introduction of the algorithm for solutions of time-dependent problems using the isogeometric L^2 projections. Section 3 delivers several examples for instantiation of the time-dependent problem, namely heat transfer, linear elasticity and non-linear flow in heterogeneous media. In Section 4 we discuss its parallelization with GALOIS framework. Section 5 describes the installation and usage of the software, as well as it provides examples on how to implement the three mentioned exemplary problems. In Section 6 we provide the parallel scalability experiments of our software. We conclude the paper in Section 7.

2. Application of the isogeometric L^2 projections for solution of non-stationary problems

In this section we derive the algorithm for solution of non-stationary problems using the L^2 projections with isogeometric finite element method. We shall demonstrate the general idea of the algorithm using as an example the following family of problems. We seek the solution $u : \Omega \to \mathbb{R}$ of the following equation:

$$\begin{cases} \dfrac{\partial u}{\partial t} = \mathcal{L}u + f(\mathbf{x}) & \text{on } \Omega \times [0,T] \\ u(\mathbf{x},t) = g(\mathbf{x}) & \text{on } \Gamma_D \times [0,T] \\ \nabla u \cdot \hat{\mathbf{n}} = 0 & \text{on } \Gamma_N \times [0,T] \\ u(\mathbf{x},0) = u_0(\mathbf{x}) & \text{on } \Omega \end{cases} \tag{1}$$

where $\Omega = [0,1]^3$, $\mathcal{L}$ is a second-order partial differential operator, $\hat{\mathbf{n}}$ is a normal vector of the domain boundary, T is a length of the time interval for the simulation, $\Gamma_D \cup \Gamma_N = \partial\,\Omega$ are disjoint subsets of the boundary corresponding to Dirichlet and Neumann boundary conditions and u_0 is an initial state.

2.1. Weak formulation

The corresponding weak formulation is obtained by multiplying (1) by test function $w \in H^1(\Omega)$, integrating by parts over Ω, and imposing the boundary conditions. The problem has the following form: find $u \in \mathcal{C}^1\left([0,T], H^1(\Omega)\right)$ such that for each $t \in [0,T]$

$$\left(\frac{\partial u}{\partial t}, w\right)_{L^2(\Omega)} = -b(u,w) + (f,w)_{L^2(\Omega)} \qquad \forall w \in H^1(\Omega) \tag{2}$$

where $(\cdot,\cdot)_{L^2(\Omega)}$ stands for the $L^2(\Omega)$ scalar product and b is an operator resulting from integrating by parts. For non-uniform Dirichlet conditions we can utilize the lift function technique (see [11]). Let $\tilde{u}$ be a function equal to $g(\mathbf{x})$ on Γ_D. Setting $u = u_0 + \tilde{u}$ reduces the above problem to a problem with uniform Dirichlet BC:

$$\left(\frac{\partial u_0}{\partial t}, w\right)_{L^2(\Omega)} = -\tilde{b}(u_0,w) + (f,w)_{L^2(\Omega)} \qquad \forall w \in H^1(\Omega) \tag{3}$$

where $\tilde{b}(u_0,w) = b(u_0+\tilde{u},w)$.

2.2. Discretization with B-spline basis functions

Restricting the problem to the subspace $V \subset H^1(\Omega)$ generated by standard tensor product of B-spline basis functions $\{B_I\}$ we obtain semi-discrete formulation of the problem: find $u \in \mathcal{C}^1([0,T], V)$ such that for each $t \in [0,T]$ Eq. (3) holds for all $w \in V$. Function u has form

$$u(\mathbf{x},t) = \sum_{i=1}^{n} U^i(t)\mathcal{B}_i(\mathbf{x}). \tag{4}$$

Since Eq. (2) is required to hold for $w \in V$, it is sufficient and necessary that it holds for elements of basis of V, as both sides are linear with respect to w, hence the semi-discrete problem is equivalent to the system

$$\sum_{i=1}^{n} \left(\mathcal{B}_i, \mathcal{B}_j\right)_{L^2(\Omega)} \frac{\mathrm{d}U^i}{\mathrm{d}t} = -b\left(\sum_{i=1}^{n} U^i\mathcal{B}_i, \mathcal{B}_j\right) + \left(f, \mathcal{B}_j\right)_{L^2(\Omega)} \quad j = 1,\ldots,n. \tag{5}$$

Let us define

$$\begin{aligned}\mathbf{M} &= \left[\left(\mathcal{B}_i, \mathcal{B}_j\right)_{L^2(\Omega)}\right]_i^j \\ \mathbf{u} &= \left(U^1, \ldots, U^n\right)^T \\ \mathbf{B}(\mathbf{u}) &= (b(u, \mathcal{B}_1), \ldots, b(u, \mathcal{B}_n))^T \\ \mathbf{h} &= \left((f, \mathcal{B}_1)_{L^2(\Omega)}, \ldots, (f, \mathcal{B}_n)_{L^2(\Omega)}\right)^T.\end{aligned} \tag{6}$$

Using this notation, Eq. (5) can be written as

$$\mathbf{M}\,\mathbf{u}' = -\mathbf{B}(\mathbf{u}) + \mathbf{h}. \tag{7}$$

Discretization of time with forward Euler method with time step Δt yields

$$\mathbf{M}\mathbf{u}_{k+1} = \mathbf{M}\mathbf{u}_k - \Delta t\,(\mathbf{B}(\mathbf{u}_k) - \mathbf{h})\,. \tag{8}$$

2.3. Solving the linear system

In the above system (8) matrix $\mathbf{M}$ is a Gram matrix of the basis, so solving it is equivalent to solving an L^2-projection problem. Furthermore, since we are using tensor product basis functions, $\mathbf{M}$ can be decomposed into a Kronecker product of Gram matrices of one-dimensional bases as follows. Let $(N_1^x, \ldots, N_n^x)$, $(N_1^y, \ldots, N_m^y)$ be two B-spline bases in 1D and let us denote by $\mathbf{M}^x$ and $\mathbf{M}^y$ their Gram matrices, respectively, so that

$$M_{ij}^x = \left(N_i^x, N_j^y\right)_{L^2(0,1)} \tag{9}$$

and similarly for $\mathbf{M}^y$. Two-dimensional tensor product B-spline basis is defined as

$$N_{ij}(x, y) = N_i^x(x)\,N_j^y(y). \tag{10}$$

Entries of its Gram matrix $\mathbf{M}$ are given by

$$\begin{aligned}M_{ij,kl} &= \left(N_{ij}, N_{kl}\right)_{L^2(\Omega)} = \int_\Omega N_{ij}(x, y)\,N_{kl}(x, y)\,\mathrm{d}\Omega \\ &= \int_\Omega N_i^x(x)N_j^y(y)\,N_k^x(x)N_l^y(y)\,\mathrm{d}\Omega \\ &= \int_0^1\int_0^1 N_i^x(x)N_k^x(x)\,N_j^y(y)N_l^y(y)\,\mathrm{d}x\,\mathrm{d}y \\ &= \left(\int_0^1 N_i^x(x)N_k^x(x)\,\mathrm{d}x\right)\left(\int_0^1 N_j^y(y)N_l^y(y)\,\mathrm{d}y\right) \\ &= \left(N_i^x, N_k^x\right)_{L^2(0,1)}\left(N_j^y, N_l^y\right)_{L^2(0,1)} \\ &= M_{ik}^x M_{jl}^y\end{aligned} \tag{11}$$

and so $\mathbf{M} = \mathbf{M}^x \otimes \mathbf{M}^y$. Analogous decomposition can be derived for higher-dimensional bases.

This tensor product structure allows us to solve systems of the form $\mathbf{M}\mathbf{x} = \mathbf{b}$ by solving multiple systems with matrices constituting the tensor product. These matrices are Gram matrices of one-dimensional B-spline basis and so easily invertible (banded), and it turns out this technique allows us to solve systems with matrix $\mathbf{M}$ in linear time.

Here we shall describe the algorithm in case where $\mathbf{M} = \mathbf{A} \otimes \mathbf{B}$, that is, for product of two matrices of sizes $n \times n$ and $m \times m$, respectively. The case with product of multiple matrices is a straightforward generalization. Using the definition of Kronecker product and denoting $\mathbf{x}^i = \left(x^{i1}, \ldots, x^{im}\right)^{\mathrm{T}}$, $\mathbf{b}^i = \left(b^{i1}, \ldots, b^{im}\right)^{\mathrm{T}}$ we can write equation $\mathbf{M}\mathbf{x} = \mathbf{b}$ as

$$\begin{bmatrix}\mathbf{B}A_1^1 & \mathbf{B}A_2^1 & \cdots & \mathbf{B}A_n^1 \\ \mathbf{B}A_1^2 & \mathbf{B}A_2^2 & \cdots & \mathbf{B}A_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{B}A_1^n & \mathbf{B}A_2^n & \cdots & \mathbf{B}A_n^n\end{bmatrix}\begin{bmatrix}\mathbf{x}^1 \\ \mathbf{x}^2 \\ \vdots \\ \mathbf{x}^n\end{bmatrix} = \begin{bmatrix}\mathbf{b}^1 \\ \mathbf{b}^2 \\ \vdots \\ \mathbf{b}^m\end{bmatrix}. \tag{12}$$

Writing explicitly the resulting equations we obtain

$$\begin{aligned}&\mathbf{B}\left(A_1^1\mathbf{x}^1 + A_2^1\mathbf{x}^2 + \cdots + A_n^1\mathbf{x}^n\right) = \mathbf{b}^1 \\ &\mathbf{B}\left(A_1^2\mathbf{x}^1 + A_2^2\mathbf{x}^2 + \cdots + A_n^2\mathbf{x}^n\right) = \mathbf{b}^2 \\ &\qquad\vdots \qquad\quad \vdots \qquad\quad \vdots \qquad\quad \vdots \\ &\mathbf{B}\left(A_1^n\mathbf{x}^1 + A_2^n\mathbf{x}^2 + \cdots + A_n^n\mathbf{x}^n\right) = \mathbf{b}^n.\end{aligned} \tag{13}$$

Let $\mathbf{y}^i = \mathbf{B}^{-1}\mathbf{b}^i$. Multiplying each of the above equations by $\mathbf{B}^{-1}$ we obtain new family of right-hand sides. By considering each component of $\mathbf{x}^i$ and $\mathbf{y}^i$ separately, we can see the resulting system is equivalent to family of systems of the form

$$\begin{aligned} A_1^1 x^{1i} + A_2^1 x^{2i} + \cdots + A_n^1 x^{ni} &= y^{1i} \\ A_1^2 x^{1i} + A_2^2 x^{2i} + \cdots + A_n^2 x^{ni} &= y^{2i} \\ \vdots \quad\quad \vdots \quad\quad \vdots \quad\quad \vdots & \\ A_1^n x^{1i} + A_2^n x^{2i} + \cdots + A_n^n x^{ni} &= y^{ni} \end{aligned} \tag{14}$$

for each $i = 1, \ldots, m$. This, together with Eq. (13) can be written as

$$\begin{bmatrix} B_1^1 & B_2^1 & \cdots & B_m^1 \\ B_1^2 & B_2^2 & \cdots & B_m^2 \\ \vdots & \vdots & \ddots & \vdots \\ B_1^m & B_2^m & \cdots & B_m^m \end{bmatrix} \begin{bmatrix} y^{11} & y^{21} & \cdots & y^{n1} \\ y^{12} & y^{22} & \cdots & y^{n1} \\ \vdots & \vdots & \ddots & \vdots \\ y^{1m} & y^{2m} & \cdots & y^{nm} \end{bmatrix} = \begin{bmatrix} b^{11} & b^{21} & \cdots & b^{n1} \\ b^{12} & b^{22} & \cdots & b^{n2} \\ \vdots & \vdots & \ddots & \vdots \\ b^{1m} & b^{2m} & \cdots & b^{nm} \end{bmatrix}$$
$$\begin{bmatrix} A_1^1 & A_2^1 & \cdots & A_n^1 \\ A_1^2 & A_2^2 & \cdots & A_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ A_1^n & A_2^n & \cdots & A_n^n \end{bmatrix} \begin{bmatrix} x^{11} & x^{12} & \cdots & x^{1m} \\ x^{21} & x^{22} & \cdots & x^{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x^{n1} & x^{n2} & \cdots & x^{nm} \end{bmatrix} = \begin{bmatrix} y^{11} & y^{12} & \cdots & y^{1m} \\ y^{21} & y^{22} & \cdots & y^{2m} \\ \vdots & \vdots & \ddots & \vdots \\ y^{n1} & y^{n2} & \cdots & y^{nm} \end{bmatrix}. \tag{15}$$

As a result, the problem of solving a system of linear equation with matrix $\mathbf{M}$ has been reduced to problem of solving m systems with matrix $\mathbf{A}$ and n systems with matrix $\mathbf{B}$. Cost of solving each is linear, so the total cost of the computation is $m\,\mathcal{O}(n) + n\,\mathcal{O}(m) = \mathcal{O}(nm)$, that is, it scales linearly with number of unknowns in the original system. The same holds for generalization of the method to product of more than two matrices.

2.4. Algorithm for solution of time-dependent problems with isogeometric L^2 projections

The above iteration scheme is straightforward to translate to a simulation algorithm. High-level structure of the simulation is presented as Algorithm 1. In short, the computation comprises some one-time initialization work (array allocation, computing initial state, etc.) and a number of time steps, each consisting of computing the right-hand side vector from Eq. (8) and solving the resulting system using Alternating Direction Solver.

Input:
- n – number of elements along each dimension
- p – order of basis
- Δt – time step size
- T_{max} – number of time steps

Data:
- t – time
- U – current solution
- U_{prev} – solution from the previous time step

```
1 PrepareSimulation;
2 for i ← 1 to T_max do
3 |   t ← t + Δt ;
4 |   U_prev ← U ;
5 |   ComputeRHS(U, U_prev, t, Δt) ;
6 |   Solve(U, n, p) ;
7 end
```

Algorithm 1: Main loop.

The most important part of the implementation is computing the right-hand side vector at each time step. Integrals involved are computed using Gaussian quadrature on each element individually. Let $\mathcal{I}$ denote the set of valid multi-indices indexing tensor product B-spline basis. Thanks to the locality of support of one-dimensional B-spline basis functions, for any $I \in \mathcal{I}$ there is only a small number of elements E such that support of $\mathcal{B}_I$ overlaps E. For given element E let $\mathcal{I}(E)$ be set of multi-indices I such that $\mathcal{B}_I$ is not identically zero on E. Algorithm used to compute the right-hand side vector is presented as Algorithm 2.

Due to using a fast linear solver, right-hand side evaluation is the most time-consuming part of the computation. Let N denote the number of mesh elements and d be the dimension of the problem. Outermost loop from line 3 of Algorithm 2 is executed N times. For B-spline basis of order p we use Gaussian quadratures of order $p+1$ to ensure exact evaluation of integrals involving products of two B-splines, so the middle loop from line 4 of Algorithm 2 is executed $(p+1)^d$ times. Finally, in one dimension each element intersects supports of $p+1$ basis B-splines, so size of $\mathcal{I}(E)$ is always $(p+1)^d$ since we use a tensor product basis. Thus, the internal loops from lines 9 and 13 of Algorithm 2 are executed $(p+1)^d$ times. Thus, time complexity of the RHS computation can be estimated by $\mathcal{O}\left(Np^{2d}\right)$.

3. Limitations

The essential requirement for using Alternating Direction Solver algorithm is that the matrix of the linear problem to be solved has a tensor product structure. This imposes certain restrictions on kinds of problems and techniques that can be directly implemented using the framework.

$w \leftarrow$ weights of Gaussian quadrature ;
$X \leftarrow$ points of Gaussian quadrature on [0, 1];
for *each element* $E = \left[\xi_{l_x}, \xi_{l_x+1}\right] \times \left[\xi_{l_y}, \xi_{l_y+1}\right] \times \left[\xi_{l_z}, \xi_{l_z+1}\right]$ **do**
  for *each quadrature point* $\boldsymbol{\xi} = \left(X_{k_x}, X_{k_y}, X_{k_z}\right)$ **do**
    $\mathbf{x} \leftarrow \Psi_E(\boldsymbol{\xi})$;
    $W \leftarrow w_{k_x} w_{k_y} w_{k_z}$;
    $F \leftarrow h(\mathbf{x},\ t + \Delta t)$;
    $u, Du \leftarrow 0$;
    for $I \in \mathcal{I}(E)$ **do**
      $u \leftarrow u + U_I^{(t)} \mathcal{B}_I(\boldsymbol{\xi})$;
      $Du \leftarrow Du + U_I^{(t)} \nabla \mathcal{B}_I(\boldsymbol{\xi})$;
    end
    for $I \in \mathcal{I}(E)$ **do**
      $v \leftarrow \mathcal{B}_I(\boldsymbol{\xi})$;
      $Dv \leftarrow \nabla \mathcal{B}_I(\boldsymbol{\xi})$;
      $r \leftarrow vF - \kappa\,(Du,\ Dv)$;
      $U_I^{(t+1)} \leftarrow U_I^{(t+1)} + W\,|E|\,(u\,v + r\,\Delta t)$
    end
  end
end

Algorithm 2: ComputeRHS($U^{(t+1)}$, $U^{(t)}$, t, Δt)—sequential version.

3.1. Time-stepping schemes

As demonstrated in Section 2.2 for the forward Euler scheme, explicit schemes, in general, can be easily used with ADS, since the left-hand side of the step formula is essentially unaffected by choice of a particular scheme. Implicit schemes, however, in general, do influence the matrix on the left-hand side and so may be incompatible with ADS if the resulting linear system cannot be decomposed as a tensor product. This is often the case even for simple methods and problems—for example, using backward Euler method to solve the heat transfer problem leads to a matrix that lacks the suitable tensor product structure. A potential solution may be to use specialized implicit schemes that preserve the tensor product structure of the matrices if the implicit method is necessary to maintain stability.

3.2. More complex geometries

Suppose we want to solve a problem on a non-cubic domain $\Omega \subset \mathbb{R}^2$. We can use isoparametric elements and map the unit cube to Ω using a function $\phi : [0, 1]^2 \to \Omega$. Then we can construct basis functions $\tilde{N}_{ij}$ using N_{ij} defined in Section 2.3 as

$$\tilde{N}_{ij} = N_{ij} \circ \phi^{-1}. \tag{16}$$

The Gram matrix of the new basis is given by

$$\begin{aligned} M_{ij,kl} &= \left(\tilde{N}_{ij}, \tilde{N}_{kl}\right)_{L^2(\Omega)} = \int_\Omega \tilde{N}_{ij}(x,y)\,\tilde{N}_{kl}(x,y)\,\mathrm{d}\Omega \\ &= \int_\Omega \tilde{N}_{ij}(x,y)\,\tilde{N}_{kl}(x,y)\,\mathrm{d}\Omega \\ &= \int_{[0,1]^2} \left(\tilde{N}_{ij}\,\tilde{N}_{kl}\right) \circ \phi\ \left|J_\phi\right|\ \mathrm{d}x \\ &= \int_{[0,1]^2} N_{ij}\,N_{kl}\,\left|J_\phi\right|\ \mathrm{d}x \\ &= \int_{[0,1]^2} N_i(x) N_k(x)\,N_j(y) N_l(y)\,\left|J_\phi\right|\ \mathrm{d}x. \end{aligned} \tag{17}$$

This can be decomposed as a product of two one-dimensional integrals as in Section 2.3 only if the Jacobian J_ϕ is a product of functions depending on each variable separately. For example, Jacobian of spherical coordinates mapping $J = r^2 \sin\theta$ is a product of r^2 and $\sin\theta$, and so the resulting Gram matrix is a product of two one-dimensional ones and ADS can be applied. In many other cases, it may not be possible.

In the case of complicated geometry for some class of PDE, it is possible to explore the possibility of application of the Finite Cell Method [12] where the indicator function $\alpha(x)$ is incorporated into the weak formulation, equal to 1 in the physical domain embedded inside the computational cube, and equal to $\epsilon \ll 1$ outside the physical domain. We will try this technique in our future research.

3.3. Boundary conditions

Both Dirichlet and Neumann boundary conditions may be imposed, as long as the alteration of the left-hand side matrix introduced by Dirichlet BC preserves the tensor product structure. In particular, this is the case as long as the Dirichlet BC are imposed on the part of the

boundary that can be represented by a union of its faces. Imposing Dirichlet boundary condition on a single face can be done by modifying only the one-dimensional matrix corresponding to the direction perpendicular to this face, so the full matrix retains the tensor product structure.

4. Instantiation for different non-stationary PDEs

The above general formulation encompasses a variety of practical problems. This section briefly describes few such examples that will later be implemented in the framework to present its usage.

4.1. Heat equation

The heat transfer problem is given by the following equation:

$$\begin{cases} \dfrac{\partial u}{\partial t} = \Delta u + f(\mathbf{x}) & \text{on } \Omega \times [0, T] \\ \nabla u \cdot \hat{\mathbf{n}} = 0 & \text{on } \partial\,\Omega \times [0, T] \\ u(\mathbf{x}, 0) = u_0(\mathbf{x}) & \text{on } \Omega \end{cases} \tag{18}$$

where f is the forcing term. For simplicity we assume zero Neumann boundary conditions. In this case $\mathcal{L} = \Delta$.

4.2. Flow in heterogeneous medium

Following [8] we can define problem of flow in heterogeneous medium by the following equation:

$$\begin{cases} \dfrac{\partial u}{\partial t} = \nabla \cdot (\kappa(\mathbf{x}; u)\nabla u) + h(\mathbf{x}) & \text{on } \Omega \times [0, T] \\ \nabla u \cdot \hat{\mathbf{n}} = 0 & \text{on } \partial\,\Omega \times [0, T] \\ u(\mathbf{x}, 0) = u_0(\mathbf{x}) & \text{on } \Omega \end{cases} \tag{19}$$

where u is the pressure, h is a forcing term, $\kappa(\mathbf{x}; u) = K_q(\mathbf{x})e^{\mu u}$ is a permeability of the medium, K_q is material data and u_0 is an initial state. In this case $\mathcal{L} = \nabla \cdot (\kappa(\mathbf{x}, u)\nabla u)$.

4.3. Linear elasticity

We consider linear elasticity problem given by

$$\begin{cases} \rho\, \partial_{tt}\mathbf{u} = \nabla \cdot \boldsymbol{\sigma} + \mathbf{F} & \text{on } \Omega \times [0, T] \\ \boldsymbol{u}(x, 0) = u_0 & \text{for } x \in \Omega \\ \boldsymbol{\sigma} \cdot \hat{\mathbf{n}} = 0 & \text{on } \partial\,\Omega \end{cases} \tag{20}$$

where $\Omega = [0, 1]^3$ is a unit cube, $\mathbf{u}$ is a 3-dimensional displacement vector to be calculated, ρ is material density, $\mathbf{F}$ is the applied external force, and $\boldsymbol{\sigma}$ is rank-2 stress tensor, given by

$$\sigma_{ij} = c_{ijkl}\epsilon_{lk}, \qquad \epsilon_{ij} = \frac{1}{2}\left(\partial_j u_i + \partial_i u_j\right) \tag{21}$$

and $\mathbf{c}$ is a rank-4 elasticity tensor. The above second-order system can be converted to system of 6 first-order equations by introducing additional variable $\mathbf{v} = \partial_t \mathbf{u}$:

$$\begin{cases} \partial_t \mathbf{u} = \mathbf{v} \\ \partial_t \mathbf{v} = \rho^{-1}(\nabla \cdot \boldsymbol{\sigma} + \mathbf{F}). \end{cases} \tag{22}$$

This allows to express the problem using a variant of the general formulation with system of equations by setting

$$\begin{cases} \mathcal{L}(\mathbf{u}, \mathbf{v}) = \left(\mathbf{v}, \rho^{-1}\nabla \cdot \boldsymbol{\sigma}\right) \\ f(\mathbf{x}) = \left(0, \rho^{-1}\mathbf{F}(\mathbf{x})\right). \end{cases} \tag{23}$$

5. Parallelization of the isogeometric L^2 projections with GALOIS

The idea of parallelizing Algorithm 2 is relatively simple due to its structure. Integrals are calculated on each element and as a result, small fragment of the right-hand side vector is updated. Computations on each element are independent of each other, so they can be executed in arbitrary order. This suggests treating them as a set of tasks that can be individually and independently assigned to available processors. Such interpretation allows to directly use tools for loop parallelization. In our implementation we utilized GALOIS framework [10] and its `Galois::for_each` template function that executes specified action for all elements of the supplied sequence, automatically distributing the work over available worker threads.

Since tasks corresponding to all the elements modify a common data structure – array representing right-hand side vector – a synchronization is required to ensure correctness of the implementation. The most obvious way to achieve that is to protect the update operation (line 17 in Algorithm 2) using a mutex, so that only one task at a time can update the array. That, however, incurs a significant synchronization overhead that severely limits scalability, since each task updates the right-hand side multiple times, thus requires many mutex manipulation operations. This is indirectly caused by the fact that local computation and external data structure updates are heavily

interweaved. More efficient approach is to strictly separate computation and updates, storing the information necessary for the full update locally and executing it in a single synchronized block afterwards. This modification greatly reduces synchronization overhead and yields significantly better scalability. This is possible due to the fact that task associated with a given element updates only a small portion of the right-hand side array ($(p+1)^3$ entries), thus a small local buffer corresponding to this part is enough to store all the changes introduced by the task. The pseudocode of this solution is displayed as Algorithm 3 (changes in comparison to sequential version highlighted).

```
1  w ← weights of Gaussian quadrature ;
2  X ← points of Gaussian quadrature on [0, 1];
3  for each element E = [ξ_{l_x}, ξ_{l_x+1}] × [ξ_{l_y}, ξ_{l_y+1}] × [ξ_{l_z}, ξ_{l_z+1}] in parallel do
4      U^loc ← 0 ;
5      for each quadrature point ξ = (X_{k_x}, X_{k_y}, X_{k_z}) do
6          x ← Ψ_E (ξ) ;
7          W ← w_{k_x} w_{k_y} w_{k_z} ;
8          F ← h (x, t + Δt) ;
9          u, Du ← 0 ;
10         for I ∈ I(E) do
11             u ← u + U_I^(t) B_I(ξ) ;
12             Du ← Du + U_I^(t) ∇B_I(ξ) ;
13         end
14         for I ∈ I(E) do
15             v ← B_I(ξ) ;
16             Dv ← ∇B_I(ξ) ;
17             r ← vF − κ (Du, Dv) ;
18             U_I^loc ← U_I^loc + W |E| (u v + r Δt)
19         end
20     end
21     synchronized
22         for I ∈ I(E) do
23             U_I^(t+1) ← U_I^(t+1) + U_I^loc
24         end
25     end
26 end
```

Algorithm 3: ComputeRHS($U^{(t+1)}$, $U^{(t)}$, t, Δt)—parallel version.

6. Validation

To verify correctness of the code we used two-dimensional heat transfer problem with zero Dirichlet boundary conditions given by

$$\begin{cases} \dfrac{\partial u}{\partial t} = \Delta u & \text{on } \Omega \times [0, T] \\ u(x, y;\ 0) = \sin(\pi x)\sin(\pi y) & \text{on } \Omega . \end{cases} \tag{24}$$

The above problem possesses an analytical solution

$$u(x_1, x_2;\ t) = e^{-2\pi^2 t} \sin(\pi x) \sin(\pi y) . \tag{25}$$

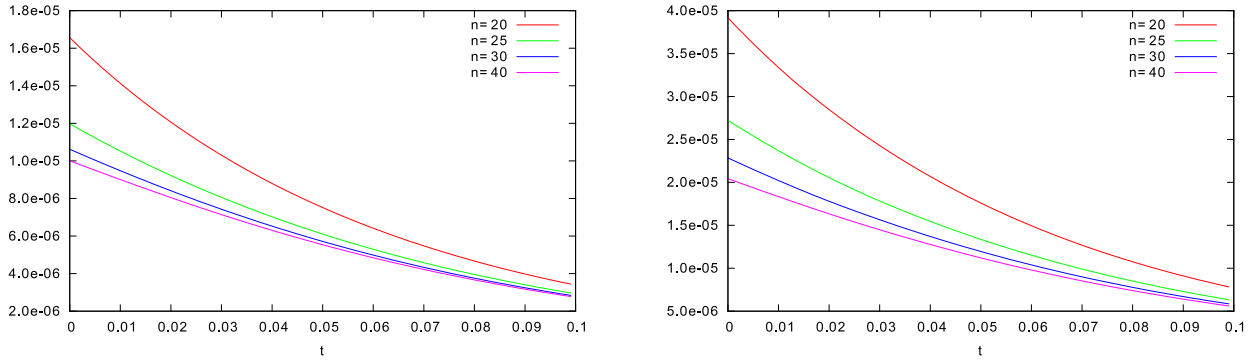
We carried out the simulation using quadratic B-splines, $n \times n$ mesh for $n = 20,\ 25,\ 30,\ 40$, time step $\Delta t = 10^{-6}$ and 100,000 steps ($T = 0.1$) and computed errors in L^2 and L^∞ (maximum) norms. The results are presented in Fig. 1. Relatively small errors indicate that the code works correctly.

7. Comparison with existing packages

7.1. PetIGA

PetIGA [13] is an IGA implementation in C and Fortran, heavily based on PETSc [14]—framework for the scalable parallel solution of PDEs. It is targeted towards high-performance scientific computation and provides excellent scalability on both shared and distributed memory architectures. Through PETSc, it offers a wide variety of linear solvers.

Our package differs from PetIGA in the following aspects. First, the PetIGA does not provide the implementation of the alternating direction solver, though it is possible to implement ADS solver in PETSc. Second, our package targets multi-core Linux cluster nodes, with the parallelism supported by GALOIS framework, while PetIGA targets distributed memory platforms with parallelism supported by MPI. Third, we have provided simple interfaces to several three-dimensional visualization packages, including Gnuplot and ParaView. Thus, we believe that our package is an attractive alternative for the solution of time-dependent problems suitable for the alternating direction solver, in shared memory environment.



(a) L^2. (b) L^∞.

Fig. 1. Errors for the test heat transfer problem instance.

7.2. MIGFEM

MIGFEM is a Matlab IGA implementation described in [15]. It supports NURBS, h, p and k-refinements. The framework does not seem to put much emphasis on performance. In particular, the paper describing the implementation does not mention parallelization. Inspection of the example programs contained in the package suggests that assembling the right-hand side is sequential. Furthermore, the package uses built-in Matlab solver.

7.3. GeoPDEs

GeoPDEs 3.0 [16] is a recent IGA implementation written in Matlab/Octave. It handles arbitrary multipatch geometries defined using NURBS in a dimension-independent way and provides output to VTK. The report describing the implementation neither discusses parallelization nor the method of solving linear systems that arise from using IGA. Inspection of the source code suggests the package uses built-in Matlab solver.

7.4. G+Smo

G+Smo [17] is a relatively new IGA library implemented in modern C++. It offers lots of interesting features, including output to multiple formats (including VTK), support for complex geometries, NURBS, and few iterative solvers. In contrast to our code, G+Smo does not provide a linear time direct solver. Furthermore, the library documentation does not seem to mention parallelization at all.

8. Installation and execution of different non-stationary simulations

In this section we will show how to use the code by solving example problems described in Section 4.

8.1. Obtaining the code

The code is available from a publicly accessible Github repository. Current version can be obtained by executing

```
$ git clone https://github.com/marcinlos/iga-ads
```

8.2. Compilation and dependencies

IGA-ADS requires working installations of the following libraries:

- LAPACK [18]
- Boost [19], version 1.58 or higher[1]
- Galois [20], version 2.2.1.

Furthermore, compiling and running unit tests require libunittest [21]. The code employs C++ 14 constructs, so a reasonably up-to-date C++ compiler is desirable. During development we used GCC 5.3.1 [22]. As a build tool, we use CMake [23].

To build the code, execute

```
$ cmake . && make
```

As a result, executable file `${PROJECT_DIR}/bin/ads` should be created. To compile with Galois support, add `-DUSE_GALOIS=ON` to `cmake` arguments (this requires Galois to be installed— see Section 8.3). To compile unit tests, add `-DCOMPILE_TESTS=ON`.

[1] Not tested with earlier versions.

8.3. Installing Galois framework

The source of Galois can be downloaded from the project webpage (http://iss.ices.utexas.edu/?p=projects/galois/download). After unpacking the code, execute the following commands in the Galois main directory:

```
$ cd build
$ mkdir release
$ cd release
$ cmake -DSKIP_COMPILE_APPS=ON ../..
$ make && make install
```

This results in compiling and installing the library and header files in a default location. For details, please refer to CMake documentation [23].

8.4. Implementing heat equation problem

Easiest way to create a new simulation is to inherit from a suitable base class `simulation_Nd`, where N is the dimension of the problem. To use them, we need to include `"ads/simulation.hpp"` header file. Here we describe two-dimensional version, 3D version is almost identical.

```
#include "ads/simulation.hpp"
class heat : public ads::simulation_2d {
```

We need buffers for right-hand side and solution in the current time step, and for the solution from the previous time step. Class `simulation_2d` provides helpful type definition – `vector_type`, which denotes a tensor type of suitable rank.

```
    vector_type u, u_prev;
```

Constructor of `simulation_2d` accepts a configuration object `config_2d`, that carries all the problem-independent parameters of the simulation (order of basis functions, number of elements in each dimension, etc.)

```
    heat(const config_2d& config)
    : ads::simulation_2d{ config }
```

We also need to initialize the buffers. Their constructors require `std::array` object containing sizes for each dimension. This information is computed from the config object by `simulation_2d` and can be obtained `shape()` method:

```
    , u{ shape() }
    , u_prev{ shape() }
    { }
```

Before we start the simulation, we want to provide some initial state. This is best done by overriding `before()` method from the base class:

```
void before() override {
    prepare_matrices();
    auto init = [this](double x, double y) { return /* ... */ };
    projection(u, init);
    solve(u);
}
```

Call to `prepare_matrices()` computes and factorizes Gram matrices of the one-dimensional B-spline bases. It can be overridden, e.g. if we want to impose Dirichlet conditions on some edges. Projection of the initial state is computed by `projection(u, init)`. Call to `solve` ensures that `u` contains coefficients of L^2-projection of the initial state on the tensor basis.

Before each step of the simulation we need to swap current and previous solutions (`u` and `u_prev`) and clear the `u`, since we will store RHS vector there. To do this, we can override `before_step` method from the base class:

```
void before_step(int /*iter*/, double /*t*/) override {
    using std::swap;
    swap(u, u_prev);
    zero(u);
}
```

Simulation step consists of computing RHS and solving the resulting linear system:

```
void step(int /*iter*/, double /*t*/) override {
    for (auto e : elements()) {
        double J = jacobian(e);
        for (auto q : quad_points()) {
            double w = weigth(q);
            value_type h = eval_fun(u_prev, e, q);
            for (auto a : dofs_on_element(e)) {
                value_type v = eval_basis(e, q, a);

                double val = h.val * v.val - steps.dt * grad_dot(h, v);
                u(a[0], a[1]) += val * w * J;
            }
        }
    }
    solve(u);
}
```

Note that computation of the RHS closely resembles pseudocode of Algorithm 2.

8.5. Parallelization

In order to create a parallel version of the code using Galois, we need to include header "ads/executor/galois.hpp" and create an executor object as a class field, specifying desired number of threads:

```
galois_executor executor{number_of_threads};
```

Instead of regular loop over the elements we use a parallel loop – `executor.for_each` and update a local (element) right-hand side vector, as in Algorithm 3. Updating the global RHS can be done using an auxiliary function `update_global_rhs`. To avoid race conditions, it must be called inside a synchronized block, which is achieved by using a function supplied by the executor.

```
executor.for_each(elements(), [&](index_type e) {
    auto U = element_rhs();

    double J = jacobian(e);
    for (auto q : quad_points()) {
        double w = weigth(q);
        value_type u = eval_fun(u_prev, e, q);
        for (auto a : dofs_on_element(e)) {
            value_type v = eval_basis(e, q, a);

            double gradient_prod = grad_dot(u, v);
            double val = u.val * v.val - steps.dt * gradient_prod;

            auto aa = dof_global_to_local(e, a);
            U(aa[0], aa[1]) += val * w * J;
        }
    }
    executor.synchronized([&]{ update_global_rhs(rhs, U, e); });
});
```

8.6. Generating output

The framework provides a way to output the results produced during the simulation to files. To use it, we need to include the appropriate header:

```
#include "ads/output_manager.hpp"
```

and create an instance of `output_manager` parameterized with the problem dimension (in this case, the dimension is 2). Only one object is needed so that it can be a field in the simulation class.

```
output_manager<2> output;
```

Initialization of the output manager needs to be performed in the constructor—we add the following line to the constructor's initializer list:

```
    , output{ x.B, y.B, 200 }
    { }
```

In general, solution u is saved as a list of its values in points on a regular grid—size of the grid in each dimension is determined by the last parameter of the `output_manager`'s constructor (so in this example, the solution is evaluated on a grid of 200×200 points). The B-spline basis functions passed as arguments determine range in each direction. The exact format of an output file is determined by the dimension of the solution—for 2D problems the solution is written as a list of lines of the form

```
x  y  u(x,y)
```

that can be plotted e.g. using Gnuplot. For 3D the results are saved as XML VTK file of `ImageData` type (see [24]).

If we want to save the initial state, we can add the following line at the end of `before()` method:

```
output.to_file(u, "init.data");
```

As a result, the solution before the first time step will be written to `init.data`.

Suppose now we want to generate output every 100 steps and save it to `out_`$<iter>$`.data`, where $<iter>$ denotes the number of the time step. To achieve this, we can override `after_step` method that is called after each time step.

```
void after_step(int iter, double /*t*/) override {
    if (iter % 100 == 0) {
        output.to_file(u, "out_%d.data", iter);
    }
}
```

Note that using this overloaded version of `to_file` method we can use `printf`-like format syntax and avoid building the file name by hand.

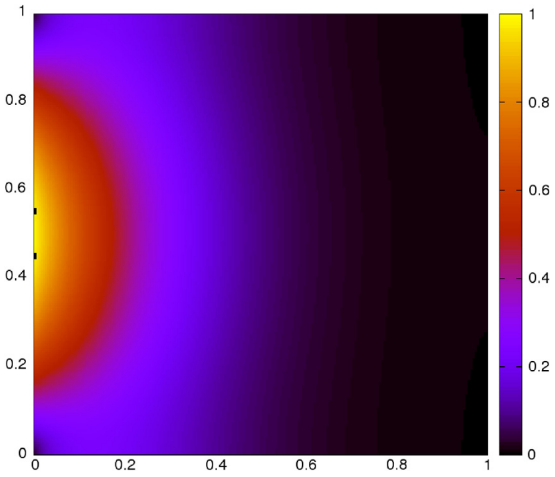


Fig. 2. Example Gnuplot output for heat transfer problem.

8.7. Running the simulation

Suppose we want to run 10,000 iterations of the above simulation using B-splines of order $p = 2$, grid of 40×40 elements and time step $\Delta t = 10^{-5}$. To do this, we need to create a configuration object describing the B-spline basis:

```
ads::dim_config dim{ 2, 40 };
```

and time steps:

```
timesteps_config steps{ 10000, 1e-5 };
```

Then, we compose these to create full configuration[2]:

```
ads::config_2d c{ dim, dim, steps, 1 };
```

Having done that, we can create and run the simulation:

```
heat sim{ c };
sim.run();
```

8.8. Visualizing the output

Running the simulation generates solution files `init.data` and `out_0.data`, `out_100.data`, ..., `out_9900.data`. The solutions can be visualized using Gnuplot. The following Gnuplot commands:

```
set view map
set xrange [0:1]
set yrange [0:1]
set cbrange [0:1]
plot 'out_5000.data' with image
```

produce the plot presented in Fig. 2 (solution after time step 5000). For more details on working with Gnuplot, please refer to the Gnuplot documentation [25].

8.9. Other features

The above code can be easily extended to handle nontrivial (e.g. non-homogeneous Dirichlet) boundary conditions. For more examples see contents of `problems/` (e.g. `heat/heat_2d.hpp` for a more elaborate heat equation code).

8.10. Non-linear flow problem

Code for non-linear flow in heterogeneous medium is similar to the heat problem code. The main simulation class is derived from `ads::simulation_3d`:

```
#include "ads/simulation.hpp"
class flow : public ads::simulation_3d {
  vector_type u, u_prev;
```

[2] The last argument – 1 – is number of required B-spline basis derivatives. Our computation uses gradients of basis functions, thus we need first order derivatives.

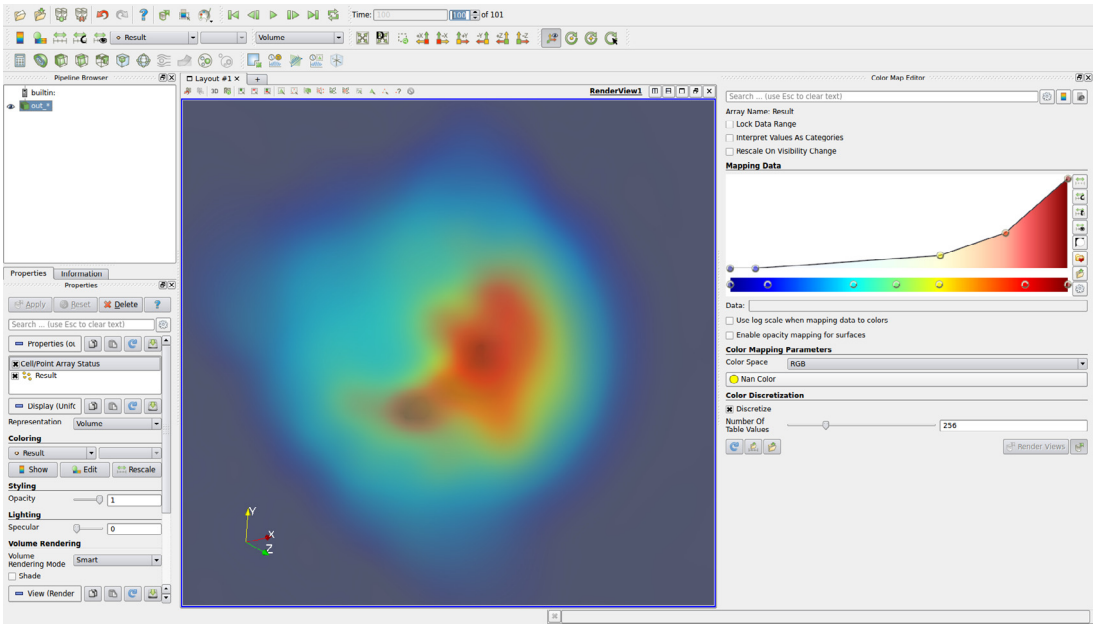


Fig. 3. Visualizing flow problem solutions with ParaView.

The only meaningful difference in the rest of the code is the computation of right-hand side. The body of element loop now looks as follows:

```
auto U = element_rhs();
double J = jacobian(e);
for (auto q : quad_points()) {
    auto x = point(e, q);
    double w = weigth(q);
    value_type u = eval_fun(u_prev, e, q);
    for (auto a : dofs_on_element(e)) {
        value_type v = eval_basis(e, q, a);

        double k = permeability(x, u.val);
        double gradient_prod = grad_dot(u, v);
        double h = forcing(x, t);
        double lhs = - k * gradient_prod + h * v.val;
        double val = u.val * v.val + steps.dt * lhs;

        auto aa = dof_global_to_local(e, a);
        U(aa[0], aa[1], aa[2]) += val * w * J;
    }
}
executor.synchronized([&]{ update_global_rhs(rhs, U, e); });
```

Functions `permeability` and `forcing` are problem parameters and can be defined arbitrarily. See `problems/flow` for a more extensive example code involving storing precomputed terrain permeability in a look-up array.

As for the heat transfer problem, we can output solutions using `output_manager`, this time using 3D version.

```
output_manager<3> output;
```

The code is the same as for the two-dimensional problems, except that we may want to give the output files `.vti` extension so that ParaView can easily recognize them.

Running the example code int `problems/flow` produces files `out_0.vti`, `out_100.vti`, ..., `out_9900.vti`. Solutions can be visualized using ParaView [26]. The files can be loaded as a group (*File* → *Open*). After clicking *Apply* button on the left pane, we can change the `Representation` from *Outline* to *Volume* and see the initial state. Adjusting color scales and opacity we can produce visualization as in Fig. 3. Sequence of images for all the time steps can be produced using *File* → *Save animation...*. For more details on working with ParaView please refer to the ParaView Guide [26].

8.11. Linear elasticity problem

Discretizing linear elasticity problem formulation (22) using explicit Newmark's scheme (with $\beta = \gamma = 0$) yields

$$\begin{cases} u_i^{(t+1)} = u_i^{(t)} + \Delta t\, v_i^{(t)} + \dfrac{\Delta t^2}{2} a_i^{(t)} \\ v_i^{(t+1)} = v_i^{(t)} + \Delta t\, a_i^{(t)} \\ a_i^{(t)} = \dfrac{1}{\rho}\left(\sigma_{ij,j} + F_i\right). \end{cases} \tag{26}$$

Since this time our problem is 3D, we can derive the simulation class from `ads::simulation_3d`. We are dealing with a system of equation, so for convenience let us define a structure representing a full solution in a single time step.

```
struct state {
    vector_type ux, uy, uz;
    vector_type vx, vy, vz;
    state(std::array<std::size_t, 3> shape)
    : ux{ shape }, uy{ shape }, uz{ shape }
    , vx{ shape }, vy{ shape }, vz{ shape }
    { }
};
```

Furthermore, it is convenient to define an auxiliary function executing specified action on all the components of the solution.

```
template <typename Fun>
void for_all(state& s, Fun fun) {
    fun(s.ux);
    fun(s.uy);
    fun(s.uz);
    fun(s.vx);
    fun(s.vy);
    fun(s.vz);
}
```

We need to store current and previous states, so we define two class fields

```
state now, prev;
```

Overall structure of the code is similar to the heat equation. In each time step we first need to swap previous and current states and zero the right-hand side vector to be calculated:

```
void before_step(int /*iter*/, double /*t*/) override {
    using std::swap;
    swap(now, prev);
  for_all(now, [](vector_type& a) { zero(a); });
}
```

Then, local contribution on each element needs to be calculated and

```
void step(int /*iter*/, double t) override {
    executor.for_each(elements(), [&](index_type e) {
        auto local = local_contribution(e, t);
```

used to update the global RHS vector.

```
        executor.synchronized([&] {
            apply_local_contribution(local, e);
        });
    });
```

Finally, systems for all the components need to be solved.

```
    for_all(now, [this](vector_type& a) { solve(a); });
}
```

Function `apply_local_contribution` simply calls `update_global_rhs` for each component:

```
void apply_local_contribution(const state& loc, index_type e) {
    update_global_rhs(now.ux, loc.ux, e);
    update_global_rhs(now.uy, loc.uy, e);
    update_global_rhs(now.uz, loc.uz, e);
    update_global_rhs(now.vx, loc.vx, e);
    update_global_rhs(now.vy, loc.vy, e);
    update_global_rhs(now.vz, loc.vz, e);
}
```

The interesting part is computing the element's local contribution to the right-hand side. The structure of loops is analogous to the heat equation case. First we loop over quadrature points and evaluate all the components of solution from the previous time steps, along with the derivatives.

```
state local_contribution(index_type e, double t) const {
    auto loc = state{ local_shape() };

    double J = jacobian(e);
    for (auto q : quad_points()) {
        auto x = point(e, q);
        double w = weigth(q);
        value_type ux = eval_fun(prev.ux, e, q);
        value_type uy = eval_fun(prev.uy, e, q);
        value_type uz = eval_fun(prev.uz, e, q);
        value_type vx = eval_fun(prev.vx, e, q);
        value_type vy = eval_fun(prev.vy, e, q);
        value_type vz = eval_fun(prev.vz, e, q);
```

Then we need to compute the strain tensor ϵ

```
        using tensor = double[3][3];
        tensor eps = {
            {      ux.dx,       (ux.dy+uy.dx)/2, (ux.dz+uz.dx)/2 },
            { (ux.dy+uy.dx)/2,       uy.dy,       (uy.dz+uz.dy)/2 },
            { (ux.dz+uz.dx)/2, (uy.dz+uz.dy)/2,       uz.dz       }
        };
```

and the stress tensor $\boldsymbol{\sigma}$. In this example we use Lamé parameters μ, λ (homogeneous, isotropic material).

```
        tensor s{};
        double lambda = 1;
        double mi = 1;
        double tr = eps[0][0] + eps[1][1] + eps[2][2];

        for (int i = 0; i < 3; ++ i) {
            for (int j = 0; j < 3; ++ j) {
                s[i][j] = 2 * mi * eps[i][j];
            }
            s[i][i] += lambda * tr;
        }
```

At this point we also need to evaluate external force **F**

```
        point_type F = force(x, t);
```

Finally, we loop over basis functions defined over the element `e` and compute the integrals.

```
        for (auto a : dofs_on_element(e)) {
            value_type b = eval_basis(e, q, a);

            double rho = 1;
            double axb = (-s[0][0]*b.dx - s[0][1]*b.dy - s[0][2]*b.dz
                                        + F[0]*b.val) / rho;
            double ayb = (-s[1][0]*b.dx - s[1][1]*b.dy - s[1][2]*b.dz
                                        + F[1]*b.val) / rho;
            double azb = (-s[2][0]*b.dx - s[2][1]*b.dy - s[2][2]*b.dz
                                        + F[2]*b.val) / rho;

            double dt = steps.dt;
            double t2 = dt * dt / 2;
            auto aa = dof_global_to_local(e, a);
            ref(loc.ux, aa) += ((ux.val+dt*vx.val)*b.val + t2*axb)*w*J;
            ref(loc.uy, aa) += ((uy.val+dt*vy.val)*b.val + t2*ayb)*w*J;
            ref(loc.uz, aa) += ((uz.val+dt*vz.val)*b.val + t2*azb)*w*J;

            ref(loc.vx, aa) += (vx.val * b.val + dt * axb) * w * J;
            ref(loc.vy, aa) += (vy.val * b.val + dt * ayb) * w * J;
            ref(loc.vz, aa) += (vz.val * b.val + dt * azb) * w * J;
        }
    }
    return loc;
}
```

where `ref` is a helper function defined as

```
double& ref(vector_type& v, index_type idx) const {
    return v(idx[0], idx[1], idx[2]);
}
```

Configuring and executing the simulation is the same as described in Section 8.7 except that it uses 3D counterpart of the configuration structure:

```
ads::config_3d c{ dim, dim, dim, steps, 1 };
```

Suppose we want to output the displacement fields at some points during the simulation. This can be done in the same way as for the other problems using 3 sets of files, e.g.

```
output.to_file(now.ux, "ux_%d.vti", iter);
output.to_file(now.uy, "uy_%d.vti", iter);
output.to_file(now.uz, "uz_%d.vti", iter);
```

It is also possible, however, to store all three components in a single file:

```
output.to_file("out_%d.vti", iter,
               output.evaluate(now.ux),
               output.evaluate(now.uy),
               output.evaluate(now.uz));
```

Loading the data and using *Warp by vector* filter and *Surface* as *Representation* we can generate pictures as presented in Fig. 4 (the colors correspond to the elastic energy of the deformations—see `problems/elasticity` for details).

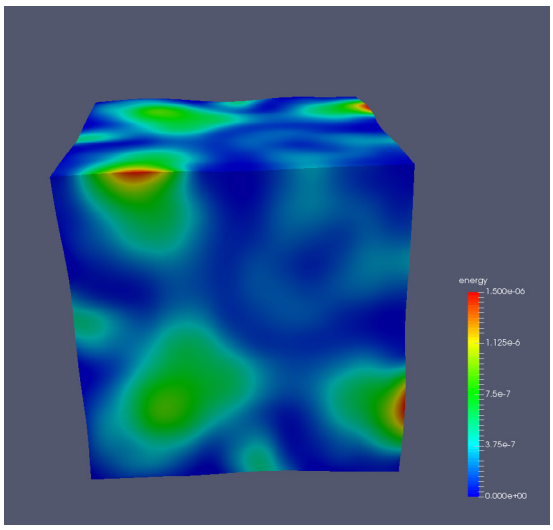
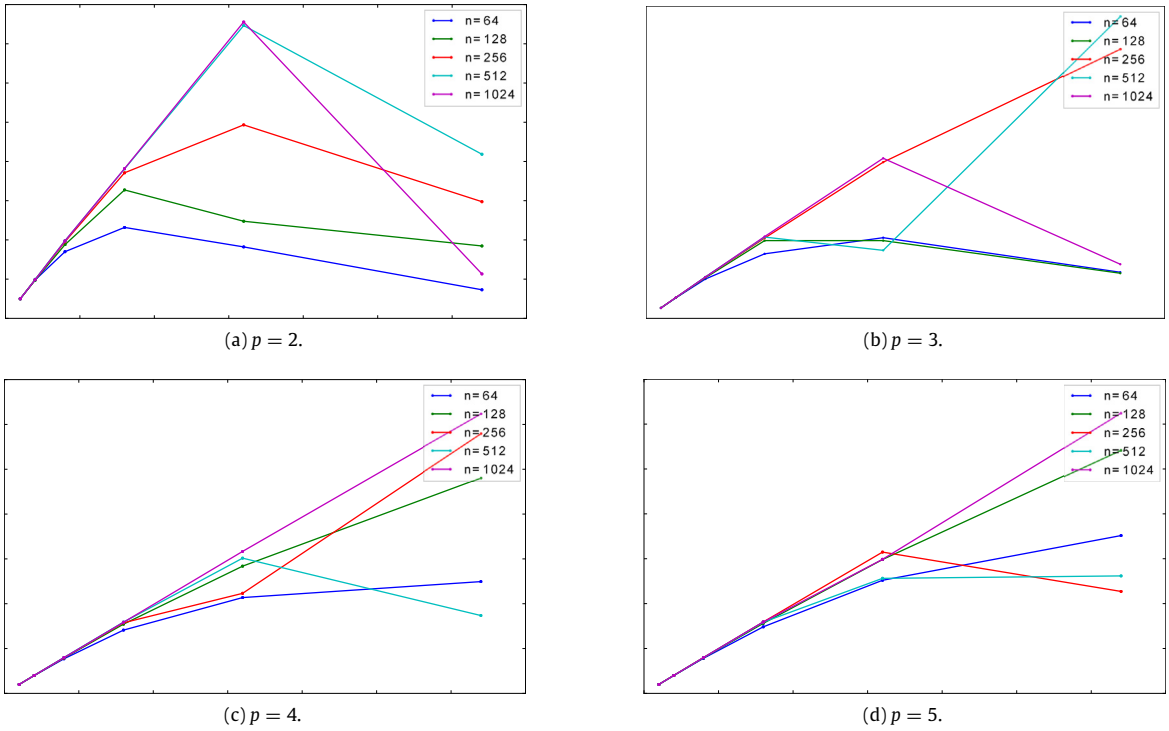


Fig. 4. Linear elasticity solution (color corresponds to elastic energy). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)



(a) $p = 2$. (b) $p = 3$.

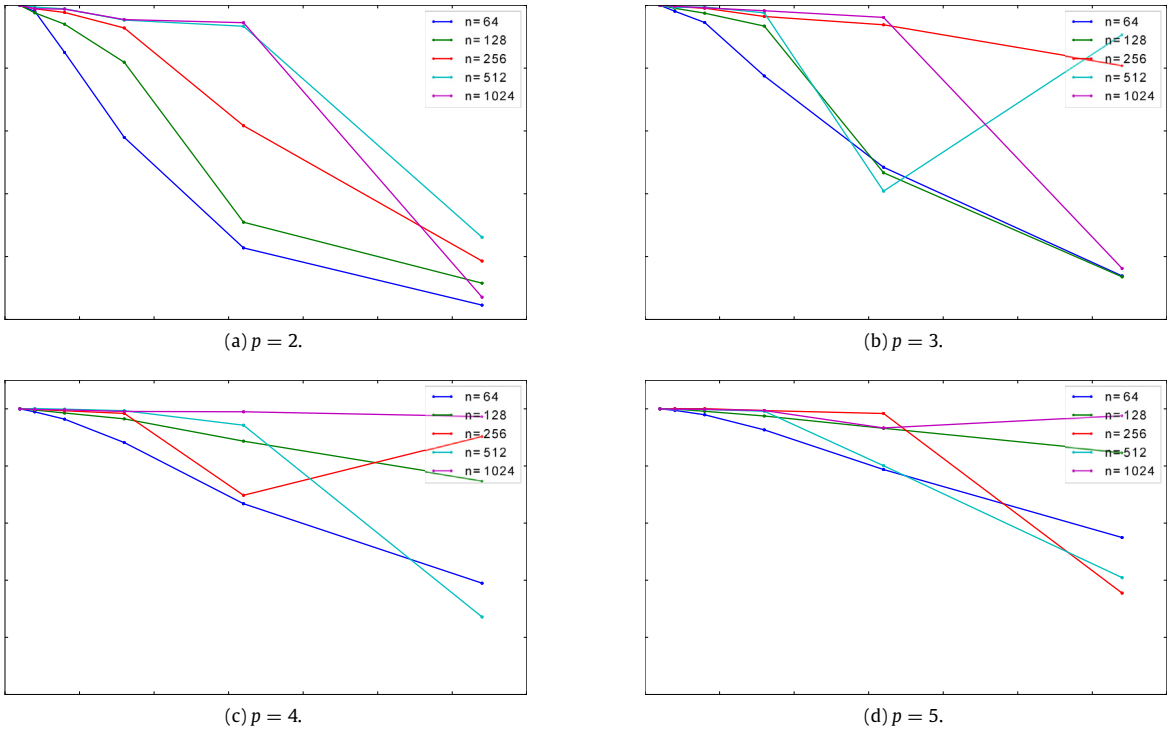
(c) $p = 4$. (d) $p = 5$.

Fig. 5. Speedup of integration in 2D depending on order of basis p and size of the mesh n.

9. Performance tests

Numerical tests of IGA-ADS have been carried out with the goal of determining the scalability of Galois-based parallel implementation. We used the Gilbert cluster belonging to ICES at the University of Texas. It comprises four Intel® Xeon® CPU E7-4860 processors, each possessing 10 physical cores (for a total of 40 cores). We compare the speedup (sequential computation time divided by computation time for k threads) and efficiency (speedup divided by the number of threads k) for various numbers of threads and problem sizes (n denotes the number of elements in each dimension). Results are displayed in Figs. 5, 6 (2D) and 7, 8 (3D). For these tests, we used the two and three-dimensional heat transfer problem described in Section 4.1.

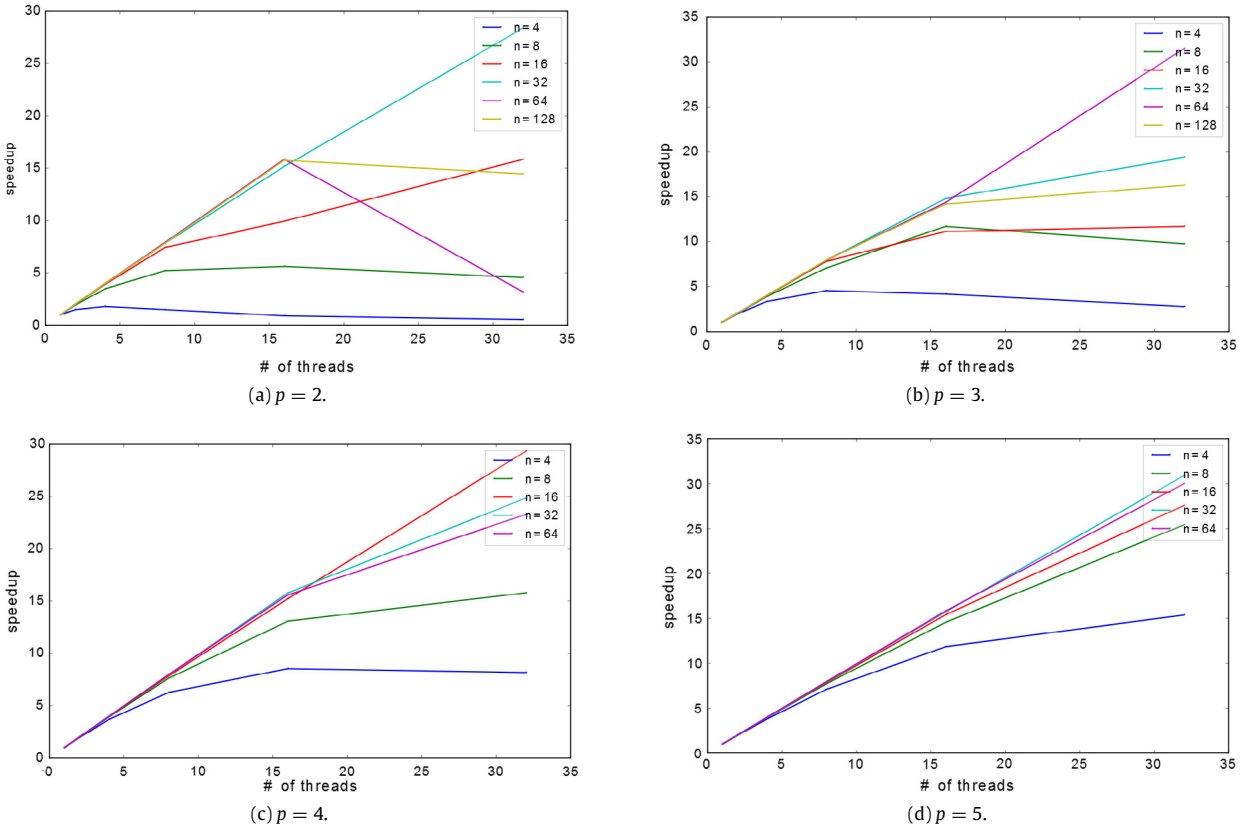
As expected, the more work per element, the better the results. For $p = 2$ and low polynomial orders the amount of calculations performed for each element is relatively low and so the gain from parallelization is significantly diminished by synchronization overhead—the efficiency drops significantly for the number of cores above 8–16 6(a). For higher order of polynomials ($p = 4, 5$) the speedup is markedly better. The situation is similar for the three-dimensional problem, but the observed speedup is higher since the amount of work per element grows. For $p = 4, 5$ the parallelization is close to perfect 7(c), 7(d). This is expected since the amount of work during the



(a) $p = 2$. (b) $p = 3$.

(c) $p = 4$. (d) $p = 5$.

Fig. 6. Efficiency of integration parallelization in 2D depending on order of basis p and size of the mesh n.

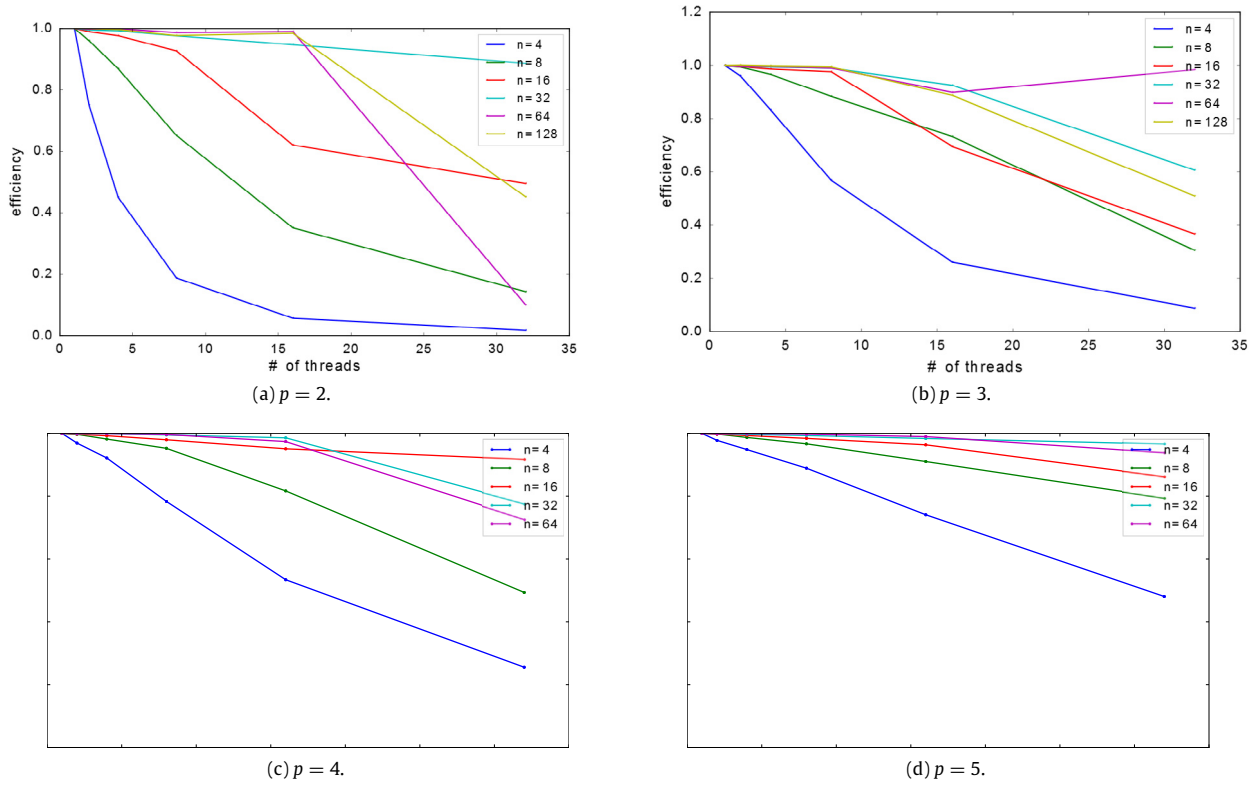


(a) $p = 2$. (b) $p = 3$.

(c) $p = 4$. (d) $p = 5$.

Fig. 7. Speedup of integration in 3D depending on order of basis p and size of the mesh n.

integration grows significantly with the order of the basis (see Section 2.4). In general, the speedup for small meshes is lower, although the impact of mesh size is less visible for high p.



(a) $p = 2$. (b) $p = 3$.

(c) $p = 4$. (d) $p = 5$.

Fig. 8. Efficiency of integration parallelization in 3D depending on order of basis p and size of the mesh n.

The plots depict only the time of the integration, i.e. assembling the right-hand side of the linear system, which constitutes the vast majority of computation time. Cost of rest of each single time step – solving the system using ADS – was found to be negligible in comparison, and preparatory steps (array allocation) become insignificant given a large number of time steps one usually needs to compute in a full simulation.

10. Conclusions

We presented a parallel explicit dynamics solver using isogeometric L2 projections, implemented within GALOIS framework. The solver enables a fast parallel solution of time-dependent problems for which an explicit dynamics method is suitable. It can be executed over shared memory multi-core parallel machines. It provides interfaces for the definition of user-defined PDE and boundary conditions. It also generates output suitable for generation of movies from the performed simulations by using Gnuplot and ParaView tools. We also provide exemplary implementations for heat transfer, linear elasticity and non-linear flows in heterogeneous media.

Acknowledgments

The work presented in this paper was supported by National Science Centre, Poland, Grants Nos. 2016/21/B/ST6/01539, 2015/19/B/ST8/01064, and 2014/15/N/ST6/04662. The visit of Maciej Paszyński at ICES was supported by J.T.O. Research Faculty Fellowship. This research of Prof. Keshav Pingali group was supported by NSF grants 1218568, 1337281, 1406355, and 1618425, and by DARPA BRASS contract 750-16-2-0004.

References

[1] D.W. Peaceman, H.H. Rachford, SIAM J. Appl. Math. 3 (1) (1955) 28–41.
[2] J. Douglas, H.H. Rachford, Trans. Amer. Math. Soc. 82 (2) (1956) 421–439.
[3] G. Birkhoff, R.S. Varga, D. Young, Adv. Comput. 3 (1962) 189–273.
[4] L. Gao, V.M. Calo, Comput. Methods Appl. Mech. Engrg. 274 (0) (2014) 19–41.
[5] M. Łoś, M. Woźniak, M. Paszyński, L. Dalcin, V.M. Calo, Proc. Comp. Sci. 51 (2015) 286–295.
[6] L. Gao, V.M. Calo, J. Comput. Appl. Math. 273 (0) (2015) 274–295.
[7] J.A. Cottrell, T.J.R. Hughes, Y. Bazilevs, Isogeometric Analysis: Toward Integration of CAD and FEA, first ed., Wiley Publishing, New York, 2009.
[8] M. Łoś, M. Woźniak, M. Paszyński, Int. J. Computer Sci. Eng. 5 (2) (2016) 99–107.
[9] M. Łoś, M. Paszyński, A. Kłusek, W. Dzwinel, Comput. Methods Appl. Mech. Engrg. 316 (2017) 1257–1269.
[10] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M.A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Mendez-Lojo, D. Prountzos, X. Sui, SIGPLAN Not. 46 (6) (2011) 12–25.
[11] L. Demkowicz, Computing with Hp-Adaptive Finite Elements, Taylor & Franics, CRC Press, New York, 2006.
[12] M. Elhaddad, N. Zander, S. Kollmannsberger, A. Shadavakhsh, V. Nubel, E. Rank, Int. J. Struct. Stab. Dyn. 15 (2015) 1–25.
[13] L. Dalcin, N. Collier, P. Vignal, A. Cortes, V.M. Calo, Comput. Methods Appl. Mech. Engrg. 308 (2016) 151–181.

[14] S. Balay, S. Abhyankar, M.F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, K. Rupp, B.F. Smith, S. Zampini, H. Zhang, H. Zhang, PETSc Web page, 2016. http://www.mcs.anl.gov/petsc.
[15] V.P. Nguyen, C. Anitescu, S.P. Bordas, T. Rabczuk, Math. Comput. Simulation 117 (2015) 89–116.
[16] R. Vazquez, IMATI Report Series 16 (02) (2016) 1–44.
[17] B. Juettler, U. Langer, A. Mantzaflaris, S. Moore, W. Zulehner, Proc. Appl. Math. Mech. 14 (1) (2014) 961–962.
[18] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK User Guide, Society for Industrial and Applied Mathematic, Philadelphia, 1999.
[19] Boost C++ library. URL http://www.boost.org/.
[20] Galois. URL http://iss.ices.utexas.edu/?p=projects/galois.
[21] libunittest. URL http://libunittest.sourceforge.net/.
[22] GCC: Gnu Compiler Collection. URL https://gcc.gnu.org/.
[23] Cmake. URL https://cmake.org/.
[24] File formats for VTK version 42. http://www.vtk.org/wp-content/uploads/2015/04/file-formats.pdf.
[25] Gnuplot. URL http://www.gnuplot.info/.
[26] A. Henderson, The ParaView Guide: A Parallel Visualization Application, Kitware Inc., 2007.