Future Generation Computer Systems 140 (2023) 151–172

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Cloud-native alternating directions solver for isogeometric analysis

Grzegorz Gurgul, Bartosz Baliś, Maciej Paszyński *

AGH University of Science and Technology, al. Mickiewicza 30, 30-059, Krakow, Poland

ARTICLE INFO

Article history:
Received 1 May 2022
Received in revised form 9 October 2022
Accepted 17 October 2022
Available online 25 October 2022

Keywords:
Cloud computing
Isogeometric analysis (IGA)
Alternating-direction implicit solver (ADI)
Think like a vertex (TLAV) paradigm
Parallel programming paradigm Pregel
Apache Giraph framework

ABSTRACT

Computer simulations with isogeometric analysis (IGA) have multiple applications, from phase-field modeling to tumor-growth simulations. We focus on the alternating-directions solver (ADS) algorithm, in which the matrix equation representing a computational problem is decomposed into parallel tasks following the binary and balanced structure of an elimination tree. In this paper, we explore the possibility of running large-scale IGA simulations using linear computational cost alternating direction solvers on top of modern data-parallel cloud computing frameworks. To this end, we propose a new way of decomposition of the elimination tree which makes the IGA alternating-direction solver effectively a large graph problem suitable for modern cloud-computing frameworks. On this basis, we propose a new algorithm for isogeometric analysis alternating-directions solver based on the Pregel computational model, used for large-scale graph-processing in the cloud. We implement a cloud-native solver using this algorithm in the Apache Giraph framework, and show that it can be applied for solution of challenging higher-order PDEs. We evaluate the solver in terms of various scalability models and run configurations. The results indicate linear scalability of the proposed algorithm with respect to the number of elements in the mesh.

© 2022 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

1.1. Context and motivation

Cloud computing has become the undisputed and widely adopted answer to many of most important challenges of modern software industry [1,2]. It has revolutionized the way software is designed and run primarily by taking away the incidental complexities of managing the underlying infrastructure and providing elastic scalability where resources are always available and paid for in a pay-per-use model [3]. A number of parallel cloud-computing techniques like MapReduce [4] or Resilient Distributed Datasets [5], and frameworks such as Hadoop [6] and Spark [7] have been proposed to leverage the cloud capabilities. This synergy seems to be pushing the boundaries of what a cloud can be used for to the area which has been historically dominated by HPC [8]. Many distributed-memory computations traditionally bound to MPI were moved to the cloud-native data-parallel frameworks such as Hadoop and Spark in hopes to improve the efficiency of working with excessively large data sets which are present in such fields as genomics or astronomy [1,9]. The widespread adoption of clouds revealed that simulation efficiency is not just the function of solver performance, but also factors like simplicity of implementation, ease of integration with other software, reliability of execution, portability between clusters and, perhaps most importantly, flexibility of scheduling leveraging the cloud's rapid on-demand resource provisioning model [8,10]. In fact, rapid resource provisioning can lead to lower turnaround times, even despite lower performance in comparison with classic HPC clusters [11].

The isogeometric analysis (IGA) [12] seems to have provided long awaited solution for a costly dichotomy between Computer-Aided Design (CAD) and Computer-Aided Engineering (CAE) aspects of any modern manufacturing process [13]. Some of its features, particularly higher continuity [14], opened many new and exciting research opportunities for solving problems which involve higher-order differential operators. These include a wide range of important engineering problems ranging from phase field modeling [15], phase separation simulations with Cahn–Hilliard [16] or Navier–Stokes–Korteweg higher order models [17], to the blood flow [18], propagation of elastic waves [19], nonlinear flow in heterogeneous media [20] or tumor growth modeling [14]. Many of these problems produce large systems of linear equations which contain integrals that typically require considerable computational cost to initialize let alone solve. For this kind of large, non-stationary simulations, the multi-frontal solver [21] is usually considered to be too expensive to apply and iterative solvers are preferred. One of such solvers is the GMRES [22] which is interfaced by e.g. the PETIGA [13]. These solvers require several matrix–vector multiplications in every

* Corresponding author.
E-mail addresses: gurgul.grzegorz@gmail.com (G. Gurgul), bartosz.balis@agh.edu.pl (B. Baliś), maciej.paszynski@agh.edu.pl (M. Paszyński).

https://doi.org/10.1016/j.future.2022.10.017
0167-739X/© 2022 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

time step, and their efficiency depends on the spectral properties of the matrix. The iterative solvers may be challenging to implement in cloud-based environment, due to the communication costs related to several matrix–vector multiplications performed in every time step of the simulation. Recently, the alternating-directions algorithm was rediscovered in context of IGA [23,24] and used for computing isogeometric L^2 projections on regular patches of elements [23] or as a typical pre-conditioner for non-regular geometries [24]. It is the case because these methods are known to provide linear $\mathcal{O}(N)$ computational complexity for every time step in serial execution and a logarithmic $\mathcal{O}(log(N))$ complexity when executed on a perfectly parallel machine, as the factorization can be done only once, for all right-hand-sides. The latter property has encouraged many researchers to devise various implementations for these solvers on shared memory multi-core platforms [25] or clusters powered by MPI [20].

The primary motivation behind this study is to *explore the possibility of running large-scale IGA simulations using logarithmic computational cost alternating direction solvers on top of modern data-parallel cloud computing frameworks* hosted as managed services offered by leading public cloud providers. We also show that the solver can be applied for solution of challenging higher-order PDEs, the Cahn–Hilliard equation, usually solved by using IGA method [16,17,26].

1.2. Objectives and contribution

We propose a new algorithm for isogeometric analysis alternating–directions solver based on the "Think Like a Vertex" paradigm, used for large-scale graph-processing. To the best of our knowledge, this is the first distributed memory implementation of an alternating-directions solver for implicit IGA simulations.

We argue that a cloud-native solver which adopts this method allows an arbitrary mesh size as well as an arbitrary time step length (since the two are independent) to be picked and linear computational cost computations to be run on a cloud-managed data-processing service that has the potential to scale seamlessly in the pay-per-use model [10,27–29].

We implement the solver in the cloud-native parallel programming paradigm Pregel, using the Apache Giraph framework. The solver is evaluated on a model Cahn–Hilliard phase-field problem, with the strong and weak scalability reported, and the Amdahl law employed to estimate the potential for parallelization.

The obtained results has led us to the following conclusions which summarize the most important scientific findings described in this paper:

- IGA-ADI is in fact a complex, iterative graph algorithm with strong data locality. IGA-ADI can be represented as a vertex-centric iterative graph algorithm and leverage modern cloud computing frameworks designed for this purpose. The algorithm can be distributed alongside the data to exploit the data locality feature of the graphs, which greatly reduces the communication overhead and benefits horizontal scalability.
- The cost of performing large-scale IGA simulations can be significantly reduced when run on a cluster of preemptible nodes. Cloud computing frameworks are resilient in design which enables the use of preemptive (spot) instances which are on average 4 times as cheap as the regular ones with exactly the same hardware. Taking the elasticity of the cloud into account, the computations can be performed faster at the same cost or cheaper at the same speed.
- We decompose the multi-frontal solver with multiple right-hand-sides into sets of basic tasks, each processing the same number of right-hand-sides over some nodes of the elimination tree, with the possibility of parallel scheduling of these tasks into the cloud parallel architecture. Partitioning of the elimination tree may be a better alternative to the traditional domain decomposition approach.
- This approach for decomposition of the elimination tree makes IGA alternating-direction solver effectively a large graph problem which can be reasoned about using corresponding terms and solved using a variety of modern cloud-computing frameworks. It is optimized for better throughput. This parallelization strategy can be also combined with the domain decomposition approach to achieve better performance for a subset of problems.

The remainder of the paper is organized as follows. We start from Section 2 presenting the idea of the IGA-ADI solver algorithm. Next Sections 3 and 4 discuss how the core of the IGA-ADI algorithm, the multi-frontal solver with multiple right-hand-sides can be expressed as a sequence of tasks (called *Productions*) and how the IGA-ADI can be described as a vertex-centric data-parallel algorithm. Section 5 describes IGA-ADI implementation as a Pregel graph algorithm, with the Giraph framework explained in Section 6. The paper is concluded with numerical experiments and parallel scalability tests in Sections 7 and 8, respectively.

2. Isogeometric alternating directions implicit algorithm (IGA-ADI)

The derivation of the alternating-directions solver for isogeometric analysis simulations (IGA-ADI solver) is presented in Section 2 of [30,31].

The alternating-directions solver for IGA can be used for performing fast simulations of time-dependent problems of the form given by (1) with some predefined initial configuration and boundary conditions

$$\frac{du}{dt} - Lu = f \tag{1}$$

where $L = L_x + L_y$ is a separable differential operator (e.g. Laplacian $L = L_x + L_y = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$). Following the ideas for the explicit dynamics solver [25] we can derive the implicit dynamics solver and represent it in terms of basic indivisible tasks which form a Directed Acyclic Graph (DAG).

First, we apply the alternating direction method with respect to time. We introduce the intermediate time steps in order to separate the differential operator and to process the x derivatives in the first sub-step given by (2), and the y derivatives in the second sub-step given by (3), which is the key operation behind this method.

$$\frac{u_{t+\frac{1}{2}} - u_t}{dt} - L_x u_{t+\frac{1}{2}} - L_y u_t = f_t \tag{2}$$

$$\frac{u_{t+1} - u_{t+\frac{1}{2}}}{dt} - L_y u_{t+\frac{1}{2}} - L_x u_{t+1} = f_{t+\frac{1}{2}} \tag{3}$$

We can multiply each equation by dt and move the terms

$$u_{t+\frac{1}{2}} - dtL_x u_{t+\frac{1}{2}} = u_t + dtL_y u_t + dtf_t \tag{4}$$

$$u_{t+1} - dtL_y u_{t+\frac{1}{2}} = u_{t+\frac{1}{2}} + dtL_x u_{t+\frac{1}{2}} + dtf_{t+\frac{1}{2}} \tag{5}$$

If we compare the implicit dynamics solver to the explicit dynamics solver [25], we can see that now on the left-hand-side we also keep the derivatives in their respective directions. In other words, we include the differential operator in the L^2 projection process.

Now, we transform the problem into the weak form by taking the L^2-scalar products with test functions v

$$(u_{t+\frac{1}{2}} - dtL_x u_{t+\frac{1}{2}}, v)_{L^2} = (u_t + dtL_y u_t + dtf_t, v)_{L^2}$$
$$(u_{t+1} - dtL_y u_{t+\frac{1}{2}}, v)_{L^2} =$$
$$(u_{t+\frac{1}{2}} + dtL_x u_{t+\frac{1}{2}} + dtf_{t+\frac{1}{2}}, v)_{L^2}$$

Let us focus on the first one moving forward. We multiply it by the test function v

$$\int (u_{t+\frac{1}{2}} - dt\frac{\partial u_{t+\frac{1}{2}}}{\partial x})vdxdy = \int (u_t + dtL_y u_t + dtf_t)dxdy$$

and then integrate by parts the second of its terms

$$\int u_{t+\frac{1}{2}} vdxdy + dt \int u_{t+\frac{1}{2}} \frac{\partial u_{t+\frac{1}{2}}}{\partial x}\frac{\partial v}{\partial x}dxdy =$$
$$\int (u_t + dtL_y u_t + dtf_t)vdxdy$$

We discretize our problem by using a two-dimensional B-spline basis where each of its elements can be given as $N_{i,j;p}(x,y) = B^x_{i;p}(x)B^y_{j;p}(y)$. We approximate $u_{t+\frac{1}{2}} = \sum_{i,j} u^{t+\frac{1}{2}}_{i,j} B^x_{i;p}(x)B^y_{j;p}(y)$ using the test function $v = B^x_{k;p}(x)B^y_{l;p}(y)$. The (6) becomes

$$\sum_{i,j} u^{t+\frac{1}{2}}_{i,j} \left[\int B^x_{i;p}(x)B^y_{j;p}(y)B^x_{k;p}(x)B^y_{l;p}(y)dxdy \right.$$
$$\left. + dt \int \frac{\partial(B^x_{i;p}(x)B^y_{j;p}(y))}{\partial x}\frac{\partial(B^x_{k;p}(x)B^y_{l;p}(y))}{\partial x}dxdy \right] =$$
$$= \int (u_t + dtL_y u_t + dtf_t)B^x_{k;p}(x)B^y_{l;p}(y)dxdy$$

Now, since $\frac{\partial}{\partial x}B^y_{j;p}(y) = 0$ and $\frac{\partial}{\partial x}B^y_{l;p}(y) = 0$, we can skip the basis functions which prevented the matrix from being a tensor product. This results in

$$\frac{\partial}{\partial x}(B^x_{i;p}(x)B^y_{j;p}(y)) = \frac{\partial}{\partial x}(B^x_{i;p}(x))B^y_{j;p}(y) + B^x_{i;p}(x)\frac{\partial}{\partial x}(B^y_{i;p}(y))$$

which implies

$$\sum_{i,j} u^{t+\frac{1}{2}}_{i,j} \left[\int B^x_{i;p}(x)B^y_{j;p}(y)B^x_{k;p}(x)B^y_{l;p}(y)dxdy \right.$$
$$\left. + dt \int \frac{\partial(B^x_{i;p}(x))}{\partial x}B^y_{j;p}(y)\frac{\partial(B^x_{k;p}(x))}{\partial x}B^y_{l;p}(y)dxdy \right] =$$
$$= \sum_{i,j} u^{t+\frac{1}{2}}_{i,j} \Psi_x = \int (u_t + dtL_y u_t + dtf_t)B^x_{k;p}(x)B^y_{l;p}(y)dxdy$$

Now, on the left-hand-side, we can group B-splines of Ψ_x with respect to y and take out the common part, as

$$\int B^x_{i;p}(x)B^y_{j;p}(y)B^x_{k;p}(x)B^y_{l;p}(y)dxdy =$$
$$\int B^x_{i;p}(x)B^x_{k;p}(x)dx \int B^y_{j;p}(y)B^y_{l;p}(y)dy$$
$$\Psi_x = \int B^x_{i;p}(x)B^x_{k;p}(x)B^y_{j;p}(y)B^y_{l;p}(y)dxdy +$$
$$\int (\frac{\partial}{\partial x}B^x_{i;p}(x))(\frac{\partial}{\partial x}B^x_{k;p}(x))B^y_{j;p}(y)B^y_{l;p}(y)dxdy$$

so our left-hand-side sum of integrals Ψ becomes the Kronecker product of two integrals

$$\Psi_x = \left[\int (B^x_{i;p}(x)B^x_{k;p}(x) + (\frac{\partial}{\partial x}B^x_{i;p}(x))(\frac{\partial}{\partial x}B^x_{k;p}(x)))dx \right] \otimes \left[\int (B^y_{j;p}(y)B^y_{l;p}(y))dy \right] \quad (6)$$

We can apply the same process to the second (6), where we just replace the derivatives with respect to the x direction, with the derivatives with respect to the y direction

$$\Psi_y = \left[\int (B^y_{i;p}(y)B^y_{k;p}(y) + (\frac{\partial}{\partial y}B^y_{i;p}(y))(\frac{\partial}{\partial y}B^y_{k;p}(y)))dy \right] \otimes \left[\int (B^x_{j;p}(x)B^x_{l;p}(x))dx \right] \quad (7)$$

This way both sides of the Kronecker product contain the products of 1-dimensional B-splines along the same direction - x and y for (6) and y and x for (7) respectively. The integral of the product of any two B-splines from the same 1-dimensional B-spline basis that are distant from each other by more than $2p+1$ elements is zero, i.e.

$$\int_0^1 B^x_{i;p}(x)B^x_{j;p}(x)dx = \begin{cases} 0 & \text{if } |i-j| > 2p+1 \\ const \neq 0 & \text{otherwise} \end{cases}$$

Each element is described by the same $(p+1)\times(p+1)$ square coefficient matrix regardless of the specific problem solved. Let us go back to (6), which can be presented in a generic matrix form

$$\mathbf{Mx} = \mathbf{b} \quad (8)$$

where $\mathbf{M} = \mathbf{A}\otimes\mathbf{B}$ and $\mathbf{A}$ is $n\times n$ square matrix and $\mathbf{B}$ is $m\times m$ square matrix so $\mathbf{M}_{ij,kl} = \int B^x_{i,p}B^x_{j,p}dx \int B^y_{k,p}B^y_{l,p}dy$ So the system is algebraically equivalent to

$$\begin{bmatrix} \int B^x_{1,p}B^x_{1,p}dx & \cdots & \int B^x_{1,p}B^x_{N_x,p}dx \\ \int B^x_{2,p}B^x_{1,p}dx & \cdots & \int B^x_{2,p}B^x_{N_x,p}dx \\ \vdots & \vdots & \vdots \\ \int B^x_{N_x,p}B^x_{1,p}dx & \cdots & \int B^x_{N_x,p}B^x_{N_x,p}dx \end{bmatrix} \otimes$$
$$\begin{bmatrix} \int B^y_{1,p}B^y_{1,p}dy & \cdots & \int B^y_{1,p}B^y_{N_y,p}dy \\ \int B^y_{2,p}B^y_{1,p}dy & \cdots & \int B^y_{2,p}B^y_{N_y,p}dy \\ \vdots & \vdots & \vdots \\ \int B^y_{N_y,p}B^y_{1,p}dy & \cdots & \int B^y_{N_y,p}B^y_{N_y,p}dy \end{bmatrix} \quad (9)$$

From the properties of the Kronecker product we have $\mathbf{M}^{-1} = (\mathbf{A}\otimes\mathbf{B})^{-1} = \mathbf{A}^{-1}\otimes\mathbf{B}^{-1}$. This implies the following derivation of the linear $\mathcal{O}(N)$ computational cost solver algorithm. From the definition of the Kronecker product we know that

$$\mathbf{M} = \mathbf{A}\otimes\mathbf{B} = \begin{bmatrix} \mathbf{A}B_{11} & \mathbf{A}B_{12} & \cdots & \mathbf{A}B_{1m} \\ \mathbf{A}B_{21} & \mathbf{A}B_{22} & \cdots & \mathbf{A}B_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}B_{m1} & \mathbf{A}B_{m2} & \cdots & \mathbf{A}B_{mm} \end{bmatrix} \quad (10)$$

The right-hand-side and the solution vectors are partitioned into m blocks of size n each

$$\mathbf{x}_i = (x_{i1}, \ldots, x_{in})^{\mathrm{T}}$$
$$\mathbf{b}_i = (b_{i1}, \ldots, b_{in})^{\mathrm{T}} \quad (11)$$

We can rewrite our system as a block matrix equation

$$\begin{cases} \mathbf{A}B_{11}\mathbf{x}_1 + \mathbf{A}B_{12}\mathbf{x}_2 + \cdots + \mathbf{A}B_{1m}\mathbf{x}_m = \mathbf{b}_1 \\ \mathbf{A}B_{21}\mathbf{x}_1 + \mathbf{A}B_{22}\mathbf{x}_2 + \cdots + \mathbf{A}B_{2m}\mathbf{x}_m = \mathbf{b}_2 \\ \vdots \quad\quad \vdots \quad\quad \vdots \quad\quad \vdots \\ \mathbf{A}B_{m1}\mathbf{x}_1 + \mathbf{A}B_{m2}\mathbf{x}_2 + \cdots + \mathbf{A}B_{mm}\mathbf{x}_m = \mathbf{b}_m \end{cases}$$

where we factor out $\mathbf{A}$

$$\begin{cases} \mathbf{A}\big(B_{11}\mathbf{x}_1 + B_{12}\mathbf{x}_2 + \cdots + B_{1m}\mathbf{x}_m\big) = \mathbf{b}_1 \\ \mathbf{A}\big(B_{21}\mathbf{x}_1 + B_{22}\mathbf{x}_2 + \cdots + B_{2m}\mathbf{x}_m\big) = \mathbf{b}_2 \\ \vdots \quad\quad \vdots \quad\quad \vdots \quad\quad \vdots \\ \mathbf{A}\big(B_{m1}\mathbf{x}_1 + B_{m2}\mathbf{x}_2 + \cdots + B_{mm}\mathbf{x}_m\big) = \mathbf{b}_m \end{cases} \quad (12)$$

Then we multiply the result by $\mathbf{A}^{-1}$ and define $\mathbf{y}_i = \mathbf{A}^{-1}\mathbf{b}_i$. We receive one 1-dimensional problem $\mathbf{A}\mathbf{y}_i = \mathbf{b}_i$ with multiple right-hand-sides

$$\begin{cases} B_{11}\mathbf{x}_1 + B_{12}\mathbf{x}_2 + \cdots + B_{1m}\mathbf{x}_m = \mathbf{y}_1 \\ B_{21}\mathbf{x}_1 + B_{22}\mathbf{x}_2 + \cdots + B_{2m}\mathbf{x}_m = \mathbf{y}_2 \\ \quad\vdots \qquad\quad \vdots \qquad \vdots \qquad\quad \vdots \\ B_{m1}\mathbf{x}_1 + B_{m2}\mathbf{x}_2 + \cdots + B_{mm}\mathbf{x}_m = \mathbf{y}_m \end{cases} \tag{13}$$

Finally, we consider each component of $\mathbf{x}_i$ and $\mathbf{y}_i$ to obtain a family of linear systems

$$\begin{cases} B_{11}x_{1i} + B_{12}x_{2i} + \cdots + B_{1m}x_{mi} = y_{1i} \\ B_{21}x_{1i} + B_{22}x_{2i} + \cdots + B_{2m}x_{mi} = y_{2i} \\ \quad\vdots \qquad\quad \vdots \qquad \vdots \qquad\quad \vdots \\ B_{m1}x_{1i} + B_{m2}x_{2i} + \cdots + B_{mm}x_{mi} = y_{mi} \end{cases} \tag{14}$$

for each $i = 1, \ldots, n$. We obtain another 1-dimensional problem with multiple right-hand-sides $\mathbf{B}\mathbf{x}_i = \mathbf{y}_i$. We can expand the systems given by (13) and (14) into their matrix forms given by (15) and (16)

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & 0 \\ A_{21} & A_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A_{nn} \end{bmatrix} \begin{bmatrix} y_{11} & \cdots & y_{m1} \\ y_{12} & \cdots & y_{m1} \\ \vdots & \ddots & \vdots \\ y_{1n} & \cdots & y_{mn} \end{bmatrix} = = \begin{bmatrix} b_{11} & \cdots & b_{m1} \\ b_{12} & \cdots & b_{m2} \\ \vdots & \ddots & \vdots \\ b_{1n} & \cdots & b_{mn} \end{bmatrix} \tag{15}$$

$$\begin{bmatrix} B_{11} & B_{12} & \cdots & 0 \\ B_{21} & B_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & B_{mm} \end{bmatrix} \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ x_{21} & \cdots & x_{2n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix} = = \begin{bmatrix} y_{11} & \cdots & y_{1n} \\ y_{21} & \cdots & y_{2n} \\ \vdots & \ddots & \vdots \\ y_{m1} & \cdots & y_{mn} \end{bmatrix} \tag{16}$$

The first system given by (15) represents the interactions of the B-spline basis functions along the X axis, with multiple right-hand-sides, corresponding to the B-spline basis functions along the Y axis. Similarly, the second system given by (16) represents the interactions of B-spline basis functions along the Y axis, with multiple right-hand-sides, corresponding to the B-spline basis functions along the X axis. Both coefficient matrices $\mathbf{A}$ and $\mathbf{B}$ are banded as the consequence of the limited support of products of pairs of 1-dimensional B-splines defined by their row and column indices, which are close enough ($|i-j| < p+1$) to each other only within the band, in the neighborhood of the diagonal. The band width is equal to $2p+1$. The algorithm is the following

- We solve the first system (15) $\mathbf{A}\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$.
- We transpose the solution $\mathbf{y}^T$.
- We solve the second system (16) $\mathbf{B}\mathbf{x} = \mathbf{y}^T$, for $\mathbf{x}$.

Each of the banded systems produced by alternating-directions solver for IGA can be solved by a variety of existing direct and iterative solvers, but the most efficient strategy, due to the banded structures of the matrices, involves the idea of a multi-frontal solver algorithm [21], a parallelizable extension of the frontal [32] algorithm.

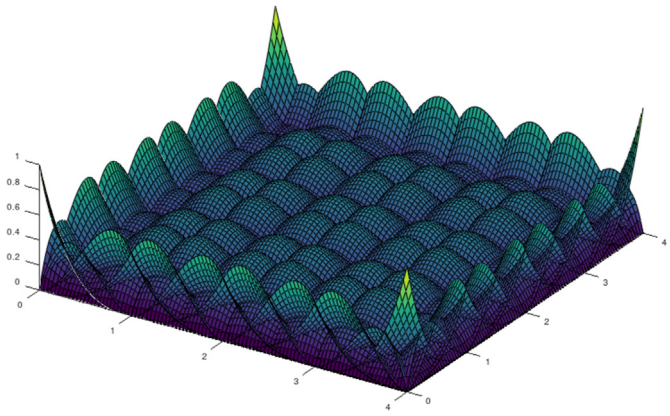


Fig. 1. A 2-dimensional computational mesh. $E_{i,j}$ denotes an individual finite element and α_i and β_j correspond to knots along x and y directions respectively.

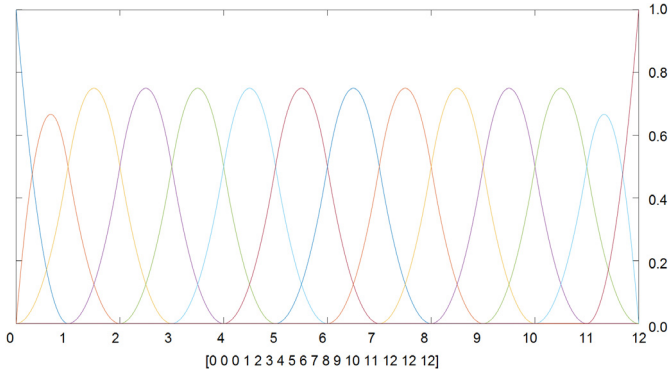


Fig. 2. Quadratic B-splines basis for 12 elements along one direction. The vector at the bottom of the chart is a knot vector which corresponds to the chart horizontal values.

We will decompose the multi-frontal solver with multiple right-hand-sides into sets of basic tasks, each processing the same number of right-hand-sides over some nodes of the elimination tree, with the possibility of parallel scheduling of these tasks into the cloud parallel architecture.

The utility of using the multi-frontal solver for solving the systems generated by the alternating-directions solver for IGA can be best explained using the example of a simple 14×14 matrix, which holds the B-spline coefficients for a single dimension of the 2-dimensional 12×12 elements problem, discretized using quadratic B-splines as depicted in Fig. 1.

The system is transformed into a Kronecker product of two 1-dimensional problems (9) given by

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{17}$$

where

$$\mathbf{A} = \begin{bmatrix} \int B^x_{1,2}(x)B^x_{1,2}(x)dx & \cdots & \int B^x_{1,2}(x)B^x_{14,2}(x)dx \\ \vdots & \vdots & \vdots \\ \int B^x_{14,2}(x)B^x_{1,2}(x)dx & \cdots & \int B^x_{14,2}(x)B^x_{14,2}(x)dx \end{bmatrix} \tag{18}$$

$$\mathbf{x} = \begin{bmatrix} x_{1,1} & \cdots & x_{1,14} \\ x_{2,1} & \cdots & x_{2,14} \\ \vdots & \vdots & \vdots \\ x_{14,1} & \cdots & x_{14,14} \end{bmatrix} \tag{19}$$

and

$$\mathbf{b} = \begin{bmatrix} \int B^x_{1,2}(x)B^x_{1,2}(x)F & \cdots & \int B^x_{1,2}(x)B^x_{14,2}(x)F \\ \int B^x_{2,2}(x)B^x_{1,2}(x)F & \cdots & \int B^x_{2,2}(x)B^x_{14,2}(x)F \\ \vdots & \vdots & \vdots \\ \int B^x_{14,2}(x)B^x_{11,2}(x)F & \cdots & \int B^x_{14,2}(x)B^x_{14,2}(x)F \end{bmatrix} \tag{20}$$

for the first direction, and

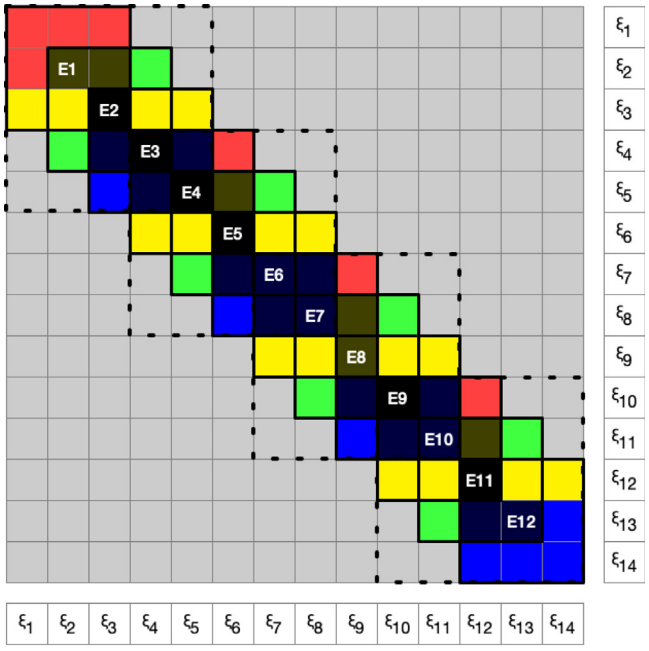
$$\mathbf{By} = \mathbf{x}^T \tag{21}$$

for the second, where $\mathbf{x}$ represents the intermediate solution after the factorization with B-splines along the direction x (see Fig. 2), and $\mathbf{y}$ represents here the final solution obtained after factorization with B-splines along the direction y, using the transposition of the intermediate solution $\mathbf{x}^T$ on the right-hand side of the second sub-system

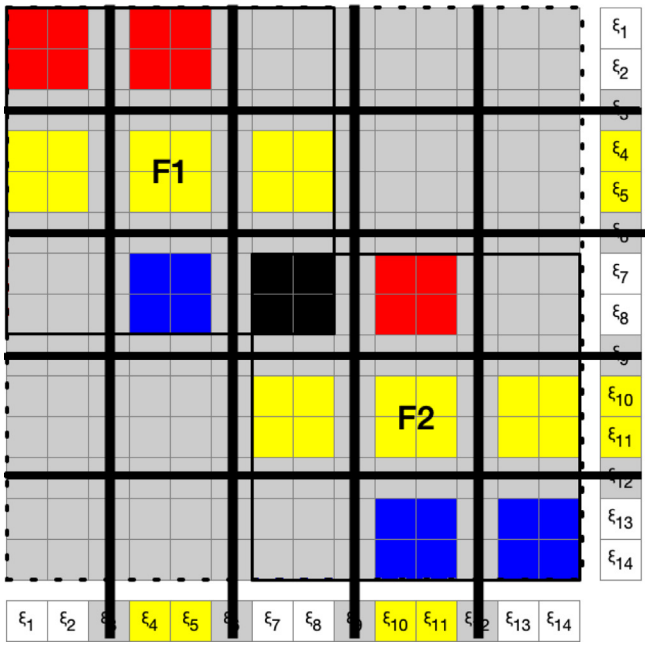
$$\mathbf{B} = \begin{bmatrix} \int B^x_{1,2}(x)B^x_{1,2}(x)dx & \cdots & \int B^x_{1,2}(x)B^x_{14,2}(x)dx \\ \vdots & \vdots & \vdots \\ \int B^x_{14,2}(x)B^x_{1,2}(x)dx & \cdots & \int B^x_{14,2}(x)B^x_{14,2}(x)dx \end{bmatrix} \tag{22}$$

$$\mathbf{y} = \begin{bmatrix} y_{1,1} & \cdots & y_{1,14} \\ y_{2,1} & \cdots & y_{2,14} \\ \vdots & \vdots & \vdots \\ y_{14,1} & \cdots & y_{14,14} \end{bmatrix}. \tag{23}$$

This problem has 14 right-hand sides and 14 sets of 14 weights for the respective B-spline functions. Let us use a generic variable ξ_{kj} for the unknowns, as the process is the same for both systems. Moreover, the factorization is performed only once and applied simultaneously on all right-hand-sides, so only a single set of unknowns ξ_j (for a single right-hand side) is presented for brevity in Fig. 3 and the following figures. All operations performed on the coefficient matrix also affect the right-hand sides, whose results are necessary for the backward substitution. As these operations are driven by the factorization process, we also skip this side of the equation. The frontal matrices can be derived from decomposing this band into the smallest set of partially overlapping 3×3 matrices, which correspond to the coefficients relevant for the individual contributions of finite elements, as depicted on panel (a) in Fig. 3. In general, each row of the matrix has $2p + 1$ non-zero entries (except the first and the last p rows), for B-splines of order p. The matrices are banded and are cheap to factorize. The first step, depicted on panel (a) in Fig. 3, is to construct the fronts, which include appropriate finite elements. There are 12 finite elements in this example, each influencing 3×3 neighboring coefficients along the band. The coefficients on the matrix diagonal contribute to 3 neighboring finite elements except the leading and trailing ones. Therefore, to eliminate at least one variable on all fronts, it is necessary to include three finite elements in each front. We construct 4 fronts which merge 3 finite elements into dense 5×5 frontal matrices $F_1 = (E_1, E_2, E_3)$, $F_2 = (E_4, E_5, E_6)$, $F_3 = (E_7, E_8, E_9)$, $F_4 = (E_{10}, E_{11}, E_{12})$, depicted on panel (a) in 3. Each of these fronts should be assigned to a different thread as they can be processed independently. Variables ξ_3, ξ_6, ξ_9, ξ_{12} become fully assembled, are moved to the pivotal point in a corresponding frontal matrix, and are eliminated through partial Gaussian elimination performed within that front. The color yellow denotes the fully assembled rows. While it is possible to eliminate ξ_1 and ξ_{14}, this only applies to the leading and trailing frontal matrix and is not worth exceptional treatment. The eliminated rows are removed from the fronts, which are denoted by the vertical bars. In the next step, depicted on panel (b) in Fig. 3, the Schur complements of the neighboring fronts are included into the new front – $F_{12} = (F_1, F_2)$ and $F_{34} = (F_3, F_4)$. The variables ξ_4, ξ_5, ξ_{10}, ξ_{11} become fully assembled and are eliminated as well. The corresponding fully assembled rows are also denoted by the color yellow. The next step illustrated on panel (a) in Fig. 4 is to eliminate the two central variables ξ_7, ξ_8, whose rows are denoted by yellow color. These steps can be repeated recursively many times if the matrix is larger. The last step, presented on panel (b) in 4, is to perform a full Gaussian elimination at the root 6×6 matrix, eliminating the last variables ξ_1, ξ_2, ξ_{13}, ξ_{14} using the coefficients merged from the Schur complements from the fronts of the previous step.

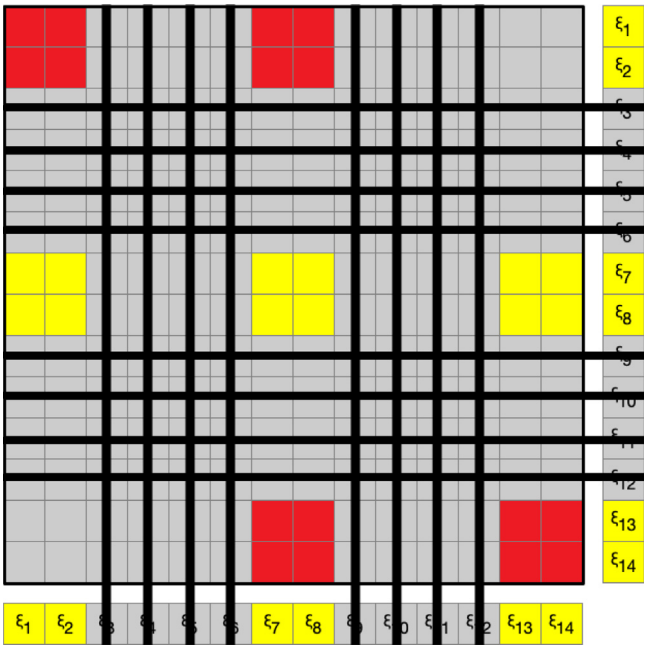


(a) The fully assembled rows (in yellow).

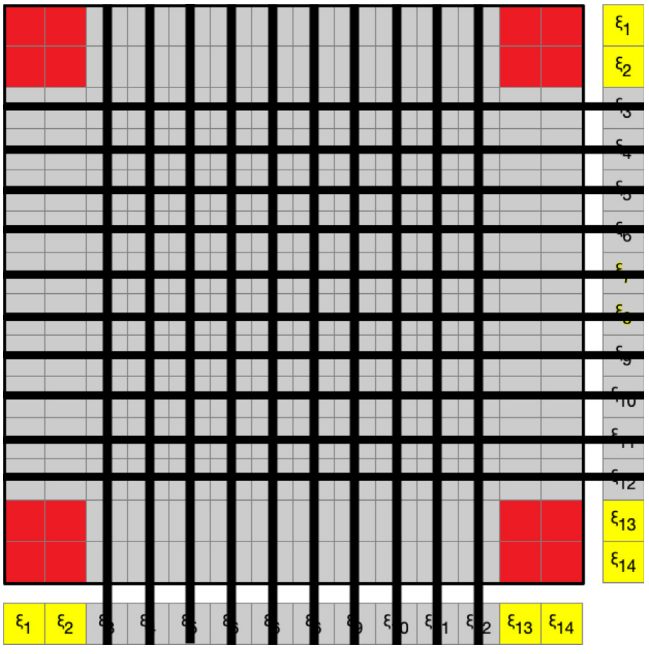


(b) The second step of factorization.

Fig. 3. A simplified visualization of the factorization of the coefficient matrix. The color yellow denotes the fully assembled rows. The color gray indicates zeros. Next, the eliminated rows are removed from the fronts. We represent this process by the horizontal black bars.



(a) The last step of the factorization.



(b) The final front.

Fig. 4. The continuation of the simplified visualization of the factorization of the coefficient matrix.

At this point no unknowns have yet been found but the frontal matrices and their respective right-hand sides are prepared for the backwards substitution phase.

All frontal matrices share common variables only with their nearest neighbors and are independent of other matrices. This relationship is reflected in their position within the elimination tree as the matrices with common variables also share the same parent. There are two distinctive ways of constructing the elimination trees, which constitute the difference between the frontal [32] and multi-frontal [21] approaches. We can best explain this by employing an analogy to the properties of addition.

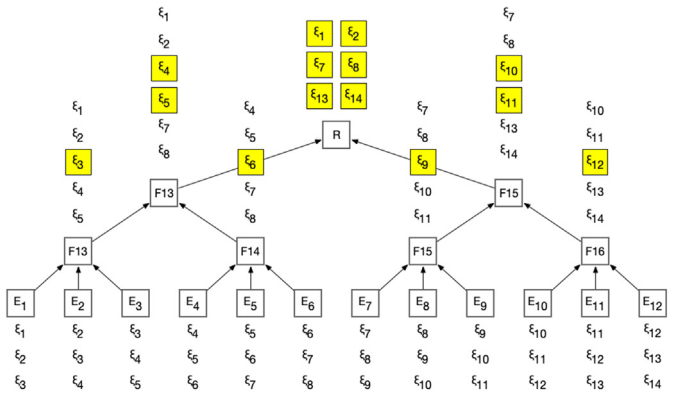


Fig. 5. An elimination tree that allows parallel factorization. The number of variables in each node is listed in a column next to the node. The variables in the yellow boxes are fully assembled and eliminated.

We can use the commutative and associative property of addition to eliminate rows and merge matrices in parallel as in (24), which corresponds to the elimination tree depicted in Fig. 5, which also corresponds to the strategy in Fig. 3.

$$\begin{aligned}&((\cdots+((F^{[1]}+F^{[2]}+F^{[3]})+F^{[4]}+F^{[5]})+\\&\quad+F^{[6]}+F^{[7]})+\dots)\\&((F^{[1]}+F^{[2]}+F^{[3]})+(F^{[3]}+F^{[4]}+F^{[5]})+\dots\\&\quad+(F^{[10]}+F^{[11]}+F^{[12]}))\end{aligned}\tag{24}$$

The alternating-directions solver for the IGA solver described in this study employs a parallel version of the elimination tree to leverage parallel processing. Furthermore, as it works only on regular meshes for which the coefficient matrix is natively banded, the elimination tree is always perfectly balanced without the need to resort to complex reordering techniques. Moreover, the presented example implies that the matrices have low bandwidth, allowing for constructing the elimination tree with a logarithmic depth. The front sizes do not grow with the elimination tree's levels, allowing for efficient parallel processing by the multi-frontal solver.

3. Multi-frontal solver expressed as a sequence of tasks

The multi-frontal solver is the most complicated component of the IGA-ADI which solves the system of linear equations with multiple right-hand-sides.

This is why it had to be properly isolated from any external sources of complexity. It uses its internal set of abstractions to conceptualize and solve the problem:

- *Vertex*, which is the primary data structure used by this solver. Each *Vertex* is a composite [33] which has references to its parent and children of same type. This allows the elimination tree to form, which can easily be navigated up and down from any *Vertex*. Fig. 6 presents the relation between the *Vertex* and the knots of the solution space. The leaf vertices hold the subset of linear equations whose unknowns are the coefficients of the functions used for projection in between two neighboring knots (or the partial solutions which lead to these unknowns). The internal vertices hold the results of merging their children together.
- *Production*, which is the representation of a set of operations that have to be performed on a *Vertex* or its children over the course of the algorithm. While the *Vertex* class can be considered a data structure for the algorithm, *Production* can be referred to as the building blocks of the algorithm itself.

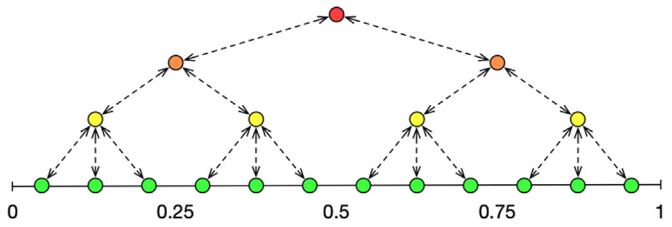


Fig. 6. The relation between the vertices of the elimination tree and individual knots of the domain for an exemplary 12 × 12 problem.
Colors denote the types of vertices — leaves (green), branches (yellow), interim (orange), root (red).

The *Production* interface defines a single method which takes any compatible *Vertex* as an argument and may, but does not need to, modify the state of this vertex or of its direct children. No *Production* is allowed to modify the state of vertices other than the one they are applied to or its direct children. This strict locality feature is the foundation of the parallelization strategy used when solving the problem. From that perspective any implementation of *Production* acts as a command object [33] from the command pattern, which encapsulates the actions, allowing the load to be distributed across different threads. On the other hand, each *Production* may also act as a visitor from the visitor pattern [33]. While all *Production* subtypes share the same interface and are interchangeable from the caller code perspective, they can be classified according to certain features. The most basic classification divides productions into those which participate in solving the problem by mutating the vertices and those which only extract the data leaving the vertices intact. The productions which participate in finding the solution are further divided into groups which correspond to the phase of the algorithm execution and to the type of vertices they are executed on:

- *Construction productions*, which build an empty elimination tree that will be used as the data structure and feature:
  - *BranchRoot*, which creates the *RootVertex*, which spans over the entire domain $(0, 1)$, and two of its direct children of type *InterimVertex*, left and right, that split the domain into the left $(0, 0.5)$ and right $(0.5, 1)$ half;
  - *BranchInterim*, which creates two *InterimVertex* children for each *Vertex* it is applied on, both taking left and right halves of the parent solution space;
  - *BranchLeaves*, which creates three *LeafVertex* children for each *Vertex* it is applied on, left, middle and right, all of which take subsequent $\frac{1}{3}$ shares of the solution space of the parent, which are mapped directly to the knots and span over $p + 1$ of them.
- *Initialization productions*, which initialize the leaves of the elimination tree based on the problem solved or the solution to the other direction. There are two types which correspond to both directions:
  - *InitializeHorizontal*, which uses the injected *Problem* to perform Gaussian integration over appropriate $(p + 1)^2$ B-splines which have non-zero support over the initialized element;
  - *InitializeVertical*, which uses the transposed solution to the first direction as the right-hand-sides for local unknowns.
- *Factorization productions*, which merge and eliminate unknowns at the vertices:
  - *MergeAndEliminateLeaves*, which merges three leaves with 3 rows each at the parent branch and eliminates two fully assembled rows to obtain the systems with a frontal matrix of 5 rows. This production is executed only once;
  - *MergeAndEliminateBranches*, which merges the Schur complements of the branches at the parent interim vertex into a system with a single 6×6 frontal matrix and eliminates one unknown that was fully assembled. This production is executed only once;
  - *MergeAndEliminateInterim*, which merges two interim vertices with 6×6 frontal matrices at the parent interim vertex into a single 6×6 frontal matrix and eliminates one fully assembled variable. This production has to be repeated as many times as is required to reach the *RootVertex*;
  - *MergeAndEliminateRoot*, which merges the Schur complements of two interim vertices into a single 6×6 frontal matrix and performs full Gaussian elimination on that matrix. This production is executed only once.
- *Backward substitution productions*, which propagate the partial solutions down the elimination tree:
  - *BackwardsSubstituteRoot*, which performs partial backwards substitution of the variables and sends appropriate ones to both interim vertices. This production is executed only once;
  - *BackwardsSubstituteInterim*, which performs partial backwards substitution of the variables stored in the parent *InterimVertex* and sends appropriate ones to the child *InterimVertex*. This production is executed as many times as required to reach the vertices one level up the branches;
  - *BackwardsSubstituteBranch*, which performs partial backwards substitution of the variables stored in the parent *InterimVertex* and sends appropriate ones to the child *BranchVertex*. This production is executed only once.

All of these productions solve certain subproblems themselves. One such problems is performing direct *LU* factorization of a local matrix. While this process is an integral part of the IGA-ADI algorithm, it is not unique to IGA-ADI and as a consequence not essential for understanding the task-based algorithm [34]. In other words, the productions are atomic parts of the IGA-ADI solver which convey appropriate meaning in the language used to describe it, separating the user from the complexity of solving common problems. From this point of view the responsibility of the *DirectionSolver* is the orchestration of the process of applying productions to the vertices, which must be done in the correct order and in the most efficient way. This is a complex task by itself and can be addressed properly in isolation from the algorithm's algebraic foundations.

Fig. 7 illustrates the process of solving any problem from the perspective of the *DirectionSolver*. The circles correspond to the vertices, and rounded rectangles denote specific productions. There are the following nodes responsible for the execution of productions constructing the elimination tree (with their workloads related to the execution of particular productions):

- CR(1) — create root (corresponds to *BranchRoot* production),
- CI(2), CI(3) — create interim (corresponds to *BranchInterim* production),
- CL(4)–CL(7) — create leaf (corresponds to *BranchLeaves* production).

The following nodes are related to integration and generation of the element local matrices:

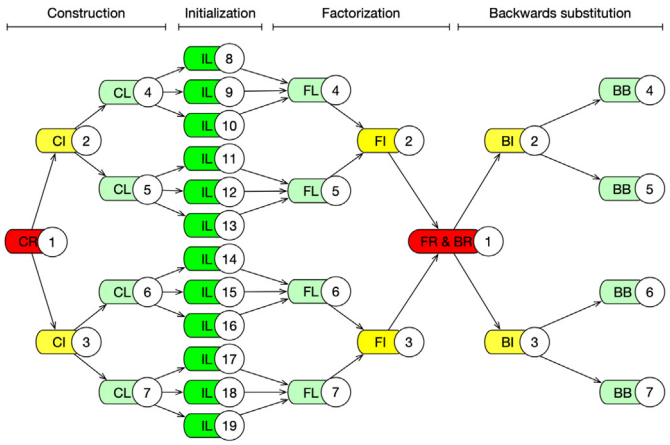


Fig. 7. The DAG of tasks to be performed by the *DirectionSolver* for a regular mesh with 12 elements in one direction.

- IL(8)-IL(19) — initialize leaves (corresponds to *Initialize-Horizontal* production in the first run, and *InitializeVertical* production in the second run, generating element local matrices).

Next, we have nodes corresponding to the factorization process:

- FL(4)–FL(7) — factorize leaves (corresponds to *MergeAndEliminateLeaves*),
- FL(2)–FL(3) — factorize leaves (these two corresponds to *MergeAndEliminateBranches*),
- FR(1) — factorize root (corresponds to *MergeAndEliminateRoot* production).

Finally, we have the nodes that corresponds to the backward substitution phase

- BR(1) — backward substitute root (corresponds to *BackwardSubstituteRoot* production),
- BI(2)–BI(3) — backward substitute interiom (corresponds to *BackwardSubstituteInterim* production),
- BI(4)–BI(7) — backward substitute branch (corresponds to *BackwardSubstituteBranch* production).

First, it needs to construct the elimination tree using the construction productions, which grow the tree level by level, starting from the predefined root vertex. After that, it has to initialize the leaf nodes with values corresponding to the problem being solved, for which it uses the initialization productions. The next stage is merging and elimination, executed consecutively on each vertex, moving up the tree level by level, for which it uses factorization productions. Once the *DirectionSolver* reaches the root of the tree it starts the backward substitution process — it applies backwards substitution productions on each vertex moving down the tree level by level until it reaches the leaves. The last step is to extract the solution from the leaf nodes and return it in the form of the *Solution*. This process is identical for all the problems compatible with the *iga-adi*, but the coefficients of the local matrices are problem-dependent. These coefficients are set up in the initialization phase and are different in every sub-step and in every direction. The *DirectionSolver* isolates itself from the knowledge of this by delegating the creation of productions to the delegate factory class it was instantiated with — the *ProductionFactory*. As a consequence, a certain set of productions are executed when solving the x direction and others when solving the y direction, while the code of the solver stays the same for both cases. Those are examples of how the *"encapsulate what varies and separate it from what stays the same"* design rule [33] can benefit the implementation of a scientific solver.

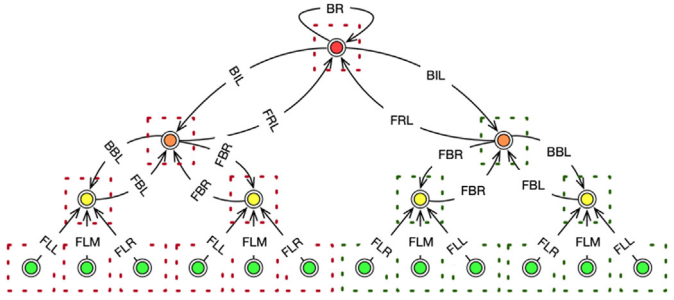


Fig. 8. IGA-ADI as a graph-native algorithm running in the vertices. Vertex colors correspond to vertex types as denoted in Fig. 6. Dotted squares correspond to the tasks executed on available threads — red and green.

4. IGA-ADI as a vertex-centric data-parallel graph algorithm

The IGA-ADI solver requires systems of linear equations to be solved for each direction within each sub-step of every time step of the simulations. The number of unknowns in each of these systems scales linearly with the number of elements in the mesh. The corresponding equations can be efficiently solved using a multi-frontal method. So far we have treated IGA-ADI as a centralized algorithm that executes the transformations on any representation of its data, following the order given by the elimination tree.

IGA-ADI is in fact a complex, iterative graph algorithm with strong data locality. These are precisely the characteristics of the problem that the vertex-centric approach aims to solve. Bearing in mind that the elimination tree is at the heart of IGA-ADI and that the working solution needs to be built on top of the "Think Like a Vertex" (TLAV) [35] abstractions that express and leverage that, we can examine IGA-ADI in the context of four variables of the TLAV model to pick the most suitable implementation architecture.

We treat edges of the elimination tree as directed ones and assign appropriate *Production* to each of them, as depicted in Fig. 8. This way the edge indicates the operation to be performed at the target vertex using specific data from the source vertex. We no longer apply the *Production* on a cluster of vertices, but on the vertices connected by the edge this *Production* is bound to. This is possible, because each *Production* contains operations which are independent. In other words, instead of merging three leaf vertices up front, we can merge them one after another, performing the elimination once all are merged. Once the processing at a single vertex is finished, it can pass the execution along its outgoing edges to its neighbor. This way we can make the assignment of threads to the data also follow the edges of the elimination tree. This means that there are clusters of vertices which are processed by the same thread sequentially.

4.1. Partitioning

The elimination tree and the data therein have to be partitioned between the worker nodes. IGA-ADI includes computationally intensive tasks, which makes a proper level of parallelism a precondition of a good partitioner.

The graph derived in IGA-ADI is just a model whose structure has no (direct) relation to the simulated process. The graph structure is predetermined by the computational mesh selected for solving the problem and most often is static. Therefore, it is possible to use a more advanced partitioner that would find an optimal partitioning which could then be used for all time steps and all subsequent simulations. The degrees of vertices in IGA-ADI are fully deterministic and uniform across each level

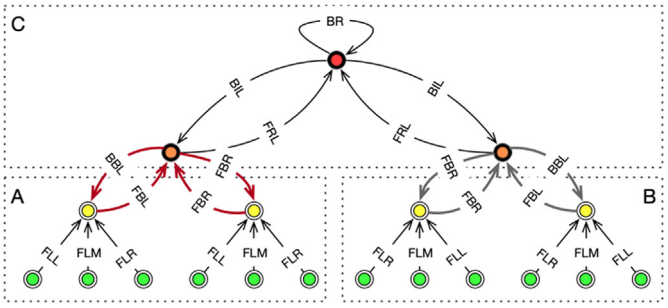


Fig. 9. The method of partitioning a IGA-ADI elimination tree with $N = 2$ partitions and regular mesh with 12 elements in either direction. Certain implementations might benefit from using multiple partitions per worker, in which case partitions have to be placed in a round-robin manner.

of the elimination tree – 1 at the leaves, 4 at the branches, 2 at the root and 3 everywhere else. This also means that the number of edges is comparable to the number of vertices. IGA-ADI stores the data at the vertices utilizing the edges for storing constants — the types of the productions to be executed at the source and the parent. Vertices should never be sent across the partitions to avoid excessive and unnecessary data transfers. All that makes the edge-cut [36] the more favorable strategy for IGA-ADI partitioning as it reduces the chattiness of the algorithm to the data that needs to be exchanged. Moreover, with proper edge-cut partitioning that minimizes the number of edges that span across two partitions only a very small percentage of data needs to actually travel through the network.

The computational meshes employed by the IGA-ADI method are regular of size n in either direction. As this mesh translates into a balanced elimination tree, the depth of the tree which corresponds to the number of serial stages in the algorithm is minimal and the breadth of the tree which corresponds to the number of perfectly parallelizable operations is maximal. Fig. 9 depicts a simple partitioning scheme for a graph which corresponds to a single direction of the problem with $n = 12$ elements in that direction and two physical nodes $V = 2$, which can be easily extended to any number of nodes and any *regular* mesh size where the size of each finite element is equal. Each partition of such graph P_i is given by the partition root vertex p_i and the height h_i of the partition measured as the height of the subtree T_i given by its root p_i. All vertices v_j of the graph that belong to a subtree of the partitioning root p_i excluding the root itself – unless it is also the root of the graph – belong to this partition. This definition of partitions leverages the locality feature of the IGA-ADI and all operations performed within the partitions are local and independent of the operations executed in partitions. The problem of choosing the correct partitioner becomes the problem of finding a set of P_i given by pairs of (p_i, h_i) that:

- Ensures low communication by increasing the heights of the partitions h_i which minimizes the number of edges shared between the partitions;
- Ensures high parallelism by decreasing the heights of the partitions h_i which increases the granularity of task scheduling for better load balancing of computational effort. This ensures that no worker is idle waiting for the others to finish their partitions.

4.2. Coordination

All tasks in the IGA-ADI that are executed on the same level of the elimination tree are independent and can be safely executed concurrently. Each destination vertex, however, requires all inputs to arrive to continue the computations. In the backwards substitution process the data flows downwards, from the parents to the children, all the way down the leaves, which is a natural bulk synchronization point. The existence of the synchronization points in the factorization process does not eliminate the possibility of asynchronous processing of the graph either, because different branches can still be processed concurrently, at a different pace. While both synchronous and asynchronous timing models are applicable, using a fully asynchronous backend for implementing the IGA-ADI in a cloud environment would be an immensely difficult challenge with no convincing prospects of significant gains in performance. Firstly, the designed solver works on regular meshes where the elimination tree is balanced, so there is no obvious need to process branches at a different pace. Secondly, the execution of each production takes roughly the same computational effort to complete regardless of the vertex, so the influence of stragglers inherent in the synchronous execution model [35] can be expected to be restricted to those with origins in the execution platform itself, such as GC in JVM. Lastly, the elimination tree has enough vertices to overshadow the cost of performing the synchronization at each level of the tree, as there will be likely much less threads than vertices to process, which allows bulk synchronicity. Therefore, we can greatly simplify the problem following the BSP idea by making each level of the elimination tree a synchronization barrier. Consequently, the natural boundary of a superstep for the IGA-ADI is a production, which can send the data from multiple source vertices in one superstep S_t, and then combine all of that data to update the state of the target vertices in the next superstep S_{t+1}. This way we can ensure that all data is always available by the time it is needed, but the actual mechanism of transferring the data is abstracted away, which means it can be optimized by the framework.

4.3. Communication

Communication models employed by TLAV frameworks fall into two general categories — shared memory and message passing [37]. They define the mechanism of exchanging the data between the vertex programs [35] which has a profound impact on the IGA-ADI solver.

The shared memory model of distributed communication is generally not applicable to the IGA-ADI solver for practical reasons. Simply put, there are no frameworks that support shared memory communication, asynchronous timing and edge cut partitioning at the same time. There are also specific reasons for which the shared memory model does not provide convincing benefits in this use case. Our partitioning ensures that little data is transferred over the network while finding the solution to the system of linear equations in one direction of the ADI method. In fact, the great majority of them can happen within the same worker process, which makes the shared memory model particularly appealing. However, solving one direction has to be repeated four times within each time step as there are two directions to solve in every projection and there are two projections corresponding to each of two sub-steps derived in the implicit time integration scheme. Consequently, the data might not need to be transferred within the process of solving one direction, but might have to be transferred during the initialization of subsequent runs. There are four global data manipulation operations that have to be performed throughout the execution of IGA-ADI:

- Initialization of direction X of the first time step with an external data source such as a bitmap. The vertices can be colocated with the data so no or little network transfer is required to initialize them. Certain frameworks might require a shuffle;

- Initialization of the leaves in direction Y using the transposed solution to direction X. This includes a potential bottleneck of transposing a large, dense, distributed matrix, which has to be performed once for each projection;
- Initialization of the next time step s_{t+1} direction X which has to be done using the data from the last time step s_t direction Y. This is similar to the initialization of the leaves in direction Y but no transposition is required and each leaf depends on at most two neighboring branches;
- Retrieval of the solution to each time step to save it to persistent storage. This requires no network transfer as the rows of the solution can be saved directly to the filesystem of the node which holds these rows.

The message passing communication model is a considerably simpler and more scalable alternative to shared memory; it is particularly natural for TLAV frameworks which employ synchronous timing and vertex-cut partitioning that also happen to be the features selected for IGA-ADI. Firstly, this means the data does not need to be sent immediately upon request but is delivered asynchronously with a larger delay that is honestly factored into the model and its abstractions. This makes them much more predictable and easier to reason about and optimize. Secondly, we can avoid a full shuffle of data in the vertex-centric IGA-ADI if we reuse the same graph structure for solving the other direction by reinitializing the leaves with the data directly from the branches as depicted in Fig. 10. This way each branch sends the appropriate columns of the solution directly to the leaves so that no data transfer is present for the destinations within the same partition and only the required columns are sent to other vertices — the reduction is performed entirely at the source vertex rather than at the destination. The remainder of the data still needs to be sent across the network but the process is much more efficient compared to the shared memory model. This is the case because there is no need to guard the consistency as the data ownership is limited to a single thread bound to a specific vertex. Moreover, the data can be sent in a *push* model which allows appropriate buffering and the use of larger batches of messages to increase the delivery throughput. Lastly, the transfers can be carried out asynchronously during the execution to prevent the network from becoming saturated. This means that the elimination tree can in fact be created only once and never recreated. This is the rule not only within a single time step but across all time steps. The data is always manipulated *in-situ*, without ever loading, aggregating or otherwise explicitly transforming the data set. Fig. 12 depicts this property. The fact that we reuse the graph throughout the execution does not mean we need to store all of the intermediate matrices. We can save a considerable amount of space by creating the matrices at the vertices lazily once needed and deleting some or all of them if we no longer need them.

Other global operations, such as the initialization of the next time step depicted in Fig. 11, can also be represented in a similar way.

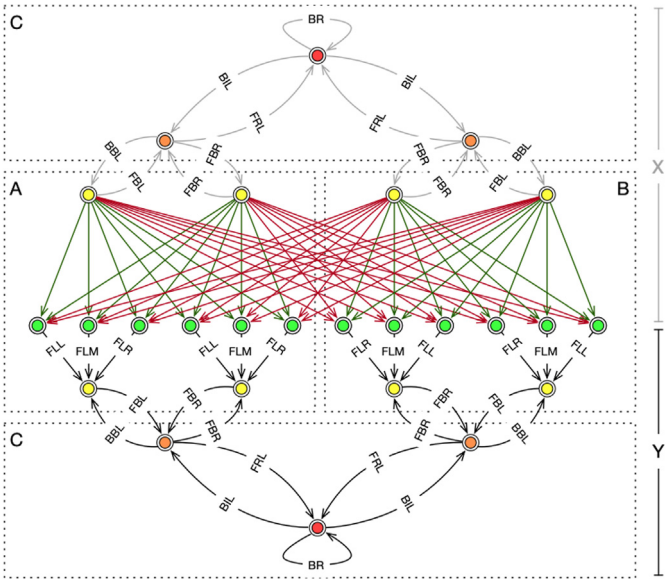


Fig. 10. The transposition step after solving direction X in the 2D vertex-centric IGA-ADI algorithm. Both directions reuse the same graph which was mirrored along the leaves to depict the remaining operations corresponding to solving direction Y. The historic edges corresponding to the factorization of the leaves along X have been skipped for brevity. Green arrows indicate no network transfer while the red ones show the opposite.

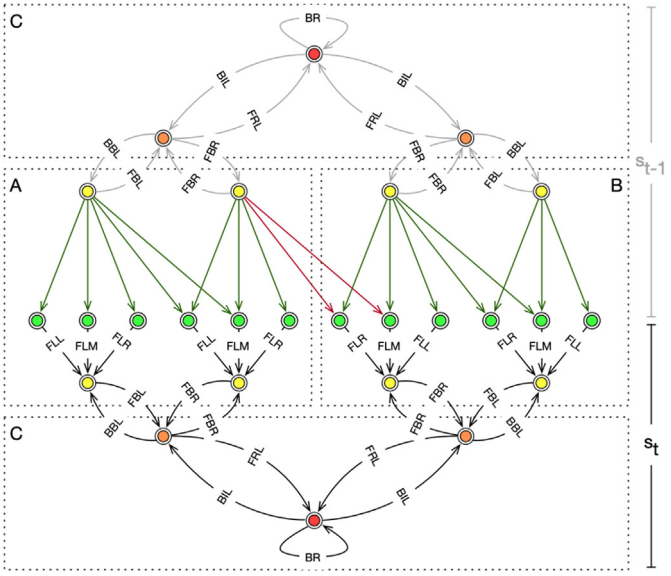


Fig. 11. The initialization of the next step S_t after solving the previous step S_{t-1} in direction Y. Direction Y, which belongs to the previous time step, and direction X at the bottom, which belongs to the next time step share the same vertices mirrored along the leaves for brevity of the operations.

4.4. Execution model

We propose that the distributed IGA-ADI solver should be modeled by assigning each directed edge E to the operation type P, which describes how the message $m_{t;p}$ should be constructed at the source vertex v_{src} in the current step S_t and how it should be consumed at the destination vertex v_{dst} in the next step S_{t+1}. Each production can also define what should happen after sending the messages to the target (*post-send*) as well as before (*pre-consume*) and after (*post-consume*) consuming them. This is not only an element of the algorithm but also an opportunity for the important memory optimizations. The responsibility of each of the vertices, given by the vertex program, is to follow the instructions in the received message $m_{t;p}$ to modify its local matrices and to send messages $m_{t+1;p}$ according to the instructions described by the corresponding, outgoing edges, to the destination vertices. Fig. 13 depicts a full lifecycle of each production P_t in the execution model proposed. It begins at the source vertex v_{src} of the edge the production P is attached to. The production extracts appropriate data from the source vertex v_{src} given by its position with respect to the destination vertex v_{dst} in the elimination tree. It uses this data to construct the

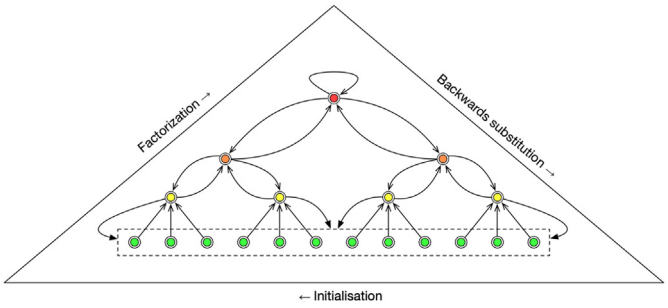


Fig. 12. IGA-ADI as an iterative graph algorithm executed on the elimination tree corresponding to a single direction. The graph is reused throughout the execution and the transitions between the directions and between the time steps take place in the initialization block from the data held at the branches in the terminal step of the last direction execution or from the external data set.

message to be sent along the edge E to the destination vertex v_{dst}. After the message is sent the production can clean up data which is no longer needed at the *post-send* stage. This marks the end of the step shifting the execution of the program to the destination vertex v_{dst}. Once the message arrives at the destination vertex production can prepare for consuming the data in the *pre-consume* stage. This often involves (re)initialization of the matrices so that at the next stage, consume, the data from the messages received can be used to initialize parts of the matrices respecting the position of the source vertex v_{src} with respect to the current vertex. Once all messages are consumed, the production can finally mutate the state of the vertex using the *post-consume* procedure, which is the last stage of its lifecycle. A single production spans across both sides of each edge, with *send* and *post-send* methods being executed at the sender side, and *pre-consume*, *consume* and *post-consume* at the receiver side, in two subsequent steps. This makes each production atomic in the sense that it defines a complete and independent data transformation of the algorithm. The execution should consist of a single block of instructions for each step which are calls to the productions lifecycle methods that first receive the messages to mutate the state of the local vertex using the production type bound to each message and then produce and send the messages using the production bound to the corresponding outgoing edge. The optimal execution model is also the push-based one, where the outgoing messages are first stored in the sender-side buffer, transmitted over the network asynchronously in batches, and then cached in the receiver-side message buffer. This guarantees that the execution of each step and consequently productions logic can never fail as no remote calls can take place within. All this greatly simplifies the implementation and testing of the algorithm.

5. IGA-ADI as a Pregel graph algorithm

To date there are no published implementations of IGA-FEM (including IGA-ADS and IGA-ADI) using Pregel and the idea of running complex scientific solvers on cloud computing frameworks is still fresh and mostly unexplored [38,39]. Contrary to what one might expect, the IGA-ADI algorithm can be implemented on top of the Pregel programming model almost intuitively. Let us first decompose each simulation into time steps and each time step into distinct phases running one after another. Fig. 14 presents how the following steps can form the Pregel IGA-ADI execution lifecycle:

- *Data initialization* – constructing an elimination tree that corresponds to the problem mesh and initializing the systems held at the leaves with the data from the distributed data set. Workers can use their local data as it is colocated in accordance with the IGA-ADI partitioner If this was not the case, a shuffle would be required. The elimination tree is constructed only once per simulation;
- *Vertical (Y) initialization* of the leaves of the elimination tree with the transposed solution to the horizontal direction stored at the branch vertices. Once this operation has been performed the graph represents the vertical ADI subproblem;
- *Horizontal (X) initialization* of the leaves of the elimination tree with the solution obtained in the previous time step (that is after finding the solution to the vertical direction which is stored at the branch vertices). Once this operation has been performed the graph represents the horizontal ADI subproblem;
- *Solution* of a single direction which translates into running graph-driven computations on the graph constructed during the data initialization, horizontal initialization or vertical initialization. The process is exactly the same regardless of the data source because the graph has exactly the same structure.

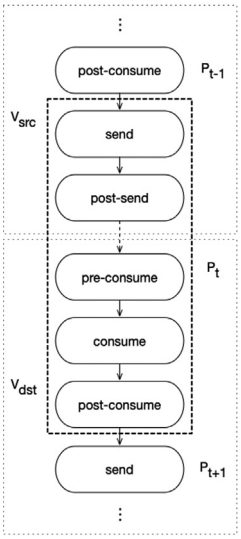


Fig. 13. The lifecycle of a production P_t in the context of the IGA-ADI algorithm. Note that P_t is executed both at the source vertex v_{src} and the destination vertex v_{dst}.

While the data is loaded synchronously, the results for each time step can be stored asynchronously as they become available. Therefore, the I/O-related delays which inevitably happen in between the time steps can be greatly reduced. Each phase can be represented as an isolated sequence of supersteps that are assembled together in a pipeline. This separation is only logical and does not influence the physical execution which runs on the joined sequence of supersteps, making no distinction between them. Each phase employs relative indexation of the supersteps which correspond to a global indexation with an offset contributed to from all prior phases. Fig. 15 depicts the first three supersteps of solving the direction phase of a regular 12×12 problem represented as a Pregel algorithm. In the initial superstep S_0 the leaf vertices 8 through 15 send the merge and eliminate leaves (FL) messages to their respective destinations and become inactive. In the next superstep S_1, vertices 4 through 7 become active. They merge the elements sent in the messages into their local matrices, respecting their origin described by the corresponding edges (L — left, M — middle or R — right). Next, they send merge and eliminate branch (FB) message with some

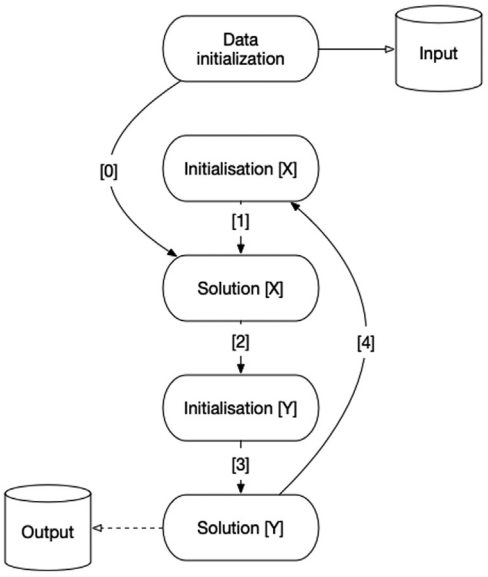


Fig. 14. Phases of the IGA-ADI Pregel simulation.

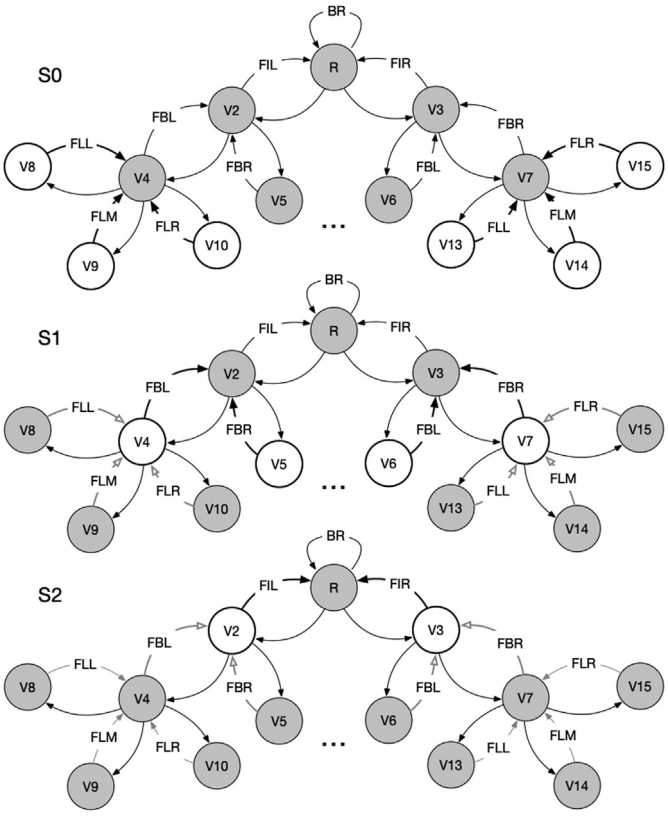


Fig. 15. The first three supersteps of IGA-ADI represented as a Pregel algorithm. Edges in bold indicate that the message has been sent along this edge in this step. White vertices are active while gray are inactive. Productions are denoted using abbreviations: FL — factorize leaves, FI — factorize interim, FR — factorize root, BR — backward substitute root, BI — backwards substitute interim, BB — backwards substitute branch, the rest are skipped for brevity.

of those updated values to vertex 2 or 3 and become inactive. In the next superstep S_2 vertices 2 and 3 merge the values received in the messages into their matrices respecting the origin (L — left, R — right) and become inactive. The process continues until the last message is sent, that is, until all edges (operations) have been traversed. Both horizontal and vertical initialization phases are executed in the corresponding way and consist of two supersteps necessary to execute a single production — initialize horizontal and initialize vertical respectively. This somewhat resembles the map and reduce steps, where the data is selected at the branches and consumed at the leaves. The data initialization is heavily influenced by the mechanics of the framework used and described in the next chapter, which covers the specific implementation details.

6. *Giraph* as the framework for implementing IGA-ADI

Giraph works on a small but powerful set of abstractions, all of which have a single, clearly defined responsibility in the simulation process. This framework provides a number of built-in implementations of these abstractions which by default are automatically configured for typical workloads. In the case of the alternating-directions IGA solver, however, a number of alternative implementations have to be provided which fall into two categories. The first group contains the abstractions that must be used to represent the alternating-direction IGA solver as a Giraph simulation:

- *Vertex* and *Edge*, which are the representation of the graph in memory. Both types are generic and can hold the values of any arbitrary type. The only requirement is that it needs to implement the interface *Writable*, which defines how the corresponding objects should be serialized and deserialized. Primitive types are generally preferred because of their lesser size and the fact that Giraph automatically selects the optimal representation based on *fastutil* memory-efficient *JVM* collections. That being said, complex types, such as the matrices required by alternating-directions IGA at the vertices, can also perform satisfactorily given appropriate implementation of the serializer and if the other components listed below are provided;
- *Computation*, which is the second core abstraction of *Giraph*. It defines the API of each superstep including a compute method that accepts a vertex and the messages sent to that vertex in the previous superstep. The implementation of this method is the vertex program, which may update the vertex value based on these messages or send new messages to other vertices to be consumed at the next superstep. The compute method is invoked globally for each vertex that is still active or that is going to receive messages in the corresponding superstep. Computations also allow the vertex programs to mutate the graph, but those are unnecessary for alternating-directions IGA;
- *MasterCompute* is the interface that allows the global behavior in between the supersteps to be defined. It collects any aggregated values from the vertices and makes the decision as to which computation should be used in the next superstep, if any. This is particularly useful for alternating-directions IGA simulations which consist of different phases that can be mapped into individual computations;
- *Aggregator*, which is used to transfer certain details about the current simulation state to the workers that execute the computations. This is to isolate the computations from the global process by parameterizing their execution. One of the important details is the local superstep number;
- *VertexInputFormat*, which defines how to convert a group of lines called the *InputSplit* into the corresponding vertices of the graph the computations need to be run on. It is executed only once per simulation, before the initial superstep;

- *EdgeInputFormat* that generates the edges between the vertices according the mesh and alternating-directions IGA method. It is executed only once per simulation, before the initial superstep;
- *TextVertexOutputFormat*, which defines how to represent each vertex as a line of text. It is executed after each time step which is after a multiplicity of a certain number of computations.

Alternating-directions IGA solver can be built directly on top of these abstractions in which case the productions are implemented as *Computation* subclasses and the implementation of the *MasterCompute* selects the *Computation* appropriate for the next superstep. In this approach, however, the control flow is bidirectional, because *Giraph* calls the solver code, but the solver code also calls the *Giraph* code back. This stands in opposition to the unidirectional flow in which the solver code never loses control over the execution (does not release the thread to execute foreign code before it finishes) and therefore can be run and tested independently of *Giraph* – without the need for using test doubles [40].

The unidirectional flow implies that the solver is a library, which can be made independent of but compatible with *Giraph*, by exposing its own API. This clear distinction between the solver and the framework the solver is executed on yields the same benefits to readability, testability and overall maintainability. The unidirectional control flow mandates the existence of a coordination layer which can be divided into the sub-layers that realize:

- *The simulation lifecycle* which is the upper layer of simulation whose primary responsibility is to select the appropriate implementation of the lower layer – the simulation phase (*Computation*) – for a given range of supersteps. This can be done by implementing appropriate subclasses of the *MasterCompute* class:
  - *InitialComputation*, which is run at the very first superstep when all vertices are active and no prior messages have been sent. It runs the productions only at the leaf vertices and deactivates all of them;
  - *InitializationComputation*, which sends the messages from the branches and consumes them at the leaves. No edges exist between the leaves as a memory optimization and this computation determines the destinations algebraically. *Giraph* can send messages to any vertex, not necessarily connected by a direct or even indirect edge;
  - *FactorizationComputation*, which is executed throughout the process of factorization, that is all the way up and down the elimination tree;
  - *TranspositionComputation*, which sends the messages from the branches and consumes them at the leaves. Similarly to *InitializationComputation* it uses no edges.
- *The superstep lifecycle*, which is the lower layer of each simulation. It can be implemented as a subclass of the *Computation* class. As opposed to the bidirectional approach mentioned above, each *Computation* corresponds to the phase which is used for one or many supersteps rather than a single superstep and a single production that is run in this superstep. Its primary responsibility is to unwrap the element from the vertex and pass each received message along with this element to the operation bound to this message. Before the superstep ends, it maps the updated element through the operations bound to the outgoing edges and sends the resulting messages along these edges to be consumed in the next superstep. This process is visualized in Fig. 16. The computations are resolved on the basis of the elimination tree and the current superstep.

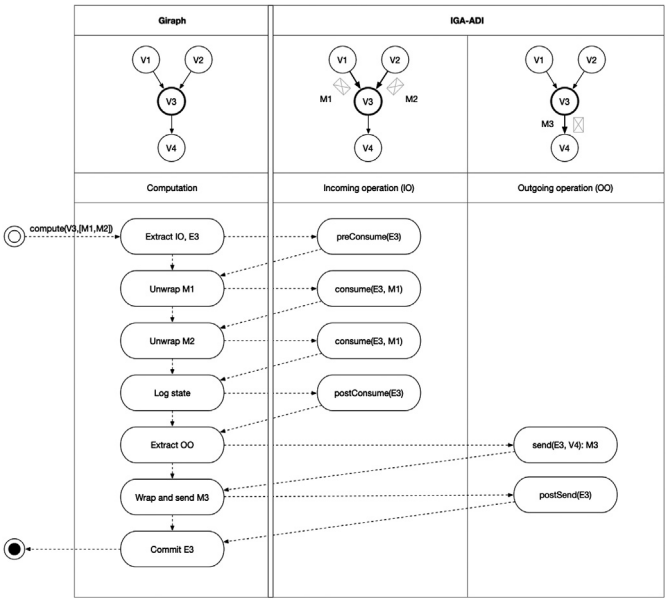


Fig. 16. Activity diagram presenting the relation between the *Computation* and *Operation*.

6.1. Vertex-centric model of the IGA-ADI distributed solver

Giraph provides a set of default implementations that match the requirements of the majority of typical simulations. That being said, the alternating-directions IGA solver requires custom implementations to achieve optimal performance. These are:

- *GraphPartitionerFactory* describes how to assign the vertices to the partitions and the partitions to the workers based on the available number of workers and a given number of partitions. The default strategy distributes the vertices randomly based on the hash of their key, which would lead to poor performance in alternating-directions IGA simulations. A custom partitioner that incorporates the knowledge of the structure of the elimination tree has to be used;
- *MessageSerializer*, which defines how to write a message into a stream of bytes and how to read it back into an object in the most efficient way. As each operation type in the proposed TLAV model of alternating-directions IGA has its message structure, there are also many serializers, each optimized to handle the corresponding message;
- *MessageStore* and *MessageStoreFactory* which are used to define the way *Giraph* stores the incoming and outgoing messages. This is particularly important if the values are of custom, complex types, like in the case of alternating-directions IGA, when *Giraph* uses a byte array to store all objects in the serialized form. This happens when the messages need to be transferred through the network and internally, which brings unnecessary overhead. This overhead would be significant but unnecessary for alternating directions IGA, which can avoid most network communication through proper partitioning. Therefore, we modify the *Giraph* source code to bypass the serialization for any intra-worker exchange of vertices.

Fig. 17 depicts a simplified class diagram of the core of the TLAV-compatible alternating-directions IGA solver. The *IgaOperation* interface provides the lifecycle, which the *Giraph* coordination layer can utilize. It is parameterized by a single message that it knows how to send and consume. Because each message is used only by a single operation, it can be evolved freely, which is the

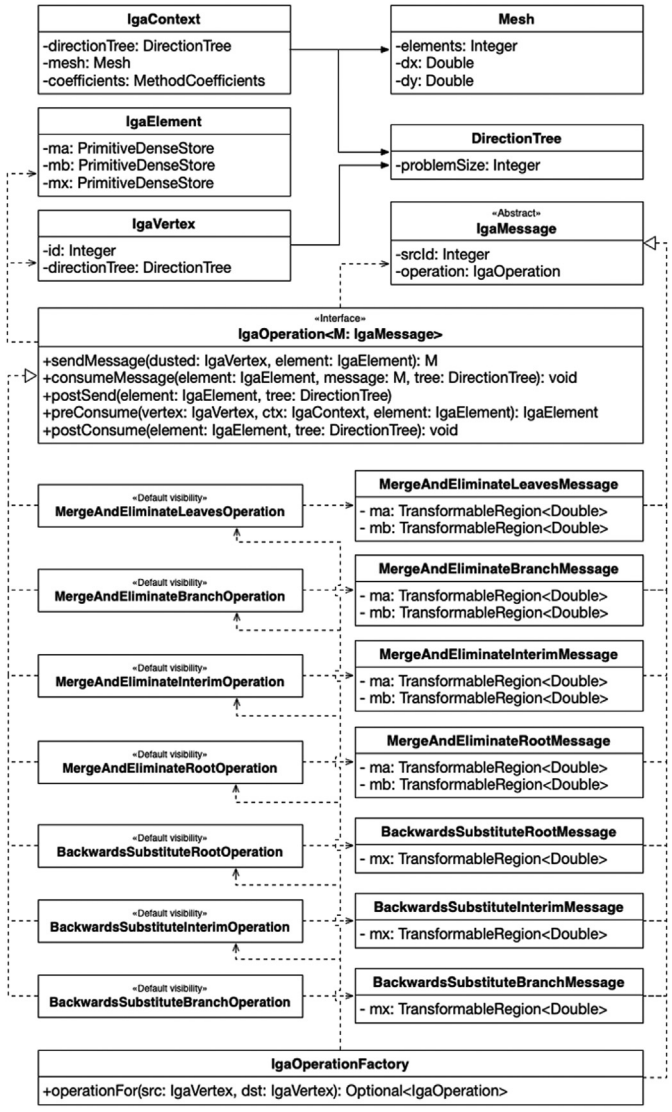


Fig. 17. The class diagram of the core of the alternating-directions IGA *Giraph* solver. This model is independent of *Giraph* and compatible with most TLAV frameworks.

benefit of making each operation span across two supersteps on both sides of each edge. Its implementation details are hidden from other operations and the rest of the application. This is the case because the *Giraph* orchestration layer does not make any distinction between particular operations as it refers to all of them only by the unified *IgaOperation* interface.

More specifically, all *IgaOperation* implementations have package-private visibility. They can be created only through the *IgaOperationFactory*, which accepts the source and destination vertices and returns an appropriate implementation of the operation (hidden under the common interface) or no value if input vertices are not directly related. Algorithm 1 describes the logic of this assignment.

6.2. Running the solver

Fig. 18 presents a high-level representation of the complete *Giraph* simulation lifecycle from the moment it is started by the user to the moment it terminates. *Giraph* is typically launched by invoking a YARN or *Hadoop* client binaries supplemented with the so-called fat jar which contains the user program code along with

Algorithm 1 The assignment of operations to the edges

Require: *src*,*dst*

```
if src.is(LeafVertex) AND dst.is(BranchVertex) then
    return MergeAndEliminateLeaves
if src.is(BranchVertex) AND dst.is(InterimVertex) then
    return MergeAndEliminateBranch
if src.is(InterimVertex) AND dst.is(RootVertex) then
    return MergeAndEliminateRoot
if src.is(InterimVertex) AND dst.is(InterimVertex) then
    if src.after(dst) then
        return MergeAndEliminateInterim
    else
        return BackwardsSubstituteInterim
if src.is(RootVertex) AND dst.is(InterimVertex) then
    return BackwardsSubstituteRoot
if src.is(InterimVertex) AND dst.is(BranchVertex) then
    return BackwardsSubstituteBranch
```

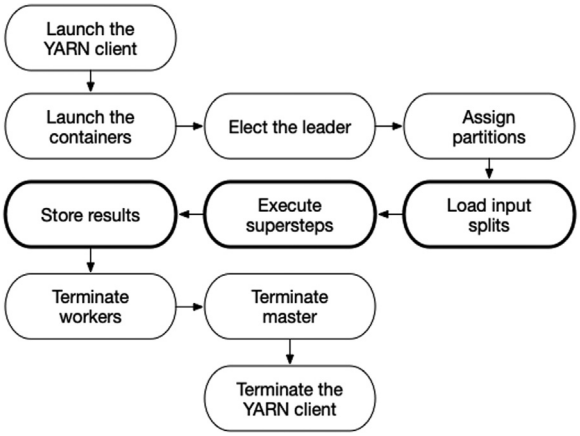


Fig. 18. High-level activity diagram of each *Giraph* simulation. The thick border indicates the activities that execute user-defined computation code.

all the required dependencies. The artifact is automatically copied into the ephemeral HDFS directory from where it is accessible by all workers. *Giraph* launches two additional containers on top of the number of workers. One of them is used to launch all other containers. The second is the *Giraph* master which is elected from the remaining containers. Capacity planning has to include the resources dedicated to the master which are identical to any other worker and the launcher container. Most *Giraph* programs require little configuration which is the reason they are launched using the default *GiraphRunner* class with all arguments provided in the command line using predefined configuration options. The alternating-directions IGA solver, however, needs to specify many more options compared to the required minimum which is computation, input and output format classes. In order to simplify the execution, a custom *Hadoop Tool* implementation – *IgaSolverTool* – is used that configures *Giraph* to solve alternating-directions IGA problems and exposes only the simulation-specific parameters that can be safely adjusted. This way the solver can be launched using the following command:

```
yarn jar iga-adi-giraph.jar
edu.agh.iga.adi.giraph.IgaSolverTool
[-s <arg>] [-d <arg>] [-e <arg>] [-w <arg>]
[-c <arg>] [-m <arg>] [-t <arg>] [-p <arg>]
--config giraph.zkList="<ZOOKEEPER_HOST>:2181"
```

where

- -s, –steps <arg> (the number of time steps to execute)
- -d, –delta <arg> (the resolution of the time steps)
- -e, –problem-size <arg> (the number of elements in the mesh)
- -w, –workers <arg> (the number of workers used)
- -c, –cores <arg> (the number of cores per worker)
- -m, –memory <arg> (the amount of memory per worker in Gigabytes)
- -t, –input-type <arg> (the type of input - bitmap or coefficients)
- -p, –problem <arg> (the name of the problem)

7. Numerical results

The solver is written in Java 11 running on JVM distribution provided by Google Cloud Dataproc 1.3-debian10 cluster with Hadoop 2.9.2. Apache Giraph 1.2.0 was used as a framework, along with popular open-source libraries:

- OjAlgo 47.3.1, for efficient and portable linear algebra computations;
- Apache Commons Lang 3.9, for general-purpose utility methods;
- Jcommander 1.78 for robust command-line interface;
- AssertJ 3.13.0 with Junit 5.5.1 for unit testing;
- Hadoop Minicluster 2.9.2, for integration testing.

The project uses Maven 3.8.1 for build automation. We chose this technology stack to ensure simplicity of implementation, ease of integration with other software, reliability of execution, and portability between clusters. We were also following established industry practices — leveraging Git for version control with feature branches and quality-reviewed pull requests, refactoring, test-driven development, and continuous integration with a fast feedback loop.

In order to validate the correctness we use the solver to solve the Cahn–Hilliard problem [15–17] which has various applications in material science, phase-field simulations or computational oncology [41]. The details of the formulation for the Cahn–Hilliard equations suitable for IGA-ADI solver are described in [41].

In our numerical example we consider a domain $\Omega \subset R^2$ with a sufficiently smooth boundary $\Gamma = \partial\Omega$. Ω contains a binary mixture and we denote the concentration of one of its components by c. The temporal evolution of the mixture is governed by the fourth-order Cahn–Hilliard phase-field equation

$$\begin{aligned} \frac{dc}{dt} &= \nabla \cdot (M(c)\nabla(F(c) - \Delta c)) \quad \text{in } \Omega \times [0; T], \\ c &= g \quad \text{in } \Gamma_g, \\ \eta &= s \quad \text{in } \Gamma_s. \end{aligned} \tag{25}$$

Here $M(c)$ is the mobility, $F(c)$ represents the chemical potential of a regular solution in the absence of interfaces, T is the length of the time interval, and $g : \Gamma_g \longrightarrow \mathbb{R}$, $s : \Gamma_s \longrightarrow \mathbb{R}$ define the boundary conditions.

In order to study the capabilities of our algorithm, we simulate typical scenarios for the Cahn–Hilliard equation, namely the random initial distribution of the concentration [16,26]. In this case, as the initial condition we use

$$c(\mathbf{x}) = \bar{c} + r, \tag{26}$$

where $\bar{c}$ is the volume fraction constant and r is a random variable with uniform distribution. In the following examples we use a random r with uniform distribution in $[-0.05; 0.05]$. The domain is a unit square $\Omega = (0, 1)^2$. We also impose periodic boundary conditions at all boundaries of the domain.

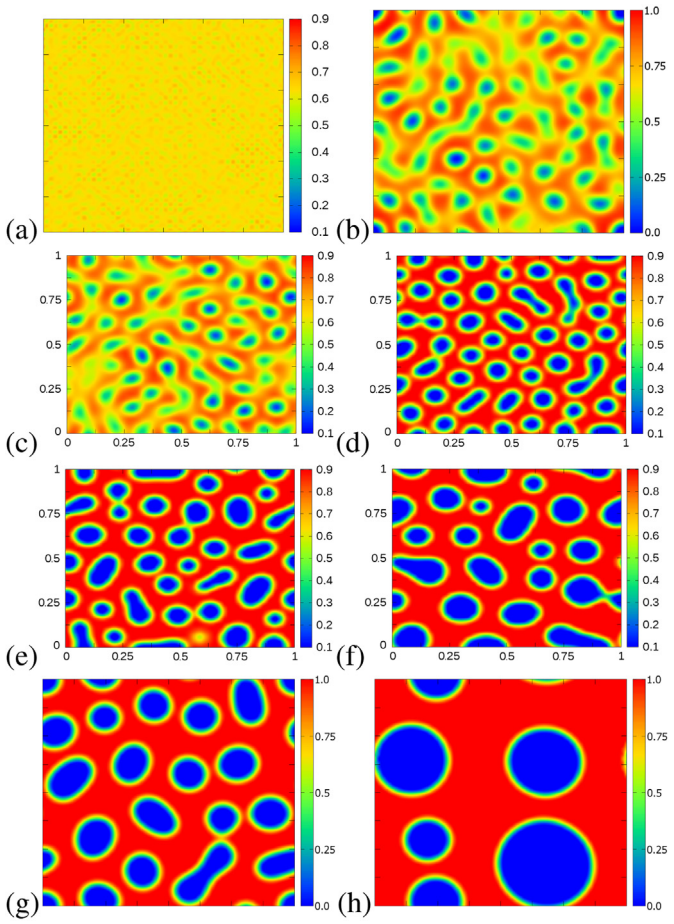


Fig. 19. The evolution of the concentration from a randomly perturbed initial condition on a mesh comprised of 512^2 quadratic C^1 elements for $\alpha = 3000$ and $\bar{c} = 0.63$. (a) Initial state. (b) Time moment $t = 3 \times 10^{-6}$. (c) Time moment $t = 6 \times 10^{-6}$. (d) Time moment $t = 1.44 \times 10^{-5}$. (e) Time moment $t = 4.3 \times 10^{-5}$. (f) Time moment $t = 8.9 \times 10^{-5}$. (g) Time moment $t = 1.36 \times 10^{-4}$. (h) Final time moment $t = 1.5 \times 10^{-3}$.

The datasets used in our experiments were generated in place. However, cloud providers generally do not charge for data ingress. Most cloud providers host large data sets themselves (like Google Cloud Public Dataset), sometimes in cooperation with academia. Storing data at rest in the cloud is known to be cost-effective and secure compared to hosting data on-premise, which is why modern data tend to reside in the cloud, to begin with. For example, it can be collected from sensors and accumulated there over time. In big data frameworks like Hadoop, once data is loaded into the cluster, computations are sent to data, not data to the computations, which was the major paradigm shift boosting the efficiency of working with data. The data from our large-scale experiments were post-processed to various forms in the same cluster, and only the results in a human-readable format were downloaded — like summaries, images, or animations.

Fig. 19 shows snapshots of the solution computed using the setup from [16] with $\alpha = \frac{1}{3\lambda} = 3000$ and $\bar{c} = 0.63$. The mobility is dependent on the concentration and is taken as (27) with $D = 1$

$$M(c) = Dc(1 - c). \tag{27}$$

For the chemical potential, we use the logarithmic function

$$F(c) = \frac{1}{2\theta} \log \frac{c}{1 - c} + 1 - 2c. \tag{28}$$

The mesh has 512^2 quadratic C^1 elements. Simulations on larger meshes or which use higher-order basis functions provided similar results. For verification of our ADS solver, we compare this simulations with the results described in [41]. We can see identical behavior of the phase-field simulations.

8. Scalability of the distributed memory IGA-ADI solver

The tests were carried out using the GCP *Cloud Dataproc* service, which is a cloud-native SAAS [42] *Hadoop* installation comparable to EMR from AWS or *HDInsight* from *Azure*, but with a more favorable pricing model with $300 free credit to be used within 12 months of trial, ideal for conducting this research.

As the scalability of the TLAV alternating-directions IGA solver needed to be evaluated within the free subscription, the simulations were restricted to 256 threads both because of the resource limits applicable to this type of subscription and the restricted budget. We use cluster with 16x n1-standard-16 nodes, with 256 total VCPU and 960 GB of memory. Our cluster consists of a large number of standard virtual machines that all host a single *Giraph* worker. This is in contrast to how practical simulations should be performed but allows the impact of the distributed systems phenomena to be observed, which limit the scalability of the solver with the available cores. These simulations approximate the horizontal scalability of the solver.

Each of the VCPU in the N1-series machines offered by GCP is in fact a virtual Hyper-thread which is run on a single physical core and displays lower performance and predictability. This is common for all cloud providers and CPU manufacturers [43]. The physical CPU for all N1 instances may vary from cluster to cluster but always feature *Intel Skylake* (*Intel Xeon Scalable Processor*, 2.0/2.7/3.5 GHz), *Broadwell* (*Intel Xeon E5, 2.2/2.8/3.7 GHz*), *Haswell* (*Intel Xeon E5 v3*, 2.2/2.8/3.8 GHz), *Sandy Bridge* (*Intel Xeon E5*, 2.6/3.2/3.6 GHz) or *Ivy Bridge* (*Intel Xeon E5 v2*, 2.2/3.1/3.5 GHz), depending on the types of the nodes used in this cluster. Moreover, each processor has a variable clock that depends on the actual core usage which can potentially be utilized by multiple workloads. These are some of many factors which contribute to the varying performance of the cluster [43]. The network between the machines is 16 GBps or 32 GBps, which in both cases is enough to prevent network saturation for this implementation of the alternating-directions IGA solver. The *Dataproc* version was 1.4 with *Debian 9*, *Hadoop* 2.9 and *Spark* 2.4.

8.1. Serial scalability

The numerical properties of the alternating-directions IGA *Giraph* implementation were first evaluated on a cluster with a single worker – (cluster with one n1-standard-96 node with 96 VCPU and 360 GB of memory) – in search of the most appropriate *Giraph* parameters listed above as well as optimization prospects. The results of the final run, performed after incorporating all of these measures, are presented in Fig. 20. The algorithm displays linear scalability with respect to the number of elements in the mesh, with low residual error even in the multi-tenant execution environment with varying performance. This proves the implementation satisfies the serial execution numerical objective. The biggest problem solved with a single VCPU had $150,994,944$ degrees of freedom, which took about one hour to complete. An excess of 92% of this time was spent on the initialization of the systems of linear equations. This promises massive returns from further parallelization as this phase contains thousands of independent tasks.

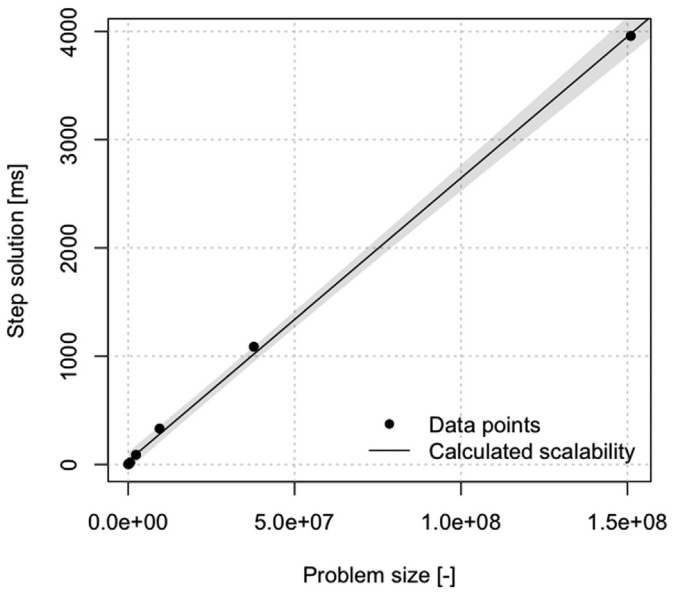


Fig. 20. Serial scalability of the TLAV alternating-directions IGA solver implemented on top of *Giraph* framework. The gray area around the fitted line indicates the 99% confidence range.

8.2. Strong scalability

Let us start by investigating the numerical behavior of the solver when run on the cluster featured up to 16 nodes with 16 vCores each. The number of threads for a single worker node was fixed and set to the maximum value, which aims to represent horizontal scaling. Each of the worker nodes had 36 GB of heap. Fig. 21 contains strong scalability charts drawn for 4 different problem sizes which were executed on 1 to 256 vCores with 2^n interval.

The results confirm that the solver computation times can be reduced by scaling either up and out or by a combination of the two. If we select $N = 12888^2$ elements in the computational mesh and the maximum number of threads $p = 256$, for which the initialization and total speedup is $S_i(256) \approx 179.89$ and $S_t(256) \approx 80.81$, the theoretical Amdahl's limits are:

$$P_t = \frac{256 * (S_t(256) - 1)}{(256 - 1) * S_t(256)} = \frac{256 * (80.81 - 1)}{(256 - 1) * 80.81} \approx 0.991$$

$$S_t = \lim_{n \to \infty} \frac{1}{(1 - 0.991) + \frac{0.991}{n}} \approx 111.11$$

$$P_i = \frac{256 * (S_i(256) - 1)}{(256 - 1) * S_i(256)} = \frac{256 * (179.89 - 1)}{(256 - 1) * 179.89} \approx 0.998$$

$$S_i = \lim_{n \to \infty} \frac{1}{(1 - 0.998) + \frac{0.998}{n}} \approx 500$$

Here, $S_t(256)$ denotes the speedup of the entire solver when using 256 cores, and $S_i(256)$ represents the speedup of the initialization when using 256 cores. $P_t(256)$ means the percentage of the total algorithm that benefits from parallelization using 256 cores. $P_i(256)$ means the percentage of the initialization algorithm that benefits from parallelization using 256 cores.

The value of total speedup limit is close to three times as high as the one derived from the single-node experiment for the same problem size.

The speedup was computed using the results obtained from running the scalability experiment using 256 threads distributed across 16 worker nodes, and as such included all phenomena with a potentially deteriorative effect on scalability. In particular, using

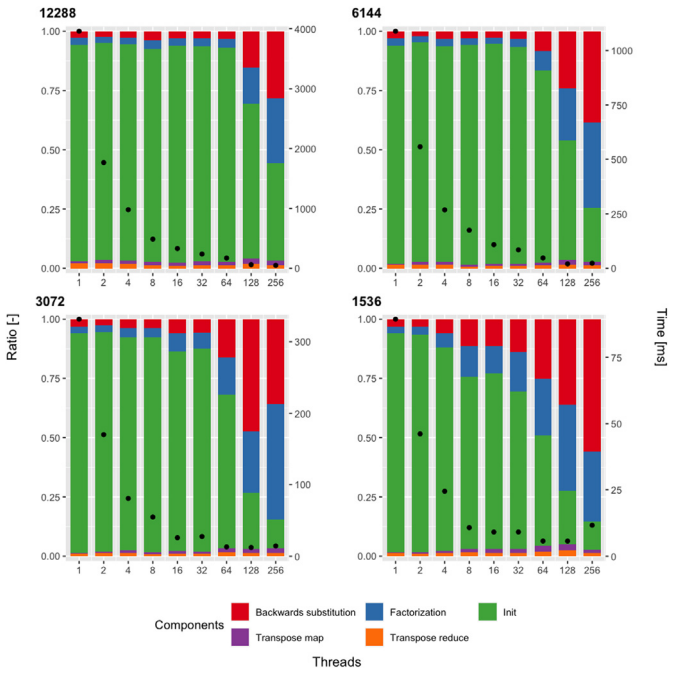


Fig. 21. Strong scalability of the TLAV alternating-directions IGA solver implemented on top of the *Giraph* framework and run on multi-node cluster. The stacked bar charts behind the data points represent the contributions of the individual phases of the solution process. The numbers in the top-left corners of each chart indicate the counts of elements in either dimension of the mesh.

4 nodes with 16 threads rather than one node with 64 threads increased the computation times only by about 10%, which might indicate that the network-related phenomena of distributed communication have a limited effect on the solver operation. Typical *Giraph* simulations would feature a few strong nodes rather than many weak ones.

8.3. Weak scalability

While we can use the distributed version of the alternating-directions IGA solver to reduce the computation times by about two orders of magnitude for a fixed problem size, it is not designed for this purpose. The primary objective is not to reduce the computation times for a fixed problem size but to reduce the rate at which the computation times increase with the size of the problem. Ideal weak scalability means that we can retain the same computation time for problems of all sizes if the size of the problem per computing unit is constant. Therefore, let us investigate The weak scalability is the variation in solution times with the number of threads for a fixed problem size per thread. Gustaffson's law can give this relation.

If we denote the workload as W and the fraction of it that can benefit from parallelization as p, and the fraction which cannot as $1-p$, then

$$W = (1-p)W + pW. \tag{29}$$

Assuming that we observe the speedup s after increasing the number of available compute resources, the theoretical workload $W(s)$ can be stated as

$$W(s) = (1-p)W + spW \tag{30}$$

because the parallelizable portion of the workload pW is affected by the speedup, and the non-parallelizable part of the workload $(1-p)W$ is not. Then Gustaffson's law becomes

$$S(s) = \frac{TW(s)}{TW} = \frac{W(s)}{W} = 1 - p + sp \tag{31}$$

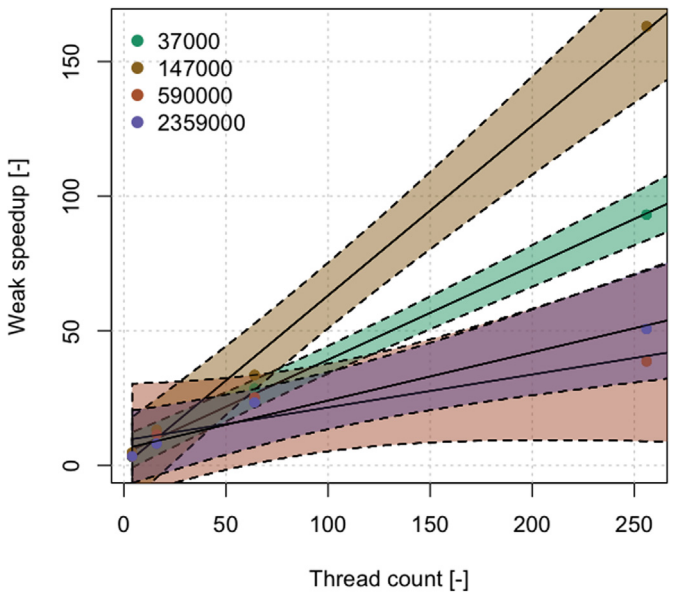


Fig. 22. Weak scalability of the TLAV alternating-directions IGA solver implemented on top of the *Giraph* framework run on the multi-node cluster for four values of elements per thread. The polygons correspond to the 95% confidence bounds.

where $S(s)$ is the speedup in latency and T is fixed execution time. We can also define scaling efficiency as

$$e = \frac{S(s)}{s} = \frac{1-p}{s} + p. \tag{32}$$

Due to the simplified design of the alternating-directions IGA solver, which can only work on square meshes, We test the scalability for problem sizes N for which $N \in 3 * 2^t$ where $t \in \mathbb{N} \geq 2$ that is 12^2, 24^2, 48^2, 96^2, 192^2, 384^2, 768^2, 1536^2, 3072^2, 6144^2, 12288^2, 24576^2 and so on. Consequently, if the number of unknowns per thread needs to be fixed, only 4^t threads can be used, where $t \in \mathbb{N}$.

Fig. 22 presents the weak scalability of the alternating-directions IGA solver computed for the single step of the simulation. These results are based only on a few data points. The residual error is relatively large but still fit for a linear model. It proves that it is possible to maintain only a linear increase in computation times for quadratic increments in problem size by maintaining a fixed problem size per thread, that is, by increasing the number of threads to solve bigger problems. The rate of the increase is inversely proportional to the slope of the fitted line. For 45°, there is no increase in computational times, and the solver has perfect weak scalability characteristics. For 147,000 elements per thread, it is about 27°, for which the efficiency is about 60%, meaning the computational times increase about 1.6 times when the problem size is squared.

8.4. Universal scalability law

The Universal Scalability Law (USL) defines a unified scalability model that accounts for all scalability phenomena and correctly predicts that the performance of the system can deteriorate with adding more compute resources starting from a certain point. This relation could be observed in the strong scalability experiments. The USL defines the relative capacity (normalized throughput) of the system $C(N)$ as [44]

$$C(N) = \frac{\gamma N}{1 + \alpha(N-1) + \beta N(N-1)} \tag{33}$$

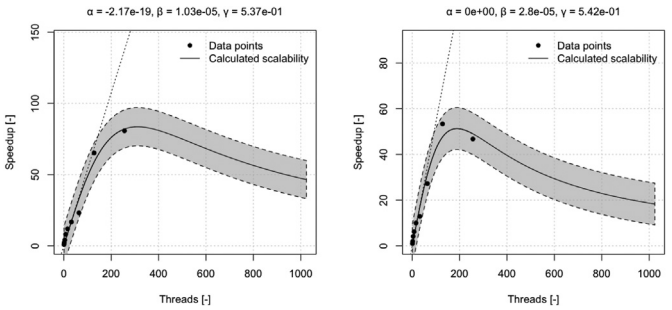


Fig. 23. 12288^2 elements and 6144^2 elements. The USL strong scalability models for different problem sizes of the TLAV alternating-directions IGA solver implemented on top of the *Giraph* framework computed on multi-node cluster. The shaded areas indicate 99% confidence intervals. The coefficients of the models are given on top of each chart.

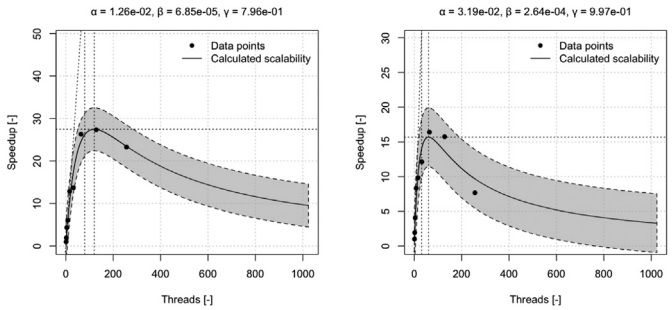


Fig. 24. 3072^2 elements and 1536^2 elements. The USL strong scalability models for different problem sizes of the TLAV alternating-directions IGA solver implemented on top of the *Giraph* framework computed on multi-node cluster. The shaded areas indicate 99% confidence intervals. The coefficients of the models are given on top of each chart.

where N is a load parameter and three coefficients in the denominator are

- 1 – concurrency – which represents the linear relation between the capacity and load in the system that would be displayed if there was no cost to scaling;
- $\alpha(N-1)$ – contention – which models the impact of sharing limited resources in the system or intrinsic serializability of the process. This relation is assumed to be linearly dependent on the load, which means that the capacity can only grow with the load but the increments become infinitely small. The contention parameter α determines how fast this happens and what the corresponding limit of system capacity is;
- $\beta N(N-1)$ – coherency – which models the consequences of the latencies inherent in distributed data transfers in point-to-point communication between the components of the system in order to ensure consistency and global progress. The coherency parameter β determines how fast the load starts to bring negative returns in capacity and what the peak capacity of the system is. This parameter is multiplied by the number of processing units squared N^2, therefore even small values of β can have significant influence on the capacity for higher load values. The capacity begins to deteriorate quickly starting from a certain value of the load and then slowly approaches a certain lower capacity limit.

In the case of describing the performance of the solver the capacity $C(N)$ is equal to the normalized execution time (speedup) $S(N) = \frac{T(1)}{T(N)}$ and the load parameter N is equal to the number of threads. In most practical simulations it is difficult or impossible to experimentally determine the performance of the system in a serial execution due to the memory or time limits.

In such a case γ can be used to represent this missing point of reference. When both $\alpha = 0$ and $\beta = 0$ the speedup is linear. When $\alpha > 0$ and $\beta = 0$ the USL becomes equivalent to Amdahl's law. When both $\alpha, \beta \geq 0$ the USL becomes an extension of Amdahl's law which correctly predicts the retrograde scalability present in many distributed systems. α, β and γ can be found using a non-linear regression. The values were found using the package *usl* for *R* [44].

Fig. 23 and Fig. 24 contain diagrams that depict the recorded speedups for problems of a different size solved on multi-node cluster and the corresponding USL scalability models used to draw the scalability curve of the solver up to 1024 threads. These models correctly predict that starting from a certain number of threads, adding more of them does not improve the capacity of the system but decreases it instead. This is the effect of non-zero coherency cost coefficient β, which was not present in either

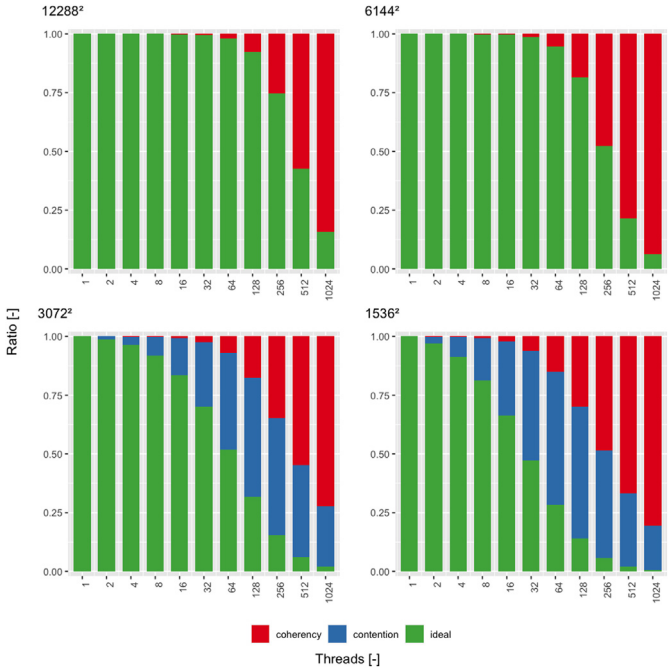


Fig. 25. The relationship between the parameters of the USL model of the TLAV alternating-directions IGA solver implemented on top of *Giraph* framework for different problem sizes solved on multi-node cluster.

Amdahl's or Gustaffson's laws. For this reason the theoretical speedup limit is lower than that computed from the Amdahl's law. For example, while the Amdahl's scalability limit for 12288^2 elements was about 111, the USL model predicts the scalability to have its peak at roughly 85 which is for about 310 threads. This is because the influence of the coherency parameter β is present, which prevents the chart from ever reaching the limit apparent from the Amdahl's law and even moving away from it for a larger number of threads. Moreover, the computation times do not seem to be influenced by the contention modeled by α, at least for larger problems, which might indicate that it is the cost of coherency alone that reduces the speedup when alternating-directions IGA is scaled out.

For this reason, the theoretical speedup limit computed through fitting the USL curve to all data points is close to that computed from a single data point.

Fig. 25 presents the breakdown of the percentage of the total simulation time into components for the strong scalability experiments run on the multi-node cluster. The first component,

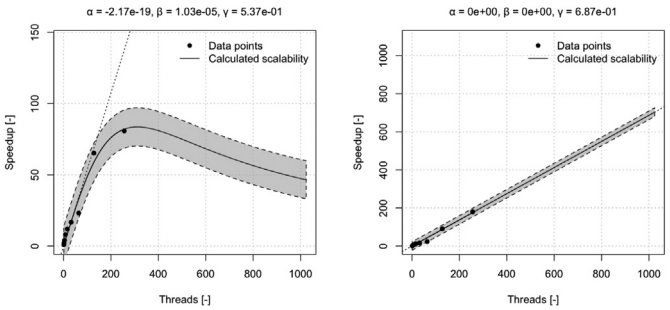


Fig. 26. Total speedup and initialization speedup. The USL models for individual phases of finding the solution using the TLAV alternating-directions IGA solver implemented on top of the *Giraph* framework for a problem with 12288^2 elements run on multi-node cluster.

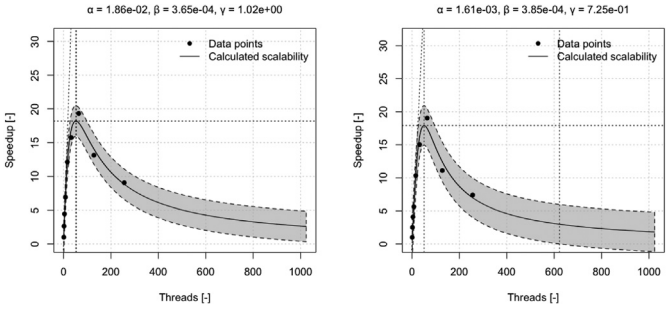


Fig. 27. Factorization and backwards substitution speedup. The USL models for individual phases of finding the solution using the TLAV alternating-directions IGA solver implemented on top of the *Giraph* framework for a problem with 12288^2 elements run on multi-node cluster.

marked in green, corresponds to the actual computation time necessary to solve the problem assuming ideal parallelization. The second component, marked in blue, represent the contention overhead modeled by the α parameter from the USL scalability model. The third component, marked in red, depicts the influence of coherency denoted by β from the USL scalability model. Note that the bar heights do not represent scalability and one should not expect the green bar to be cut in half in order to double the number of available threads. In fact, it is the opposite — if it does not fall and dominate the relation then the scalability is near linear. This type of diagram is a good visualization of the computation efficiency that is represented by the height of the green bar.

It is also apparent that in the case of vertical scaling it is primarily the cost of contention which limits scalability — with an exception of the results for 1536^2, which might be the deficiency of the USL model or inaccuracy of the results. In the case of horizontal scaling, however, it is the cost of coherency, at least for larger problems. Moreover, while the returns from vertical scaling decrease almost linearly when applying additional threads, the returns from horizontal scaling are nearly the same at first and then fall drastically.

Let us focus on the scalability of individual phases of finding the solution — initialization, factorization, backwards substitution, as well as the transposition map and reduce steps grouped by a problem size and cluster type (see Fig. 26, Fig. 27, and Fig. 28) The total speedup chart presented previously is also present in each group for convenience. Fig. 28 presents the charts for the problem with 12288^2 elements solved on multi-node cluster.

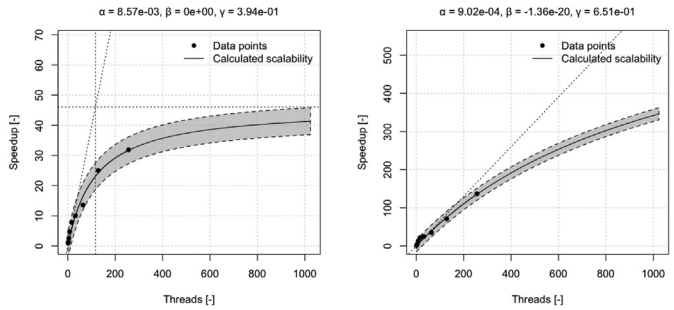


Fig. 28. Transposition map and reduce speedup. The USL models for individual phases of finding the solution using the TLAV alternating-directions IGA solver implemented on top of the *Giraph* framework for a problem with 12288^2 elements run on multi-node cluster.

There are phases which scale linearly or almost linearly and with good efficiency but there are also phases which can be considered the bottleneck for alternating-directions IGA scalability. The initialization phase is perfectly parallelizable with about 80% efficiency. This matches our expectations as in this phase the worker nodes initialize individual matrices in isolation so there is no need for communication and no coherency penalty. In the transposition reduction step the workers initialize the element matrices with data from the perpendicular direction, as a result of which this phase exhibits good scalability for similar reasons. On the other hand, the factorization and backwards substitution phases scale well but only to about 32 threads.

Their USL model is very similar, which is not surprising given they have very similar execution schemes. They feature a strong coherency component which leads to an increase in the computation times when new threads are added. It is only these two phases that together shape the total speedup model to display retrograde scalability. This claim can be supported by 21, which presents the percentage of the solution time that each of these phases take. The initialization phase might dominate the solution time for larger problems and for fewer threads but the factorization and backwards substitution phases clearly become dominant for a larger number of threads. Therefore, the influence of their USL model becomes stronger with the increasing number of threads leading to the final shape of the summary model. The reason why these two phases display poor or retrograde scalability is that the elimination graph is a binary tree and the operations are executed in parallel batches, where one batch aggregates all operations from a single level of the tree. We partitioned the elimination tree so that the tip of the elimination tree was handled by only a single worker to avoid expensive network transfers with a performance that is hard to predict. Consequently, the height of the top subtree increases with the number of threads and decreases with the problem size. Fewer vertices in batches increase the percentage of computational effort dedicated to synchronization as there is not enough work to amortize this small cost which accumulates over hundreds of supersteps. Lastly, the transposition mapping phase is only congestion-restricted, which is because the nodes exchange the messages through the limited-capacity network with maximal throughput. The transposition reduction step scales well because it is similar to the initialization step but with less computational pressure.

8.5. Comparison to other implementations

We compare the scalability of our cloud-native solver to the traditional implementation from [41] executed on a cluster node

with two Intel Xeon E5-2680v4 CPUs with 14 cores 2.40 GHz each. The execution times summarized in Table 2 there show near-linear scalability when increasing the mesh size. For 160×160 mesh, we have 10 s spent on the initialization (integration) and 20 s spent on the execution of the solver. The solver discussed in [41] relies on the parallelization of the integration process, and the factorization is performed in a sequential manner. Our cloud-native solver on a ten times larger mesh with 1536×1536 elements takes less than 0.1s using one core, according to right bottom panel in Fig. 21.

On the other hand, comparing our solver to a highly-optimized finite-difference sequential implementation of [45], using one core of i7-6600U processor, we can see from Table 1 there, that the one-time step of the mesh 800×800 is solved there within $7.0/100 = 0.07$s. Assuming linear scaling of the solver, we can predict 0.15s for mesh size of 1536×1536, using one core.

Summing up, the execution time of our cloud-native solver is comparable to a highly-optimized sequential solver using a lower-order finite difference method (0.15s versus 0.1s). Moreover, our solver can be two orders of magnitude faster when increasing the computing cores number.

9. Conclusions

The biggest problem solved on multi-node cluster (B) featured 2,415,919,104 degrees of freedom – or 49,152 elements in one direction – which indicates that the TLAV version of alternating-directions IGA can solve problems of any practical size by applying the appropriate number of processing nodes, which may be commodity hardware provisioned in the public cloud. In this particular instance solving a single time step takes about 10 min, which is clearly unacceptable in realistic applications, where there are thousands of time steps to compute in sequence, even using the implicit time-stepping method presented in this study. That being said, the scalability models computed for smaller problem sizes indicate that this time could be substantially reduced by applying additional resources as roughly 95% of the total computation time in this experiment was spent on the process of the initialization of the systems of linear equations which displays linear scalability that could be better exploited. The generation of the system involves the integration of the right-hand side vectors with suitable quadratures

$$\int B^x_{i,2}(x)B^x_{j,2}(x)F(x,y)dxdy = \sum_{i=1,\ldots,N_q} w_q B^x_{i,2}(x_q)B^y_{j,2}(y_q)F(x_q,y_q) \tag{34}$$

These can be sped up using the sum factorization technique or by using further decomposition of the integrals into basic undividable tasks and following the trace theory scheduler, [46,47] The cluster which could deliver optimal performance should be built from a few powerful nodes rather than from a large number of less powerful ones. This should allow the benefits of both methods of scaling to be exploited better in order to combat the influence of congestion and coherency and increase the efficiency. Moreover, the probability of failure increases with the size of the cluster which, while being gracefully handled by this version of the solver, also has an impact on computation efficiency. This plays a particularly important role in the case of opportunistic computations where the probability of eviction also increases with the number of nodes. Such experiments, however, could not be performed under the constraint of available funds, as the ideal compute cluster for such a task would need to feature high-cpu nodes, such as *c2-standard-60* compute-optimized virtual machines, each hosting 60 VCPU and 240 GB of memory priced at more than $3 per hour. Assembling a cluster of just 10 nodes with a total of 600 VCPU and 2.4 TB of memory costs an excess of $30 per hour which, multiplied by the number of necessary simulations, totals at considerable amount. Even if the funds were not the issue, each cloud provider would limit the availability of hardware for individual customers by introducing quotas. While it is possible to request an upgrade for most of them – like the CPU quota relevant for these simulations, which might prevent us from creating the cluster mentioned above – it is ultimately the task of the cloud provider to decide whether this constitutes a reasonable use case and how high the quota should be. That being said, businesses or universities typically use the quotas as a tool to control the costs, keeping the limits within the reasonable levels to prevent accidental generation of large pay-per-use amounts, and cloud providers use that data mainly for capacity planning.

Compute-intensive tasks present in scientific simulations usually require more computational power than memory. Cloud providers offer machines with a higher number of CPU threads, but they also feature a higher amount of memory than necessary. In the case of GCP, each VCPU is bound to at least 1 GB or RAM regardless of the type of configuration. This increases the costs of the cluster. One possible way to remediate this deficiency is to schedule low-cpu but high-memory jobs along with high-cpu and low-memory jobs, which is supported by YARN. From this perspective GPU-based clusters with thousands of threads per node seem superior, but they have their own limitations [48], and are out of the scope of this study as no general-purpose cloud-native graph-processing framework acclaimed and maintained by the open-source community that could utilize them has been proposed to date [37].

The results obtained also indicate that the predictability of running the computations in the cloud is lower than in the cluster hosed on-premises for a number of reasons. First of all, clouds by definition are multi-tenant and there can be many concurrent workloads scheduled on the same hardware, especially if smaller virtual machine types are used. While cloud providers employ a number of mechanisms to ensure the impact of this phenomena is limited, perfect isolation between these workloads can never be achieved. This is the reason for which a percentage of alternating-directions IGA computations took significantly less or more time in subsequent runs, and some workers finished ahead or behind others. Recognizing this deficiency and in response to user needs, cloud providers generally offer the option to rent machines exclusively. GCP, for instance, features so-called sole-tenant nodes, which guarantee that *Google* will not run software from any other client on the same hardware. This service, however, has many limitations in its current state. Firstly, it is available only in select availability zones (all of which are in the United States) and at a considerably higher price compared to regular nodes. Secondly, the smallest (and only) available node to date is *n1-node-96-624* with 96 threads and 624 GB of memory. Lastly, it is not possible to claim these nodes without longer term commitment. A month of use of this specific hardware costs approximately $3500, which makes sole-tenant nodes a reasonable choice, particularly for smaller companies, which do not have to commit themselves to costly maintenance of their own data centers. That being said, many researchers who intend to perform ad-hoc simulations may find this offering unfit for their needs, which is also the case for this study. The requirement for long-term commitment may in fact contradict the elasticity of provisioning cloud hardware.

CRediT authorship contribution statement

Grzegorz Gurgul: Conceptualization, Formal analysis, Investigation, Software, Validation, Visualization, Writing – original draft. **Bartosz Baliś:** Conceptualization, Supervision, Writing – review & editing. **Maciej Paszyński:** Funding acquisition, Conceptualization, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

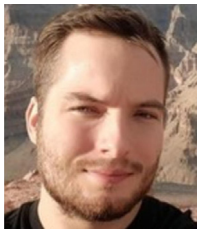
Data will be made available on request.

Acknowledgments

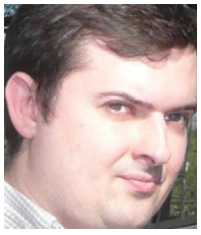
This work of have been supported by Polish National Science Centre grant no. 2017/26/M/ST1/00281. The European Union's Horizon 2020 Research and Innovation Program of the Marie Skłodowska-Curie grant agreement No. 777778 provided additional support.

References

[1] Y. Wang, R. Goldstone, W. Yu, T. Wang, Characterization and Optimization of Memory-Resident MapReduce on HPC Systems, in: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IEEE, 2014, pp. 799–808.
[2] S. Heidari, R.N. Calheiros, R. Buyya, iGiraph: A Cost-Efficient Framework for Processing Large-Scale Graphs on Public Clouds, in: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), IEEE, 2016, pp. 301–310.
[3] A. Mazrekaj, I. Shabani, B. Sejdiu, Pricing Schemes in Cloud Computing: An Overview, Int. J. Adv. Comput. Sci. Appl. 7 (2) (2016) 80–86.
[4] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M.J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A {fault-tolerant} abstraction for {in-memory} cluster computing, in: 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), 2012, pp. 15–28.
[6] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop Distributed File System, in: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST, IEEE, 2010, pp. 1–10.
[7] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: Cluster Computing with Working Sets, HotCloud 10 (10–10) (2010) 95.
[8] M.A.S. Netto, R.N. Calheiros, E.R. Rodrigues, R.L.F. Cunha, R. Buyya, HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges, ACM Comput. Surv. 51 (1) (2018) 1–29.
[9] F.A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, et al., Rethinking Data-Intensive Science Using Scalable Analytics Systems, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015, pp. 631–646.
[10] J.L. Reyes-Ortiz, L. Oneto, D. Anguita, Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf, Procedia Comput. Sci. 53 (2015) 121–130.
[11] B. Balis, K. Figiela, K. Jopek, M. Malawski, M. Pawlik, Porting HPC applications to the cloud: A multi-frontal solver case study, J. Comput. Sci. 18 (2017) 106–116.
[12] J.A. Cottrell, T.J.R. Hughes, Y. Bazilevs, Isogeometric Analysis: Toward Integration of CAD and FEA, John Wiley & Sons, 2009.
[13] L. Dalcin, N. Collier, P. Vignal, A. M.A. Côrtes, V.M. Calo, PetIGA: A framework for high-performance isogeometric analysis, Comput. Methods Appl. Mech. Engrg. 308 (2016) 151–181.
[14] M. Łoś, M. Paszyński, A. Kłusek, W. Dzwinel, Application of fast isogeometric L2 projection solver for tumor growth simulations, Comput. Methods Appl. Mech. Engrg. 316 (2017) 1257–1269.
[15] L. Dedè, M. Borden, J, T.J.R. Hughes, Isogeometric Analysis for Topology Optimization with a Phase Field Model, Arch. Comput. Methods Eng. 19 (2012) 427–465.
[16] H. Gomez, V.M. Calo, Y. Bazilevs, T.J.R. Hughes, Isogeometric analysis of phase-field models: Application to the Cahn-Hilliard Equation, Comput. Methods Appl. Mech. Engrg. 197 (2008) 4333–4352.
[17] H. Gomez, T.J.R. Hughes, X. Nogueira, V.M. Calo, Isogeometric analysis of the isothermal Navier–Stokes–Korteweg equations, Comput. Methods Appl. Mech. Engrg. 199 (2010) 1828–1840.
[18] V.M. Calo, N. F. Brasher, Y. Bazilevs, T.J.R. Hughes, Multiphysics model for blood flow and drug transport with application to patient-specific coronary artery flow, Comput. Mech. 43 (1) (2008) 161–177.
[19] M. Łoś, M. Woźniak, M. Paszyński, L. Dalcin, V.M. Calo, Dynamics with Matrices Possessing Kronecker Product Structure, Procedia Comput. Sci. 51 (2015) 286–295.
[20] M. Woźniak, M. Łoś, M. Paszyński, L. Dalcin, V.M. Calo, Parallel Fast Isogeometric Solvers for Explicit Dynamics, Comput. Inform. 36 (2) (2017) 423–448.
[21] I.S. Duff, J.K. Reid, The Multifrontal Solution of Indefinite Sparse Symmetric Linear Equations, ACM Trans. Math. Software 9 (3) (1983) 302–325.
[22] Y. Saad, Iterative Methods for Sparse Linear Systems, SIAM, 2007.
[23] L. Gao, V.M. Calo, Fast isogeometric solvers for explicit dynamics, Comput. Methods Appl. Mech. Engrg. 274 (2014) 19–41.
[24] L. Gao, V.M. Calo, Preconditioners based on the Alternating-Direction-Implicit algorithm for the 2D steady-state diffusion equation with orthotropic heterogeneous coefficients, J. Comput. Appl. Math. 273 (2015) 274–295.
[25] M. Łoś, M. Woźniak, M. Paszyński, A. Lenharth, M.A. Hassaan, K. Pingali, IGA-ADS: Isogeometric analysis FEM using ADS solver, Comput. Phys. Comm. 217 (2017) 99–116.
[26] H. Gomez, T.J.R. Hughes, Provably unconditionally stable, second-order time-accurate, mixed variational methods for phase-field models, J. Comput. Phys. 230 (13) (2011) 5310–5327.
[27] C. Vecchiola, S. Pandey, R. Buyya, High-Performance Cloud Computing: A View of Scientific Applications, in: 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks, IEEE, 2009, pp. 4–16.
[28] I.A.T. Hashem, I. Yaqoob, N.B. Anuar, S. Mokhtar, A. Gani, S.U. Khan, The rise of "big data" on cloud computing: Review and open research issues, Inf. Syst. 47 (2015) 98–115.
[29] S. Ostermann, R. Prodan, Impact of Variable Priced Cloud Resources on Scientific Workflow Scheduling, in: European Conference on Parallel Processing, Springer, 2012, pp. 350–362.
[30] G. Gurgul, M. Paszyński, Object-oriented implementation of the alternating directions implicit solver for isogeometric analysis, Adv. Eng. Softw. 128 (2019) 187–220.
[31] G. Gurgul, M. Łoś, M. Paszyński, V.M. Calo, Linear computational cost implicit solver for parabolic problems, Computer Science 70 (3) (2020).
[32] B. Irons, A frontal solution program for finite-element analysis, Int. J. Numer. Methods Eng 1970 (2) (1970) 5–32.
[33] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994.
[34] M. Paszynski, Fast Solvers for Mesh-Based Computations, CRC Press, 2015.
[35] R.R. McCune, T. Weninger, G. Madey, Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing, ACM Comput. Surv. 48 (2) (2015) 25.
[36] S. Verma, L.M. Leslie, Y. Shin, I. Gupta, An Experimental Comparison of Partitioning Strategies in Distributed Graph Processing, Proc. of the VLDB Endow 10 (5) (2017) 493–504.
[37] S. Heidari, Y. Simmhan, R.N. Calheiros, R. Buyya, Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges, ACM Comput. Surv. 51 (3) (2018) 60.
[38] A. Gittens, A. Devarakonda, E. Racah, M. Ringenburg, L. Gerhardt, J. Kottalam, J. Liu, K. Maschhoff, S. Canon, J. Chhugani, et al., Matrix factorizations at scale: a comparison of scientific data analytics in sparkand C+MPI using three case studies, in: 2016 IEEE International Conference on Big Data, IEEE, 2016, pp. 204–213.
[39] A. Gittens, K. Rothauge, S. Wang, M.W. Mahoney, J. Kottalam, L. Gerhardt, M. Ringenburg, K. Maschhoff, Alchemist: An Apache Spark - MPI interface, Concurr. Comput.: Pract. Exper. 31 (16) (2019).
[40] R.C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Pearson Education, 2009.
[41] V. Puzyrev, M. Łoś, G. Gurgul, V.M. Calo, W. Dzwinel, M. Paszyński, Parallel splitting solvers for the isogeometric analysis of the Cahn-Hilliard equation, Comput. Methods Biomech. Biomed. Eng. 22 (16) (2019) 1269–1281.
[42] B.P. Rimal, E. Choi, I. Lumb, A Taxonomy and Survey of Cloud Computing Systems, in: 2009 Fifth International Joint Conference on INC, IMS and IDC, IEEE, 2009, pp. 44–51.
[43] P. Lama, S. Wang, X. Zhou, D. Cheng, Performance Isolation of Data-Intensive Scale-out Applications in a Multi-tenant Cloud, in: 2018 IEEE International Parallel and Distributed Processing Symposium, IEEE, 2018, pp. 85–94.
[44] S. Mŏding, Analyze System Scalability in R with the Universal Scalability Law, 2020.
[45] V. Calo, P. Minev, V. Puzyrev, Splitting schemes for phase-field models, Appl. Numer. Math. 156 (2020) 192–209.
[46] A. Szyszka, M. Woźniak, R. Schaefer, Concurrent algorithm for integrating three-dimensional B-spline functions into machines with shared memory such as GPU, Comput. Methods Appl. Mech. Engrg. 398 (115201) (2022) 1–30.
[47] M. Woźniak, A. Szyszka, S. Rojas, A study of efficient concurrent integration methods of B-spline basis functions in IGA-FEM, J. Comput. Sci. (64) (2022) 1–42.
[48] Q. Xu, H. Jeon, M. Annavaram, Graph processing on GPUs: Where are the bottlenecks? in: 2014 IEEE International Symposium on Workload Characterization (IISWC), IEEE, 2014, pp. 140–149.



Grzegorz Gurgul obtained his Ph.D. in 2021 at Institute of Computer Science at the AGH-UST University of Science and Technology. His research interest includes cloud computing, isogeometric analysis, and software engineering.



Bartosz Baliś is an associate professor at the Institute of Computer Science of the AGH University of Science and Technology. He obtained his Ph.D. in Computer Science in 2009 and his Habilitation in Computer Science in 2019 from the AGH University. He is a co-author of over 60 international peer-reviewed scientific publications, including papers in high-ranked journals. His research interests include scientific workflows, e-Science, cloud computing, and distributed computing. Dr. Balis has been a member of conference program and organizing committees, including Euro-Par 2020 workshops General Co-Chair, HPCS 2018–19 Tutorials Co-Chair, IEEE/ACM SC18 Birds of a Feather Planning Committee, IEEE/ACM SC16 Workshops Planning Committee; and a PC member for International Conference on Computational Science (ICCS) (2007–2021), Lambda Days (Research Track) (2018), ITU KALEIDOSCOPE (2011, 2013–2017), International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH) (2016–2018), IEEE e-Science Conference (2006). He has participated in national and international EU-funded research projects, including CrossGrid, CoreGRID, K-Wf Grid, ViroLab, Gredia, UrbanFlood, PaaSage and WATERLINE. He is also a senior researcher in the CERN ALICE experiment.

Maciej Paszyński is a full professor of Computer Science at Institute of Computer Science, AGH University, Krakow, Poland. He obtained his Ph.D. in Mathematics with Applications to Computer Science from Jagiellonian University in 2003, and his Habilitation in Computer Science in 2010 from AGH University. He co-authored over 160 papers in impact factor journals. He presented over 100 presentation at conferences and workshops. He is an editor of Journal of Computational Science and thematic track chair of the International Conference on Computational Science. He collaborates with research groups from The University of Texas at Austin, Basque Center for Applied Mathematics in Bilbao, Spain, Curtin University at Perth, Western Australia, and Catholic University of Vaplaraiso, Chile. His research interests include artificial intelligence and HPC for advanced simulations.