Report 1, 2017 [*]

Kirill Voronin

Abstract

In the report results obtained for the implemented parallel mesh generator and 4d elements are presented.

Introduction

Finite element method is nowadays a common approach for solving various application problems in physics which is known first of all for its easy-to-implement methodology. Along with the other closely related approaches, e.g., finite volume methods or discontinuous Galerkin methods finite element method has proved its efficiency by handling complicated problems in numerous applications. Application problems usually require detailed meshes with large number of nodes in order to resolve all geometrical or physical features of the considered domain.

Traditionally, these methods are used for discretization in space whereas different approaches, e.g., finite differences or Runge-Kutta methods, are used for time discretization. However, in many cases separation of time and space can introduce additional difficulties to the problem, e.g., for problems in time-dependent domains or when conditionally stable time discretization methods are used.

One of the alternative approaches which has gained much attention and developed greatly recently are space-time methods which provide a simultaneous approximation in both time and space. Apparently, one of the main disadvantages of space-time methods is the increased size of the problem to be solved. However, when combined with appropriate upscaling techniques, these methods can be quite efficient and at the same time avoid some of the drawbacks which are typical for traditional discretization methods.

Certainly, for space-time methods appropriate space-time meshes are to be constructed. While a number of mesh generating software exist for two- and three-dimensional cases, almost none are known to the authors which consider four-dimensional case (three-dimensional space + time). Since space-time approaches introduce an additional dimension to the mesh,

[*]This work was supported by ..., grant No.

it is also important to be able to construct the space-time mesh in parallel. First, this will reduce memory requirements since the space-time mesh as a whole might not fit into the memory of a single computational node. Second, it will reduce the total time for mesh generation.

This paper is focusing on the mesh generating algorithm which constructs tensor-product extension $(d+1)$-dimensional space-time meshes for given d-dimensional (triangular or tetrahedral) space meshes in parallel. In section 1 space-time CFOSLS approach is outlined together with basic discretization details. The essence of the paper - mesh generating algorithm is presented in section 2. Apart from mesh generating it is visualization which might be important for space-time approaches. This question is addressed to in section 3. Finally, several numerical experiments in which we study the space-time approximation properties as well as the meshing algorithm performance are presented in section 4.

1 Space-time CFOSLS

Consider the following problem in a space-time cylinder $\Omega_T = \Omega \times [0,T]$ where Ω is a regular(?) domain:

$$\frac{\partial S}{\partial t} + \mathrm{div}_x\left(\mathcal{L}(S)\right) = q(\mathbf{x},t), \quad (\mathbf{x},t) \in \Omega_T$$

which can be either hyperbolic ($\mathcal{L} = \mathbf{b}S$ with velocity $\mathbf{b}(\mathbf{x},t)$) or parabolic ($\mathcal{L} = -k\nabla S$) depending on the choice of $\mathcal{L}$.

For this initial boundary value problem we also have the initial condition

$$S(\mathbf{x},0) = S_0(\mathbf{x})$$

and appropriate boundary conditions. Specifically, for the hyperbolic problem which is considered as an example in the paper, a correct choice can be

$$\mathbf{b}\cdot\mathbf{n} = 0, \ (\mathbf{x},t) \in \partial\Omega \times [0,T],$$

i.e., no charasteristics are going inside the domain through the lateral boundary of the space-time cylinder, together with $\mathrm{div}_x\,\mathbf{b} \geq 0$ in order for the problem to be correctly posed.

Then one can introduce

$$\boldsymbol{\sigma} = \left[\begin{array}{c} \mathcal{L}(S) \\ S \end{array}\right]$$

and proceed with the equivalent first order formulation:

$$\left\{\begin{array}{c} \mathrm{div}_{x,t}\,\boldsymbol{\sigma} = q \\ \boldsymbol{\sigma} - \left[\begin{array}{c} \mathcal{L}(S) \\ S \end{array}\right] = 0 \end{array}\right.$$

Now, the idea of CFOSLS [?] is to consider the above problem above as a minimization problem for the following quadratic functional:

$$J(\boldsymbol{\sigma}, S) = \|\boldsymbol{\sigma} - \begin{bmatrix} \mathcal{L}(S) \\ S \end{bmatrix} \|^2_{0,K^{-1}} + \| \operatorname{div}_{x,t} \boldsymbol{\sigma} - q\|^2_0$$

in $H(\operatorname{div}) \times V$ under the constraint

$$(\operatorname{div}_{x,t} \boldsymbol{\sigma}, w) = (q, w) \text{ for all } w \in L_2(\Omega_T).$$

Here $K = K(\mathbf{x})$ is a symmetric and positive definite matrix coefficient, $\| \cdot \|_0 (\| \cdot \|_{0,K^{-1}})$ denotes the (weighted) L_2 norm w.r.t to the space-time domain Ω_T and V is an appropriate functional space for S: L_2 in hyperbolic case and H^1 in parabolic case. It is assumed implicitly that initial and boundary conditions are incorporated into the definition of the considered functional spaces.

In hyperbolic case, after writing down the minimum conditions for $J(\boldsymbol{\sigma}, S)$ the scalar unknown S can be eliminated by using the following (assuming for simplicity that K is identity):

$$\begin{bmatrix} \mathbf{b} \\ 1 \end{bmatrix}^T \begin{bmatrix} \mathbf{b} \\ 1 \end{bmatrix} S = \begin{bmatrix} \mathbf{b} \\ 1 \end{bmatrix}^T \boldsymbol{\sigma}$$

Then, the variational form of the problem reads as:

$$\begin{array}{lll} ((I - \frac{1}{\mathbf{b}^T\mathbf{b}}\mathbf{b}\mathbf{b}^T)\boldsymbol{\sigma},\ \boldsymbol{\theta}) + (\operatorname{div}_{x,t} \boldsymbol{\sigma},\ \operatorname{div}_{x,t} \boldsymbol{\theta}) + (\lambda,\ \operatorname{div}_{x,t} \boldsymbol{\theta}) & = (f,\ \operatorname{div}_{x,t} \boldsymbol{\theta}), & \text{for all } \boldsymbol{\theta} \in H(\operatorname{div}), \\ (\operatorname{div}_{x,t} \boldsymbol{\sigma},\ \mu) & = (f, \mu), & \text{for all } \mu \in L_2. \end{array}$$

One can notice that the $\operatorname{div} - \operatorname{div}$ term (the second one in the first equation) can be actually eliminated by changing λ (the righthand side of the first equation becomes zero then). This is equivalent to excluding the second term in the definition of the CFOSLS functional $J(\boldsymbol{\sigma}, S)$.

To approximate the obtained problem in $H(\operatorname{div})$ - L_2 one can use $(d + 1)$-dimensional Raviart-Thomas finite elements [?]. On a $(d + 1)$-dimensional element T Raviart-Thomas finite elements of order k are defined as

$$\mathbf{RT}_k(T) = P_k(T)^{d+1} + \mathbf{x}_{xt} \cdot \tilde{P}_k(T)$$

where $P_k(T)$ and $\tilde{P}_k(T)$ are the set of polynomials of degree $\leq k$ and set of homogeneous polynomials pf degree k on T and $\mathbf{x}_{xt} = (\mathbf{x}, t)^T$.

Then, the global approximation space for $H(\operatorname{div})$ is defined as

$$\mathbf{RT}_k(\Omega_T, \mathcal{T}_{h\tau}) = \left\{ \mathbf{v} \in L_2(\Omega_T)^{d+1} | \ \mathbf{v}|_T \in \mathbf{RT}_k(T) \text{ for all } T \in \mathcal{T}_{h\tau} \right\}$$

whereas a compatible approximation of L_2 (so that the pair of spaces satisfied *inf-sup* condition) is then given simply as discontinuous polynomials

$$Q_k(\Omega_T, \mathcal{T}_{h\tau}) = \left\{ q \in L_2(\Omega_T)^{d+1} | \ q|_T \in P_k(T) \text{ for all } T \in \mathcal{T}_{h\tau} \right\}.$$

for a given space-time triangulation $\mathcal{T}_{h\tau}$ of the domain Ω_T. Construction of the space-time triangulations is discussed in the next section.

After approximating the $H(\text{div}) - L_2$ formulation using Raviart-Thomas finite elements one can finally arrive at the linear system which takes the following form:

$$\begin{bmatrix} M + BB^T & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{\sigma}_h \\ \mu_h \end{bmatrix} = \begin{bmatrix} B^T f \\ f \end{bmatrix}.$$

2 Mesh construction

The developed $(d+1)$-dimensional $(d = 2, 3)$ parallel mesh generator is dedicated for constructing meshes of tetrahedrons (or pentatops) for space-time cylinders given arbitrary triangular(or tetrahedral) meshes (called *base meshes* below) in the cylinder base. The input parameters for the algorithm are: base mesh, number of time slabs and time step. Constructed space-time meshes are essentially tensor-product extensions of the base mesh into the higher-dimensional space.

2.1 Serial algorithm

Here main steps of serial algorithm are discussed. Although mesh storage formats used by mesh generating software can slightly vary, our implementation is based on MFEM [?] framework and therefore we use MFEM mesh format for unstructured meshes as a model. It includes the following key data structures:

1. elements (given by vertex indices)
2. boundary elements (given by vertex indices)
3. vertices (given by their coordinates)

Having these data available for the base mesh, one needs to create similar data structures for the space-time mesh.

2.1.1 Generating vertices

In general, one might consider introducing additional nodes inside the time slabs [?] in order to produce a mesh which approximates better certain features of the space-time domain. However, in the implemented algorithm we take the viewpoint that such mesh improvements should be incorporated into the refinement strategies which are outside of the scope of the mesh generation. Thereby, since no additional vertices are added inbetween the time sections, generation of the space-time vertices is straight-forward. The space-time

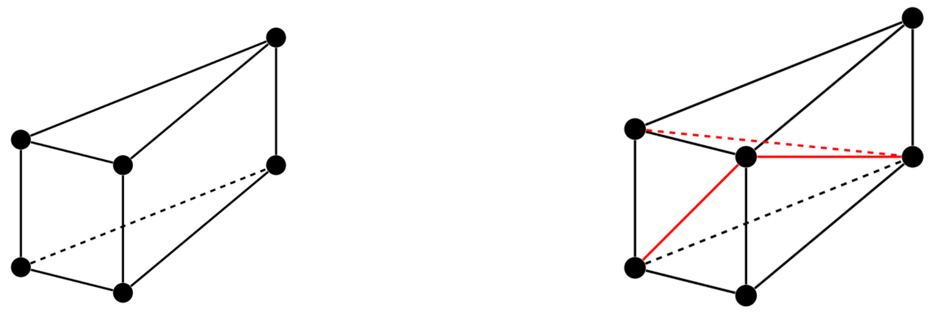

Figure 1 – A space-time prism in three-dimensional case (left) and its tesselation into tetrahedrons (right).

vertices are nothing else but $(d + 1)$-dimensional duplicates of base mesh vertices with varying time coordinate.

2.1.2 Generating elements

The implemented algorithm for generating $(d + 1)$-dimensional mesh elements follows the guidelines from [?] and [?]. Elements are generated in an element-by-element fashion by the following local procedure applied to each of the base mesh elements:
Local procedure to generate elements:

In a loop over all time slabs:

1. For a given time slab form a $(d+1)$-dimensional prism with a given base mesh element in the base,
2. Decompose the prism into d elements (tetrahedrons or pentatops), see Fig.1.

Thus, each prism will be decomposed into $(d + 1)$-simplices which are the new space-time elements.

For the last step two options were considered and implemented.

First approach. The more general approach [?] is to construct a Delaunay triangulation. For creating a Delaunay triangulation one of the standard geometrical packages, e.g., Qhull [?] can be used. However, one should remember that for a prism with regular faces (in our case the lateral faces are regular) Delaunay tesselation is not unique. This might lead to nonconforming meshes due to the incompatible tesselation of the common faces for neighbouring prisms. In order to circumvent this, one can joggle (perturb) time coordinates for the vertices in the prism upper base (see Fig.2). In *serial* algorithm the joggling can be random or based on any other information which is global for neighbouring elements, e.g. global vertex indices. For parallel implementation using information which is global for neighbouring elements is not enough since then this idea may lead to additional data transfers and synchronization points. Another thing to keep in mind is that due to the round-off errors one might get sliver simplices which should be eliminated afterwards.

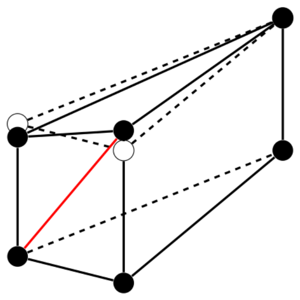

Figure 2 – A space-time prism with perturbed vertices in three-dimensional case. Now the Delaunay tesselation is unique.

Finally, the Delaunay based approach can be written as the following local procedure:
Decomposing $(d+1)$-dimensional prism: Delaunay triangulation

1. Perturb vertex coordinates so that Delaunay triangulation of the prism is unique;
2. Create a Delaunay triangulation of the prism (using qhull [?]);
3. Eliminate sliver (almost degenerate) $(d+1)$-simplices.

The elimination can be done by checking the relative simplex volume. The regular volume should be of order $h^3\tau$ where h is the base mesh step and τ is the time step.

Second approach. Another approach is to use a standard tesselation [?] of a $(d+1)$-dimensional prism into $(d+1)$-dimensional simplices.
Decomposing $(d+1)$-dimensional prism: Standard simplex tesselation

1. If prism vertices in the lower and upper bases are denoted as $A_1, ..., A_k$ and $A'_1, ..., A'_k$ correspondingly (see Fig.3), then one can take the following simplices for the prism decomposition: $\{A_1, ..., A_k, A'_1\}$, $\{A_2, ..., A_k, A'_1, A'_2\}$, ..., $\{A_k, A'_1, ..., A'_k\}$.

In order to have compatible tesselations across neighbouring elements, the authors of [?] suggest to reorder the prism vertices so that the global ordering of vertex indices is preserved. This idea will be slightly modified for the parallel setup.

2.1.3 Generating boundary elements

It is easy to notice that the boundary of the space-time mesh consists of the following parts:

1. lower cylinder base,

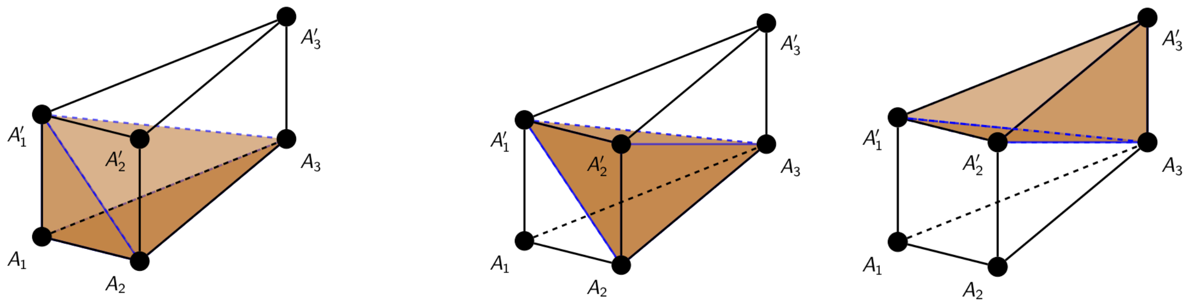

Figure 3 – Decomposition of a three-dimensional space-time prism into three tetrahedrons based on the vertex numbering.

2. upper cylinder base,
3. element faces whose projections onto the base mesh along the time direction are boundary elements of the base mesh.

The first and second types of boundary elements can be added to the structure for space-time boundary elements at any moment. The key question is then how to handle the third type, i.e., how to determine whether any faces of a given base mesh element belong to the boundary of the base mesh. Again, here several options are possible.

In case when no face-to-boundary connection table (or any equivalent) is available for the base mesh, one can create a list of all base mesh boundary elements and look up in it for each base face which leads to $O(n_f \cdot \log n_{be})$ operations where n_f is the number of faces and n_{be} is the number of boundary elements in the base mesh.

Usually, however, face-to-boundary connection table is known so one can simply use it for checking if the given base element's face belongs to the boundary.

2.2 Parallel algorithm

As an input for the parallel algorithm we consider a base mesh which is already distributed among the processes. In this case the distributed mesh parts (for our model data structures in MFEM) have their own local-to-process numbering as well as data structures for shared entities: faces, planars (in 4D), edges, and vertices.

The parallel implementation consists of two main steps which are:

1. generating local parts of the space-time mesh on the processes,
2. setting up communication structures governing the shared entities between processes.

Certainly, local parts are generated via the serial algorithm on each process.

As for the shared entities, one should notice that as soon as shared entities are known for the base mesh, the shared entities can be also defined for the space-time elements by considering projections of space-time entities onto the base. For example, shared planars in 4D (which are essentially triangles) can be defined as those planars whose projection on the base is either a shared edge or a shared face (triangle) of the tetrahedral base mesh.

It is quite easy to compute the total number of shared planars for each process. Consider a lateral face of the 4D space-time prism (which itself is a three-dimensional prism in 4D) decomposed into three tetrahedrons, see Fig.4. There are 2 bases and 2 inner triangles which are projected onto the same base triangle. In addition, for each lateral face of the considered 3D prism (which is a rectangle in 4D), there are 2 triangles which are projected onto the same base edge.

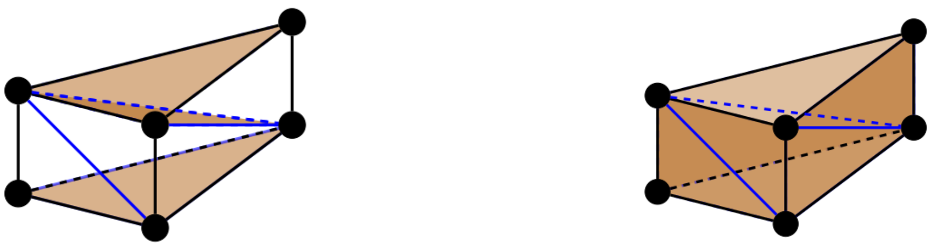

Figure 4 – Two types of shared planars in the lateral face which can be considered as a three-dimensional prism. Projections of the planars onto the base plane are eigher base mesh elements (left) or base mesh edges (right).

Similarly, knowing shared entities for the base mesh one can define the number of all shared space-time entities via projections:

$$n_{shared\,faces,d+1} = d \cdot n_{steps} \cdot n_{shared\,faces,d}$$

$$n_{shared\,edges,d+1} = (2 \cdot n_{steps} + 1) \cdot n_{shared\,edges,d} + n_{steps} \cdot n_{shared\,vertices,d}$$

$$n_{shared\,vertices,d+1} = (n_{steps} + 1) \cdot n_{shared\,vertices,d}$$

$$n_{shared\,planars,4} = (2 \cdot n_{steps} + (n_{steps} + 1)) \cdot n_{shared\,faces,3} + 2 \cdot n_{steps} \cdot n_{shared\,edges,3}$$

Here n_{steps} is the number of time slabs in the space-time cylinder, subscripts d and $d+1$ correspond to the space-time and base meshes.

For example, generation of shared planars can be done by performing the following:
Creating shared planars for the space-time mesh:

In a loop over space-time planars

1. Project the given planar onto the base.
2. Look in the set of shared edges and shared faces of the base mesh for the projection. If found, add a shared planar.

The look-up in the set of base mesh shared entities (step 3 above) implies $O(n_{entity,d+1} \cdot \log n_{sharedentities,d})$ operations. The reason for loops over space-time entities is that if standard routines are used for generating faces, edges and planars, one does not know in advance how they are ordered. As a possible improvement, one could reimplement these generating routines so as to know which space-time entities will be shared.

Special care should be taken so that for each group of the processes which share any mesh entities, the order in which shared entities appear in the lists is the same for all processes in the group (requirement by MFEM). Another important requirement is to ensure compatible orientation of shared entities across the neighbouring processes and at the same time preserve the local-to-entity indices ordering which can be required by refinement routines.

In comparison to serial case, parallel setup introduces one additional complication related to the mesh compatibility across neighbouring processes. Usually, mesh parts are handled by different processes and have their own local-to-process vertex numbering. Therefore, one needs to ensure compatibility of the mesh across neighbouring mesh parts which is no longer preserved if one uses local-to-process vertex numbering.

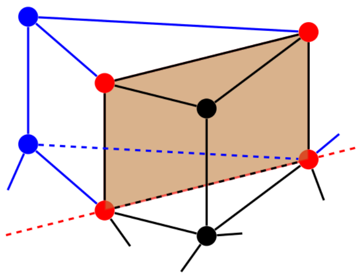

Figure 5 – A shared face between two neighboring space-time prisms in 3D is essentially a rectangle (with red vertices). The same diagonal should be chosen in the rectangle to ensure mesh compatibility.

Here one can notice that in general it is not required to have exactly the same vertex perturbation for common vertices across neighbouring elements and processes in Delaunay triangulation. The same is true for the vertex ordering if one uses simple prism decomposition into simplices. However, in any case one still needs to generate a compatible triangulation.

The simple solution is to use local-to-element "geometric" ordering of vertices. That is, one can just reorder the vertices so that the following "lexicographical" order is preserved: vertex A is said to be "larger" than vertex B if x-coordinate of A is larger of x-coordinate of B, or their x-coordinates are equal and the next coordinate of A is larger than the next one of B, etc. For example, vertex $(0.0; 1.0; 2.0)$ is larger than $(0.0; 1.0; 0.0)$, but less than $(0.0; 2.0; 1.0)$.

Thus, vertex joggling can be defined to be proportional to the vertex index in this ordering or this ordering can be directly used in simplex decomposition described above. The same trick can be used whenever one needs to orient a space-time entity. Namely, the orientation can be said to be "positive" if its vertex ordering is an even permutation of the "geometric" ordering defined above.

Remark. The presented algorithm is completely parallel since the local procedures depend only on the local-to-process data. The only communication and synchronization point is required in the very end while finalizing the communication structures which are, for example, communicators for groups of processes which share some entities.

3 Visualization

For space-time meshes in 4D it is also important to visualize the obtained results. A common approach is to slice the space-time domain by a set of hyperplanes defined by $t = t_k$ for a series of time moments t_k. The intersection of the $(d+1)$-dimensional space-time mesh and a hyperplane is a d-dimensional mesh (called *slice mesh* below). For tetrahedral and pentatop meshes the slice mesh consists of two types of cells: triangles/quadrilaterals in 3D and tetrahedrons/wedges in 4D (see Fig.6).

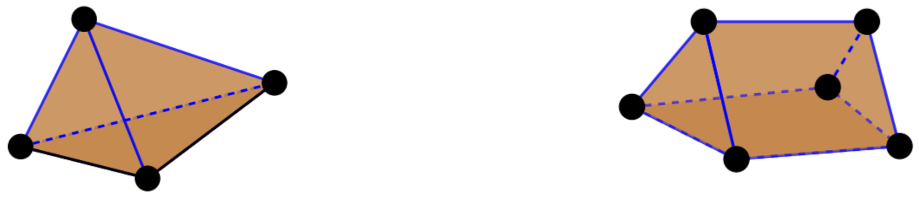

Figure 6 – Two types of slice mesh cells in 4D: tetrahedrons (left) and wedges(right).

In [?] an approach based on cutting edges of the mesh elements was presented. The idea is the following:
Local procedure to compute the slice mesh cells:

1. Check whether a considered space-time element is intersected by a given hyperplane.
2. If it is intersected, two cell types are possible: a tetrahedron or a wedge in 4D, or a triangle or a quadrilateral in 3D.
3. The vertices of the slice mesh cell can be computed by intersecting each edge of the space-time element with the hyperplane, see Fig. 7. This can be done by solving a small local system of linear algebraic equations [?].

The procedure can be implemented in parallel while the final output can be processed, e.g., by ParaView [?] using unstructured VTK [?] or any other standard format. An example of a slice mesh distributed over two parallel processes is presented in Figures 1 and 2. The

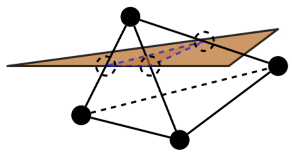

Figure 7 – An example of slicing a space-time element for a three-dimensional case. The slice cell in 3D is in general either a triangle or a quadrilateral.

slice mesh consists of tetrahedrons and wedges. The space-time domain for this mesh was a cylinder with three-dimensional ball in the base:

$$\Omega_T = \left\{(x, y, z, t) | \, x^2 + y^2 + z^2 \leq 1, t \in [0, 1]\right\}.$$

In Fig.1 local-to-process mesh parts are shown for the slice with $t = 0.1$ plane and in Fig. 2 the mesh slices for $t = 0.1$ and $t = 0.4$ are shown.

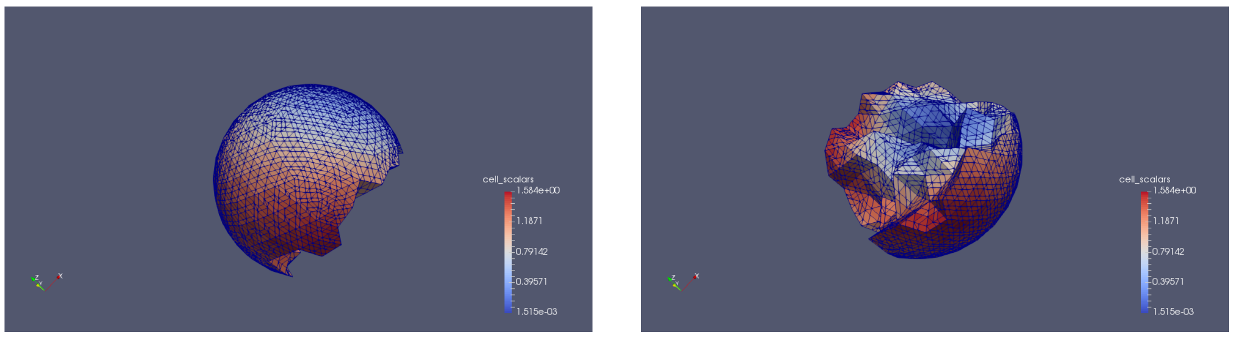

Figure 8 – Slice mesh for $t = 0.1$: process 0 (left) and process 1 (right).

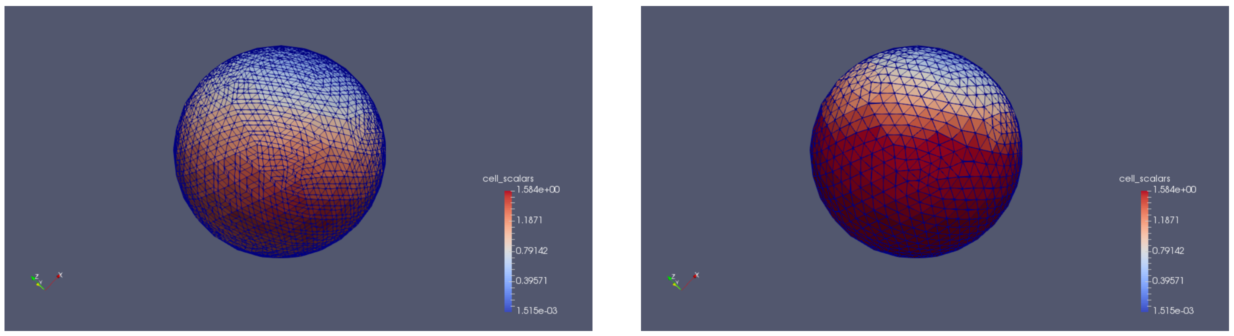

Figure 9 – Slice mesh for $t = 0.1$ (left) and $t = 0.5$ (right).

4 Numerical experiments

Numerical experiments were perfomed using Coeus cluster [?] at Portland State University (PSU). The computational nodes used for numerical experiments have the following specification: Intel Xeon ???, GHz, RAM, Infiniband ...

4.1 Mesh generator

In general, time for solving the linear system is usually higher than time for mesh generation. Hence the goal of the performance testing for mesh generator is to show that the implemented mesh generating algotihm scales well and hence will not be a performance bottleneck when the number of processes will increase. As already mentioned, the presented algorithm requires very few communication between parallel processes and its nature is essentially parallel since it mainly consists of local procedures. Hence, one should expect it to show a very good scaling with respect to the number of processes. Time and memory scalability of the mesh generator are shown by several simple examples below.

In table 1 time scalability for space-time mesh generation is presented for a mesh constructed using n_{proc} number of MPI processes from a base mesh with 114,688 elements and 40 time slabs which result in 18,4 mln of elements for the space-time mesh. In the second column configuration of the parallel system is given as $n \times m$, where n is the number of computational nodes and m is the number of cores per node. In the third column the mesh generation time is presented without considering the construction of the base mesh or doing any additional refinements afterwards. The last column shows the ratio of times between the considered configuration and the previous one (with the ideal ratio being proportional to the change in the number of computational cores which is two in the table).

Table 1 – Parallel mesh generator: Timing

n_{proc}	configuration	time (s)	ratio
1	1×1	301	-
2	2×1	144	2.1
4	4×1	71	2.02
8	8×1	36	1.97
16	16×1	18	2.0
32	16×2	9	2.0
64	16×4	4.9	1.83
128	16×8	2.8	1.75

As expected, the scaling is almost perfect when a few number of processes are used and deteriorates a little bit due to the unavoidable overhead when the local-to-process mesh parts become quite small.

The second table shows memory consumption of the mesh generator for different number of MPI processes. Here the final mesh has 24.5 mln of space-time elements and 1.2 mln of space-time nodes. The first column is again the number of processes, the second is the configuration of the system. The third column shows the peak memory required on the root node. Now the last column shows the ratio between memory consumption for the two consecutive configurations.

Table 2 – Parallel mesh generator: Memory consumption

n_{proc}	configuration	memory(Gb)	ratio
1	1×1	21.4	-
2	2×1	11.0	1.95
4	4×1	5.5	2.0
8	8×1	2.8	1.96
16	16×1	1.4	2.0

From table 2 we can conclude that the memory scales almost linearly.

Thus, the presented parallel algorithm is able to handle large space-time meshes which can resolve the geometry of the considered problem with desired accuracy.

4.2 Approximation on space-time meshes

In this section several hyperbolic problems in 3D and 4D are considered on the space-time meshes constructed by the implemented algorithm.

The linear system arising from the discretization was solved by MINRES with a block diagonal preconditioner: diagonal matrix for (1,1) and BoomerAMG preconditioner (from HYPRE [**?**]) for the Schur complement $B^T M^{-1} B$ for the block (2,2). One should notice here that for hyperbolic case matrix M is symmetric and only positive semi-definite (but positive definite on the kernel of B due to the restrictions on velocity b) with a large kernel. As it turned out, the aforementioned preconditioner is not optimal (unlike in parabolic case).

The exact solution for the first test example in 3D was:

$$S(x,y,t) = e^t \cdot \sin\left((x-0.5)^2 + y^2\right)$$

in the cylinder $\Omega_T = \{(x,y,t)|\, x^2+y^2 \leq 1, t \in [0,1]\}$. The velocity was given by the formula:

$$\mathbf{b}(x,y,t) = \begin{pmatrix} -y \\ x \end{pmatrix}$$

In the table below errors for σ and S in L_2 norms, as well as the convergence order are given. Here h is the space-time mesh step.

Table 3 – Hyperbolic problem, 3D, cylinder on the circle

h	$\parallel \sigma \parallel_{L_2}$	order	$\parallel S \parallel_{L_2}$	order	iter
0.26	0.12	-	0.11	-	225
0.12	0.06	1.0	0.05	1.14	407
0.06	0.03	1.0	0.025	1.0	695
0.03	0.014	1.1	0.013	0.94	1140
0.015	0.007	1.0	0.006	1.11	1576

As one can notice, the results demonstrate almost perfect first order for both **sigma** and S.

In 4D case we considered the following example:

$$S(x, y, z, t) = e^t \cdot \sin\left((x - 0.5)^2 + y^2 + (z - 0.25)^2\right)$$

in the cylinder with 3D ball in the base $\Omega_T = \{(x, y, z, t)|\, x^2 + y^2 + z^2 \leq 1, t \in [0, 1]\}$ with velocity:

$$\mathbf{b}(x, y, z, t) = \begin{pmatrix} -y \\ x \\ 0 \end{pmatrix}$$

Again, in the table below errors for σ and S in L_2 norms, as well as the convergence order are presented:

Table 4 – Hyperbolic problem, 4D, cylinder on the ball

h	$\parallel \sigma \parallel_{L_2}$	order	$\parallel \sigma \parallel_E$	order	$\parallel S \parallel_{L_2}$	order	iter
0.26	0.1	-	0.04	-	0.075	-	640
0.13	0.047	1.09	0.02	1.0	0.037	1.02	1282
0.065	0.023	1.03	0.01	1.0	0.018	1.04	2454

Again, along with the non-optimality of the preconditioner (the iteration number increases as $\frac{1}{h}$) one can see the first order of convergence as predicted by the theory.

The numerical solution on a coarse mesh is presented in Fig. 3. Here three slices at time momemnts $t = 0.1$, $t = 0.5$ and $t = 0.9$ are given. The color shows the norm of $\boldsymbol{\sigma}$ - $|\boldsymbol{\sigma}(\mathbf{x})|$ pointwise.

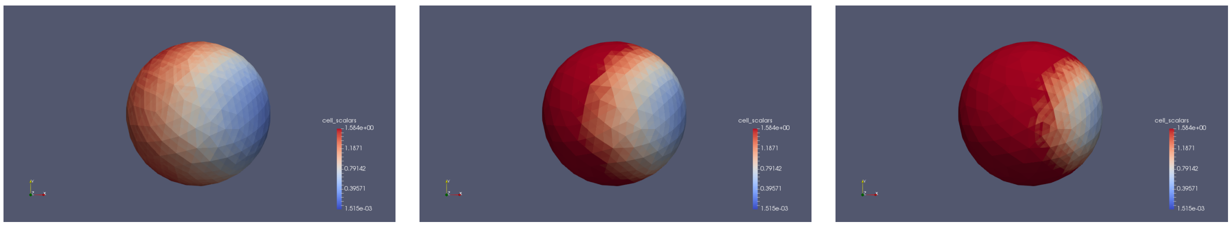

Figure 10 – Numerical solution for hyperoblic problem in 4D. Slices for $t = 0.1$ (left), $t = 0.5$ (middle) and $t = 0.9$ (right).

5 Acknowledgements

6 To-do list:

- Rewrite the introduction
- Add references to introduction
- Check the existing references
- Check the figure references
- Iterate the contents

References

[1] M. Behr. "Simplex Space-Time Meshes in Finite Element Simulations", Int. J. Numer. Meth. Fluids 2008; 57:14211434.

[2] M. Neumueller, O. Steinbach. Refinement of flexible spacetime finite element meshes and discontinuous Galerkin methods, Comput. Visual Sci. (2011) 14:189205, DOI 10.1007/s00791-012-0174-z

[3] E. Karabelas, M. Neumueller. Generating admissible space-time meshes for moving domains in $d+1$-dimensions, https://arxiv.org/abs/1505.03973.

[4] M. Neumueller, P.S. Vassilevski, U. Villa. Space-time CFOSLS Methods with AMGe Upscaling, 23th International Conference on Domain Decomposition Methods, 2016 (not sure?). CFOSLS references

[5] P.A. Raviart, J.M. Thomas. A mixed finite element method for second order elliptic problems, Mathematical Aspects of the Finite Element Method, (I. Galligani, E. Magenes, eds.), Lectures Notes in Math. 606, Springer Verlag, 1977.

[6] HYPRE: A library of High Performance Preconditioners. https://www.llnl.gov/CASC/hypre/.

[7] Modular Finite Element Methods (MFEM), http://mfem.org

[8] Qhull. http://www.qhull.org

[9] U. Ayachit. The ParaView Guide: A Parallel Visualization Application. Kitware Inc., Clifton Park (2015).

[10] W. Schroeder; K. Martin; B. Lorensen.The Visualization Toolkit (4th ed.), Kitware, (2006), ISBN 978-1-930934-19-1.

[11] Coeus cluster