Struct Multidisc Optim (2015) 52:845–859
DOI 10.1007/s00158-015-1252-x

RESEARCH PAPER

PolyTop++: an efficient alternative for serial and parallel topology optimization on CPUs & GPUs

Leonardo S. Duarte[1] · Waldemar Celes[1] · Anderson Pereira[2] · Ivan F. M. Menezes[3] · Glaucio H. Paulino[4,5]

Received: 6 February 2014 / Revised: 2 April 2015 / Accepted: 10 April 2015 / Published online: 5 June 2015

Abstract This paper presents the PolyTop++, an efficient and modular framework for parallel structural topology optimization using polygonal meshes. It consists of a C++ and CUDA (a parallel computing model for GPUs) alternative implementations of the `PolyTop` code by Talischi et al. (Struct Multidiscip Optim 45(3):329–357 2012b). PolyTop++ was designed to support both CPU and GPU parallel solutions. The software takes advantage of the C++ programming language and the CUDA model to design algorithms with efficient memory management, capable of solving large-scale problems, and uses its object-oriented flexibility in order to provide a modular scheme. We describe our implementation of different solvers for the finite element analysis, including both direct and iterative solvers, and an iterative 'matrix-free' solver; these were all implemented in serial and parallel modes, including a GPU version. Finally, we present numerical results for problems with about 40 million degrees of freedom both in 2D and 3D.

✉ Glaucio H. Paulino
paulino@gatech.edu

[1] Pontifical Catholic University of Rio de Janeiro, Computer Science, Rio de Janeiro, Brazil

[2] Pontifical Catholic University of Rio de Janeiro, Tecgraf, Rio de Janeiro, Brazil

[3] Pontifical Catholic University of Rio de Janeiro, Mechanical Engineering, Rio de Janeiro, Brazil

[4] University of Illinois at Urbana-Champaign, Civil and Environmental Engineering, 790 Atlantic Drive, Urbana, GA 30332, USA

[5] School of Civil and Environmental Engineering, Georgia Institute of Technology, 790 Atlantic Drive, Atlanta, GA 30332, USA

Keywords Topology optimization · Polygonal elements · Direct solver · Iterative solver · Parallel solver

1 Introduction

Structural topology optimization methods are utilized to find an ideal material distribution within a given domain that is subjected to loading and support conditions. Most computationally oriented papers in topology optimization, that rely on the finite element method (FEM), employ either triangular or quadrilateral meshes consisting of linear elements with constant design variables. However, numerical instabilities such as checkerboard patterns are well-known in density based methods (Talischi et al. 2010; Rozvany et al. 2003). One can use regularization schemes such as filtering to suppress such numerical instabilities, but these measures often involve heuristic parameters that can augment the optimization problem and lead to significant weight increases (Sigmund 2001; Andreassen et al. 2010). Recently polygonal discretizations have been proposed to achieve stable topology optimization formulations using lower order elements (degrees of freedom sampled at the nodes and constant design variable within the element) as reported in references (Talischi et al. 2010, 2012b, 2014); these elements are the focus of our present work.

Another important issue in the implementation of topology optimization is the ability to extend, develop and modify the code to solve more complicated and large-scale problems. For example, by using a modular code it is easier to replace the current analysis method with a more suitable analysis package for solving different problems. The formalism of this modular approach is crucial when one seeks to improve the analysis routines, including the compliance equation and its sensitivities, without changing the topology optimization formulation, including the material interpolation and regularization schemes (e.g. filters and other manufacturing constraints).

The theory around topology optimization has received considerable attention when compared to practical challenges in developing efficient and modular codes, especially when the goal is to solve complex, large-scale problems with millions of degrees of freedom (dofs). Some educational codes, such as the 99-line (Sigmund 2001) and 88-line (Andreassen et al. 2010), serve as a resource to the community and were developed to solve specific topology optimization problems, with an implementation that mixes the analysis routines and the optimization formulation. Perhaps the main goal in these cases was to keep the code compact and simple, rather than to achieve the flexibility required to solve more general problems. Different versions of these codes are necessary if we need to change, for example, the finite element analysis (FEA) to deal with polygonal meshes, or if we want to change the optimization formulation, e.g. from compliance minimization to compliant mechanisms (Pereira et al. 2011).

The `PolyTop` code presented by Talischi et al. (2012b) features a modular Matlab structure to solve topology optimization problems; much like the 99- and 88-line codes (Sigmund 2001; Andreassen et al. 2010), the `PolyTop` code also has an educational focus. The code is structured to separate the analysis routine and the optimization algorithm from the specific choice of the topology optimization formulation. The finite element (FE) model and the topology optimization parameters are passed to the `PolyTop` kernel by a Matlab script, allowing the user to investigate different problems without changing the basic kernel. The formalism offered by this decoupling approach provides an easy way to extend, develop and modify the code to test different topology optimization formulations. Moreover, the `PolyTop` code addresses practical issues regarding use of polygonal meshes in arbitrary design domains (rather than boxes). This paper deals with the computational cost associated with polygonal meshes. Numerical results are presented using polygonal meshes with up to 120 thousand elements (Talischi et al. 2012b). However, when using the `PolyTop` code to deal with large-scale problems, it is difficult to have total control of memory allocation in Matlab. Efficient algorithms are necessary to build the filtering sparse matrix, which is used to link the design variables to the analysis parameters. The process of building this matrix causes a memory bottleneck in the current `PolyTop` code, while a performance bottleneck occurs because the linear system of equations within the finite element analysis module have to be solved.

The number of publications that focus on solving large-scale problems in topology optimization has increased considerably over the last few years – see, for example, the review paper by Deaton and Grandhi (2013). In 2012, Suresh introduced an algorithm for large-scale 3D problems in topology optimization (Suresh 2012), which is an extension of the 2D topological-sensitivity based method (Suresh 2010). The 3D model explores the congruence between hexahedron/brick finite elements and modern multi-core computer architectures. Suresh presented numerical results with 700 thousand dofs, which can be solved in 16 minutes in a CPU and, in 125 seconds in a GPU. He also obtained results for relatively large-scale problems with 15 million dofs, that required 19 hours using the CPU, and in 2 hours using the GPU. He solved an even greater problem, with 92 million dofs, which took 12 days of processing on the CPU (no results on the GPU were provided because of memory problems). However, the current model uses uniform grids (instead of unstructured meshes), which are susceptible to checkerboard pattern problems (Talischi et al. 2010), as mentioned previously. Recently, Aage and Lazarov implemented a parallel framework for topology optimization using the method of moving asymptotes (Aage and Lazarov 2013). They simulated fluids and solid mechanics problems with linear scalability up to approximately 800 CPUs on a Cray XT4/XT5 supercomputer. They presented results for a 3D mesh using linear hexahedral elements with almost 15M dofs spending approximately 5 minutes for each topology optimization iteration. Another recent work published by Amir et al. (2013) presents a computational approach to reduce the time for solving 3D structural topology optimization problems. They obtained performance improvement by exploiting the specific characteristics of a multigrid preconditioned conjugate gradient (MGCG) solver.

Despite recent practical results, many improvements are still required to develop an efficient and modular code, capable of dealing with different topology optimization problems. The PolyTop++ code contributes to this effort, addressing issues pertaining to the use of polygonal meshes in arbitrary design domains and offering a hierarchical modular structure that is easier to extend to different finite element solvers, different topology optimization formulations, and different physics.

2 Formulation, optimizer and modular code

The main goal of topology optimization is to find the most efficient material distribution inside a domain $\Omega \subseteq \mathbb{R}^N$. Many topology optimization formulations can be defined in the form:

$$\begin{cases} min \quad f(\rho, \mathbf{u}) \\ s.t.: \; g_i(\rho, \mathbf{u}) \leq 0, \qquad i = 1, \ldots, N_c \end{cases} \tag{1}$$

where f and g_i are, respectively, the objective and constraint functions, N_c is the number of constraints, and ρ represents the density function.

We consider linear elasticity to be the governing state equation, which is typical in continuum structural optimization. The solution $\mathbf{u} \in \mathcal{V}$ satisfies the variational problem

$$\int_{\Omega} m_E(\rho)\mathbf{C}\nabla\mathbf{u} : \nabla\mathbf{v}d\mathbf{x} = \int_{\Gamma_N} \mathbf{t}\cdot\mathbf{v}ds, \qquad \forall\mathbf{v}\in\mathcal{V} \tag{2}$$

$$\mathcal{V} = \{\mathbf{v}\in\mathbf{H}^1(\Omega;\mathbb{R}^d) : \mathbf{v}|\Gamma_D = 0\} \tag{3}$$

where m_E is a material interpolation function, $\mathbf{C}$ is the stiffness tensor, Γ_N and Γ_D are portions of $\partial\Omega$, where the non-zero traction $\mathbf{t}$ and the displacements are specified, respectively.

In this work, as well as in the original PolyTop implementation (Talischi et al. 2012b), we solve compliance minimization problems using the following discrete form:

$$\begin{cases} min \quad \mathbf{F}^T\mathbf{U} \\ s.t. : \ \frac{\mathbf{A}^T m_V(\mathbf{Pz})}{\mathbf{A}^T\mathbf{1}} - \overline{v} \leq 0 \end{cases} \tag{4}$$

where $\mathbf{F}$ is the nodal load vector, which is independent of the design variables $\mathbf{z}$, $\mathbf{U}$ is the nodal displacement vector that arise from the equilibrium equation $\mathbf{K}(m_E(\mathbf{z}))\mathbf{U} = \mathbf{F}$, and $\mathbf{K}$ is the global stiffness matrix. $\mathbf{A}$ is the vector of element volumes, m_V is the volume interpolation function, $\mathbf{P}$ is the matrix that maps the design variables $\mathbf{z}$ into the element variables $\mathbf{y}$ by $\mathbf{y} = \mathbf{Pz}$, $\mathbf{1}$ denotes an array consisting of unit entries, and $\overline{v}$ is an input parameter that defines the design volume fraction.

In the so-called SIMP (Solid Isotropic Material with Penalization) approach (Bendsoe 1989; Deaton and Grandhi 2013; Schmidt and Schulz 2012; Sigmund 2001; Talischi et al. 2010, 2012b; Andreassen et al. 2010) the element material properties are considered constant and the element densities are the design variables of the problem:

$$m_E(\rho) = \varepsilon + (1-\varepsilon)\rho^p, \qquad m_V(\rho) = \rho \tag{5}$$

where p is the penalty parameter, and ε the Ersatz parameter. The SIMP method is used here because this work is based on the PolyTop model (Talischi et al. 2012b). Moreover, this method is very popular and has been used successfully to solve a variety of problems. However, alternative well-known methods can also be found in the literature, such as the level-set (Deaton and Grandhi 2013; Gain and Paulino 2013; Osher and Fedkiw 2001; Osher and Sethian 1988), phase-field (Bourdin and Chambolle 2003; Deaton and Grandhi 2013; Gain and Paulino 2013), and topological-sensitivity Pareto-optimal (Suresh 2010, 2012) based methods.

Figure 1 describes the structure of the topology optimization method implemented in this work to solve the compliance minimization problem. The boxes inside the dashed lines in the diagram represent the main steps in the loop until the change between the element densities in subsequent iterations achieve the specified tolerance level and the final topology is obtained.

The diagram also shows the decoupling approach between the FE analysis stage, the constraint stage, and the interpolations functions chosen for mapping the sensitivities

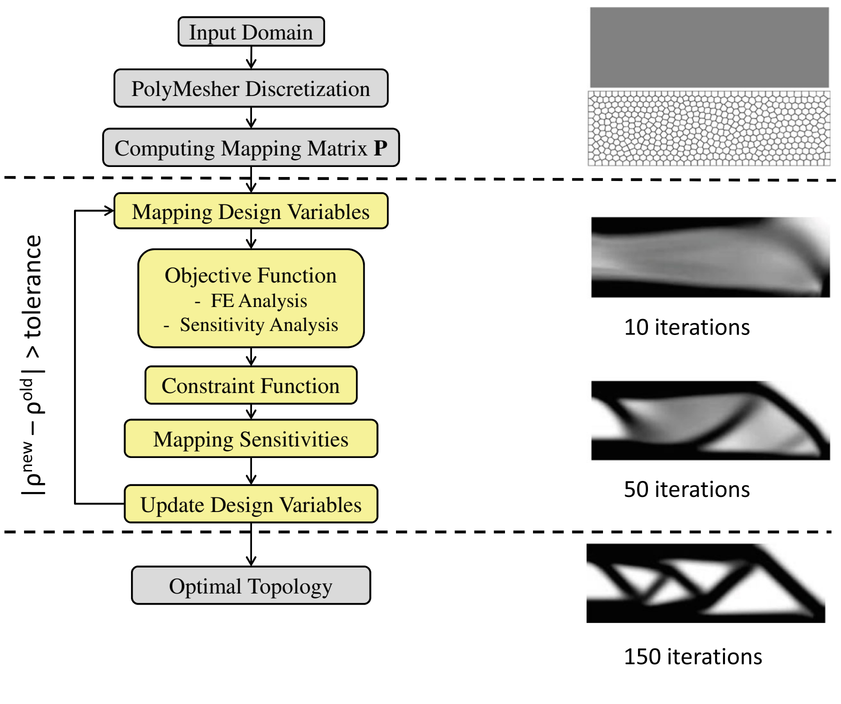


Fig. 1 Topology optimization steps for the compliance minimization problem

to the design variables. The structure of the discrete optimization problem allows separation of the analysis routine from the particular topology optimization formulation. *The analysis functions do not need to know about the choice of interpolation functions which corresponds to the choice of sizing parametrization or the mapping* **P** *that places constraints on the design space. Therefore, a general implementation of topology optimization in the context of this discussion must be structured in such a way that the finite element routines contain no information related to the specific topology optimization formulation*. Because of this, we define the analysis module as a collection of functions that compute the objective and constraint functions. This module comunicates with the finite element module to access the mesh information. An advantage of this approach is that the analysis functions can be extended, developed and modified independently of any modification to the topology optimization formulation.

In addition, certain quantities used in the analysis module, such as element areas **A** and local stiffness matrices $\mathbf{K}_e$, as well as connectivity of the global stiffness matrix **K**, need to be computed only once in the course of the optimization algorithm, as well as the mapping matrix **P**. In order to use a gradient-based optimization algorithm for solving the discrete problem, we must compute the gradient of the cost functions with respect to the design variables **z**. The sensitivity analysis can be separated along the same lines discussed above. The analysis functions compute the sensitivities of the cost functions with respect to their own internal parameters. Note that the analysis functions only compute the sensitivities of a certain variable g_i with respect to the internal parameters **E** and **V**, i.e.:

$$\frac{\partial g_i}{\partial \mathbf{z}} = \frac{\partial \mathbf{E}}{\partial \mathbf{z}}\frac{\partial g_i}{\partial \mathbf{E}} + \frac{\partial \mathbf{V}}{\partial \mathbf{z}}\frac{\partial g_i}{\partial \mathbf{V}} \tag{6}$$

In the compliance problem, $f = \mathbf{F}^T\mathbf{U}$, we have (Talischi et al. 2012b):

$$\frac{\partial f}{\partial \mathbf{E}_e} = -\mathbf{U}^T\frac{\partial \mathbf{K}}{\partial \mathbf{E}_e}\mathbf{U} = -\mathbf{U}^T\mathbf{K}_e\mathbf{U}, \quad \frac{\partial f}{\partial \mathbf{V}_e} = 0 \tag{7}$$

This means that the FE function that computes the objective function also returns the negative of the element strain energies as the vector of sensitivities $\partial f/\partial \mathbf{E}$. The remaining terms in (6) depend on the formulation, i.e., how the design variables **z** are related to the analysis parameters. For example, if $\mathbf{E} = m_E(\mathbf{Pz})$ and $\mathbf{V} = m_V(\mathbf{Pz})$ it implies that:

$$\frac{\partial \mathbf{E}}{\partial \mathbf{z}} = \mathbf{P}^T\mathbf{J}_{m_E}(\mathbf{Pz}), \quad \frac{\partial \mathbf{V}}{\partial \mathbf{z}} = \mathbf{P}^T\mathbf{J}_{m_V}(\mathbf{Pz}) \tag{8}$$

where $\mathbf{J}_{m_E}(y) := diag(m'_E(y_1), \ldots, m'_E(y_N))$ is the Jacobian matrix of map m_E, and $\mathbf{J}_{m_V}(y) := diag(m'_V(y_1), \ldots, m'_V(y_N))$ is the Jacobian matrix of map m_V. The evaluation of expression (6) is carried out outside the analysis routine and the result, $\partial g_i/\partial \mathbf{z}$, is passed to the optimizer to update the values of the design variables.

3 Linear system solvers

The linear solver plays a very important role in topology optimization because it is used in each step of the optimization. Usually, this stage becomes the performance bottleneck of the simulation. Accordingly, any improvement in solver performance will have a major impact on the overall performance of the system. For that reason, we focus on the finite element analysis stage to highlight the code's ability to deal with extensions and modifications by illustrating how to handle different linear solvers. We will provide a detailed description of the different linear solvers implemented in this work (for the finite element analysis).

We address issues concerning the use of polygonal meshes to develop efficient algorithms for the different solvers. Since we are solving elasticity problems, the corresponding coefficient matrix of the linear system is symmetric, positive-definite and usually sparse. We employed modules to handle direct and iterative solvers, assembling the global stiffness matrix, and also used an iterative 'matrix-free' solver, in which only the local stiffness matrix of each element is computed. All these solvers were implemented using serial and parallel computing. Table 1 presents the solvers used in this work.

3.1 Unsymmetric multifrontal package (UMFPACK)

The *UMFPACK* is a free package with a set of routines for solving sparse linear systems using the Unsymmetric MultiFrontal method (Davis and Duff 1997). We use its C interface to implement our serial direct solver for the FEA. Our code computes the local stiffness matrix of each element and then assembles the global stiffness matrix using the triplet format (Davis 2006). The solver package requires that the global stiffness matrix is stored in the compressed-column sparse (CCS) form (Duff et al. 1989); therefore our matrix triplet form is converted to the compressed-column form fusing the package routine *umfpack_dl_triplet_to_col*. Considering that we must update the

Table 1 Linear system solvers in PolyTop++

Solver	Type	Version
UMFPACK (Davis and Duff 1997)	Direct	Serial
PCG (Saad 2003)	Iterative	Serial
EbEPCG (Augarde et al. 2006)	Iterative	Parallel
PARDISO (Schenk and Gärtner 2004)	Direct	Parallel

element stiffness matrices at each iteration of the topology optimization process, and that the mesh topology remains unchanged, we use the triplet format as an easy way to store the global stiffness matrix. The nodal displacements, that arise from the solution of the linear system of equations are passed to the next stage of the topology optimization algorithm.

3.2 Preconditioned conjugate-gradient (PCG)

The preconditioned conjugate-gradient (PCG) solver was implemented using a Jacobi preconditioner (Saad 2003). The PCG is an iterative method for solving linear systems with symmetric and positive-definite matrices. For this solver, the global stiffness matrix is assembled by using the local stiffness matrix of the elements, as we did for the *UMFPACK*. However, in order to reduce the memory requirement, we used a different sparse format to store the global stiffness matrix. The class *cCRSMatrix* handles matrices in a compressed-row sparse (CRS) format (Davis 2006). As mentioned previously, since we have to update the stiffness of the finite elements at each iteration of the topology optimization looping, the class *cCRSMatrix* has auxiliary arrays to find the exact position of an element inside the sparse matrix in order to avoid rebuilding the entire matrix.

The Algorithm 1 shows the basic steps of the PCG method. We note that each step needs one matrix-vector product and a product between the diagonal of the sparse matrix and a vector. The *cCRSMatrix* class holds the matrix-vector products and its main method *Solve* implements the pseudo-code for solving the linear system of equations.

Algorithm 1 Conjugate gradient with preconditioner

```
$\mathbf{x}_0$ is an initial guess, $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
for $i = 1, 2, \ldots$ do
    Solve $\mathbf{M}\mathbf{w}_{i-1} = \mathbf{r}_{i-1}$
    $\rho_{i-1} = \mathbf{r}_{i-1}^T \mathbf{w}_{i-1}$
    if $i = 1$ then
        $\mathbf{p}_i = \mathbf{w}_{i-1}$
    else
        $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
        $\mathbf{p}_i = \mathbf{w}_{i-1} + \beta_{i-1}\mathbf{p}_{i-1}$
    end if
    $\mathbf{q}_i = \mathbf{A}\mathbf{p}_i$
    $\alpha_i = \rho_{i-1}/\mathbf{p}_i^T \mathbf{q}_i$
    $\mathbf{x}_i = \mathbf{x}_{i-1} + \alpha_i \mathbf{p}_i$
    $\mathbf{r}_i = \mathbf{r}_{i-1} - \alpha_i \mathbf{q}_i$
    if $\mathbf{x}_i$ accurate enough then
        quit
    end if
end for
```

Due to the ill-conditioned nature of the linear system of equations, this method requires many iterations to converge, as pointed out by Wang et al. (2007). Therefore, the matrix-vector product becomes the bottleneck in the overall simulation. However, this problem is overcome by using an iterative method where the matrix-vector product, which is the most expensive operation in the solution of the system, can be easily parallelized. Section 3.3 presents the parallel matrix-free PCG solver implemented here.

3.3 Element-by-Element PCG (EbEPCG)

One of the main contributions of this work is the efficient use of polygonal meshes. Our main goal is to enable the `PolyTop` framework to handle larger problems compared to the educational Matlab version. As we mentioned, this type of mesh is less susceptible to numerical instabilities such as checkerboard patterns; however, it may increase the computational cost of the system, which can make the solution of large-scale problems unfeasible. For this reason, the use of parallelism becomes indispensable.

Another issue encountered when solving large-scale problems is the computation of the global stiffness matrix (**K**). Assembling **K** is the main bottleneck of the current `PolyTop` system, as presented by Talischi et al. (2012b) and shown in Table 2. We present a new parallel algorithm of a 'matrix-free' PCG solver using polygonal meshes in the GPU. We have chosen the 'matrix-free' PCG solver presented by Augarde et al. (2006) because, in this method, the full matrix **K** is not required. Instead, only the element stiffness matrices are used to solve the linear system of equations. We know that the storage for all the individual element stiffness matrices will exceed the memory taken by the global assembled matrix. However, there are issues to using a global stiffness matrix to solve large scale problems: Using a direct solver with a global stiffness matrix is impracticable because of the extra memory space required for the factorization process (this is shown later in the results section). The use of an iterative solver with a global stiffness

Table 2 `PolyTop` code runtime profile for different numbers of elements. Assembling the global stiffness matrix **K** is the most expensive part of the code (Talischi et al. 2012b)

Mesh size	2.7 K	7.5 K	30 K	120 K
Assembling **K**	37.8 %	41.1 %	37.2 %	28,5 %
Solving **KU** = **F**	23.0 %	28.6 %	32.1 %	29.8 %
Mapping **z**, **E** and **V**	6.1 %	6.8 %	6.9 %	20.1 %
Update Design Vars.	1.1 %	2.1 %	2.8 %	1.4 %
Plotting	16.0 %	7.7 %	3.3 %	1.8 %
Others	15.7 %	12.8 %	17.7 %	18.4 %

matrix would be a valid alternative. In this paper, however, we have opted to use an element-by-element solver because of its flexibility and adequacy for an efficient parallel GPU-based algorithm, which is one of the focuses of the manuscript. It is important to mention that an ideal element-by-element solution for large problems would be to store no local stiffness matrix at all, and calculate the required member of the local matrix at the time it is needed. When using polygonal elements, the process of calculating the local stiffness matrix has a high computational cost; however, identical elements share identical local matrice, which allows us to simulate large-scale problems with small memory usage, as long as we can model the domain with multiple instances of a small set of elements.

Both serial and parallel 'matrix-free' PCG solvers for the FEA were used in the topology optimization algorithm and employed the same strategy to enable the evaluation and validation of the results.

3.3.1 Race condition

The race condition issue appears when two different parallel threads may need to be written in the same memory position. In the parallel implementation of the EbEPCG, the matrix-vector product computed in two different threads may have to give results in the same dof, as shown in Fig. 2. In a strategy where each thread is assigned to a finite element node, race conditions should never appear. However, this approach has the following disadvantages: the GPU's SIMD architecture cannot be altered because of the variable number of adjacent elements per vertex; the need to use an adjacency data structure to represent the mesh; the impossibileity of having a coalesced memory access for the local stiffness matrix. To address this problem, we use an approach based on graph coloring. The idea consists of computing the matrix-vector product for a set of elements that do not have any nodes in common. By considering enough groups (colors), the matrix-vector product required by the PCG solver can be computed with no race condition. We employed the greedy coloring algorithm (Gebremedhin et al. 2005) due to its simplicity and the small increase required in the preprocessing runtime. Figure 3 shows an example of a polygonal mesh colored with 6 different colors by the greedy algorithm. We use this algorithm in all parallel versions of the 'matrix-free' PCG solver.

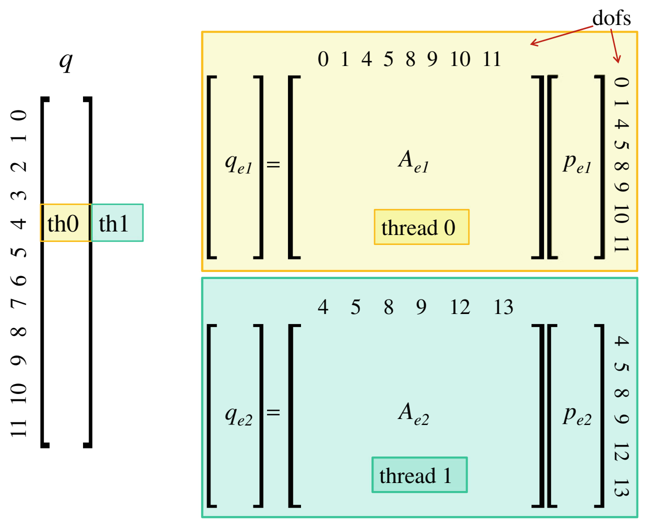


Fig. 2 Race condition on the matrix-vector product of two different elements computed in parallel by two different threads. The elements share common dofs and may have to write in the same memory position in array q

3.3.2 Naive parallel version

The standard approach for computing the matrix-vector product without assembling the global stiffness matrix is given by

$$\mathbf{U}_{new} = \mathbf{KU} = \sum_{e=1}^{N} \mathbf{K}_e \mathbf{U}_e, \tag{9}$$

where $\mathbf{K}_e$ and $\mathbf{U}_e$ are, respectively, the stiffness matrix and the displacement vector of the element e. The element based approach is attractive because the same instruction can be used to compute the matrix-vector product of all elements through multiple threads, which is consistent with the SIMD (Single Instruction Multiple Data) architecture of CUDA (Nvidia 2013). We tested this approach for the MBB beam problem (Messerschmitt-Bolkow-Blohm Olhoff et al. 1991) with a polygonal mesh with 5K elements and loading and support conditions as stated in Fig. 4. The computing platform used here is provided in Table 3. Unfortunately, Fig. 8 shows that the standard element approach is unfavorable for a parallel implementation; the performance of the naive parallel version is about 20 % better in the CPU, and about 70 % worse in the GPU, compared to the serial version.

Analysing the results of the naive parallel implementation, we identified the reason for the low performance of the parallel version in comparison to the serial one. The main problem occurs because concurrent threads process a different number of vertices per polygon, particularly in a

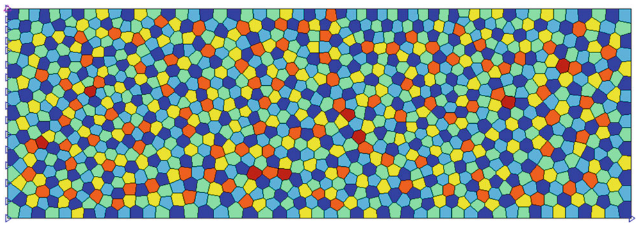


Fig. 3 Polygonal mesh colored by the greedy coloring algorithm (Gebremedhin et al. 2005) to avoid race condition

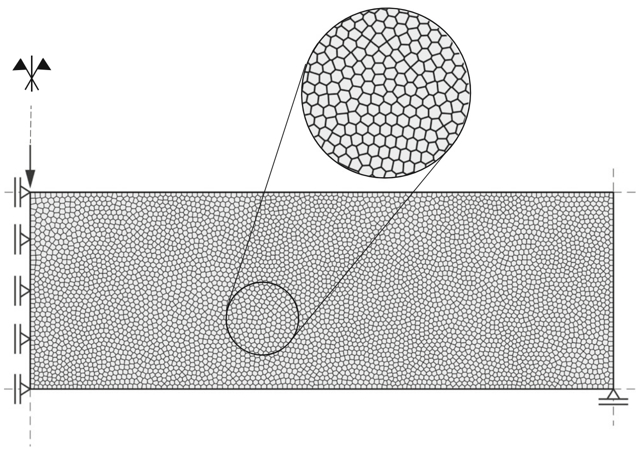


Fig. 4 MBB beam problem statement with its domain geometry, loading and boundary conditions

GPU that uses SIMD architecture, where concurrent threads should follow the same line of execution to reach maximum performance. In this example, we tested a polygonal mesh with 5K elements, composed of 27 4-gons, 1028 5-gons, 3325 6-gons, and 620 7-gons, where the different sizes of the stiffness matrices of each element increase the number of divergent branches in parallel computing. This problem is more critical in the GPU code, since in the CUDA model, the group of threads inside a warp must wait for the other threads that branched off to a different execution path. Moreover, when using elements with a different number of vertices, it is more difficult to access memory in an aligned way, and in a coalesced way, in order to take advantage of the cache in the CPU, and of the memory bandwidth in the GPU.

3.3.3 Optimized matrix-vector product

In polygonal meshes, different elements present a different number of nodes, which may cause low performance of parallel implementations due to the high divergency of branches and cache misses (concurrent threads do not access contiguous memory space). Therefore, our strategy consists in creating two main subdivisions of the mesh elements: the first one subdivides the elements by their colors, and the second one subdivides the elements by their number of vertices, as shown in Fig. 5.

The stiffness matrices of the elements are stored in a 1D global array, in such a way that all the columns of each element matrix are serially stored. Because all the element matrices are symmetric, only their lower triangular parts need to be stored. Figure 6 presents this special data storage in memory.

The parallel block execution uses each thread to compute the product $q_i = a_{i,j} p_j$, where, for a given element, $a_{i,j}$ are the components of the local stiffness matrix, p_j are the components of the local displacement vector, and q_i are the components of the local force vector. Notice that only the columns of the element matrices are accessed sequentially to compute the corresponding dofs. Initially, all threads compute the first column. Then, all threads switch simultaneously to the second column, and this procedure continues until all dofs are computed, as illustrated in Fig. 7.

Figure 8 shows the result of the optimized 'matrix-free' PCG solver for the parallel version, after the improvements presented here for the matrix-vector products. Our parallel algorithm implemented in the CPU is almost 1.8 times faster than the equivalent serial code, and 1.14 times faster than our naive parallel code (presented in a previous section), while the GPU version is approximately 20 times faster than the equivalent serial code in the CPU, and about 33 times faster than the naive parallel code implemented in GPU. We realize that comparing a serial code in the CPU with a

Table 3 Computing platform used to obtain the numerical results

Computing Platform	
O.S.	Windows 64-bit
Language	C++
CPU	Intel Core i7-2820QM
Clock	@2.30GHz
RAM	16.0GBs
Library	OpenMP (OpenMP Architecture Review Board 2011)
GPU	Nvidia GTX Titan
Clock	@837MHz
RAM	6.0GBs
Library	CUDA (Nvidia 2013)

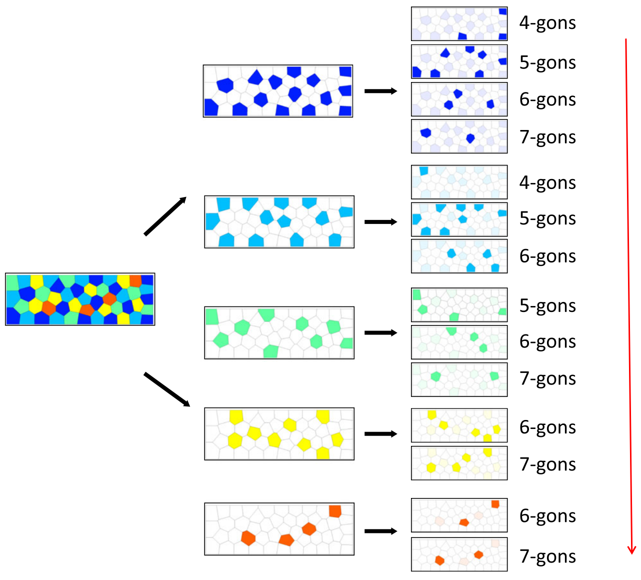


Fig. 5 Subdivision of elements by their colors and their number of vertices to optimize matrix-vector products

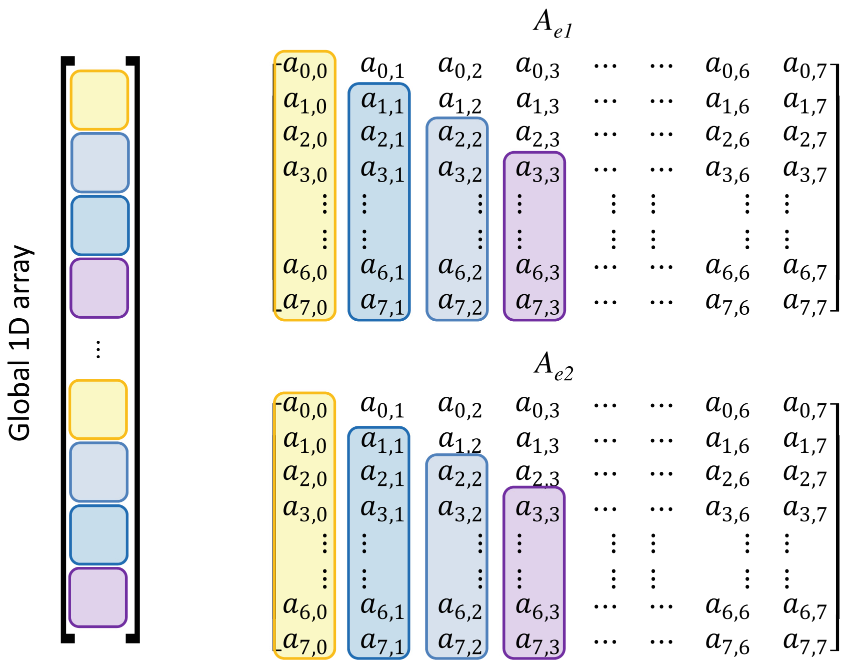


Fig. 6 Rearrangement of the stiffness matrices of the elements in a 1D global array to optimize access

parallel code in the GPU is not completely fair; however, it does show the performance gains that can be obtained when engineering applications move from the traditional serial paradigm to parallel computing. Because of this, we also compare our optimized parallel GPU code with parallel codes in the CPU and GPU (even with a naive approach), and the relative increase in speed is within the expected range based on similar works (Suresh 2013, 2012; Zegard and Paulino 2013; Papadrakakis 2011).

3.4 PARDISO

PARDISO (Schenk and Gärtner 2004) is a shared-memory multiprocessing parallel direct solver package. We use it as a parallel direct solver in the PolyTop++ code. The package can be used to solve large sparse symmetric and unsymmetric linear systems of equations. In this work, we have used it for solving sparse symmetric positive-definite matrices in the FEA, performed in each topology optimization iteration.

As in the PCG Solver, described in Section 3.2, this algorithm uses the class *cCRSMatrix* to store the sparse symmetric matrix in a compressed-row format. Again, we use the auxiliary arrays of the sparse matrix class to update the stiffness of the finite elements before running the FEA. The connectivity sparse matrix and its auxiliary arrays are computed from the global stiffness matrix stored in the triplet format, which is constructed just once before the topology optimization loop.

Considering that *PCG* is an iterative solver and *PARDISO* is a direct solver, we employ the modular feature of PolyTop++ to reuse code from one solver to the other. Thus we took advantage of the object-oriented C++ language to reuse the features already available in the *PCG* solver.

4 Results

This section presents the numerical results obtained by the PolyTop++ code using different types of meshes with different sizes, and is organized as follows:

- Validation results with respect to the Matlab code considering the compliance curves and the final optimal topologies for the MBB beam problem;
- Code profiling to check the performance bottlenecks;
- Performance results of the entire optimization process compared with the Matlab version;
- Performance of the different solvers, using meshes with more than 1 million polygonal elements;
- Optimal layout results for non-cartesian domains, comparing performance between GPU and CPU solvers;
- Results for the 3D Cantilever Beam problem using polyhedral and brick elements. For the latter, the same local stiffness matrix is shared among all elements to handle examples with 36M dofs in a single GPU.

For all examples presented here, we used the computing platform described in Table 3, while the finite element mesh is given by an input neutral file generated by `PolyMesher` Talischi et al. (2012a), a companion paper of the `PolyTop` (Talischi et al. 2012b). However, any other application could be used to generate the input data file for our C++ code.

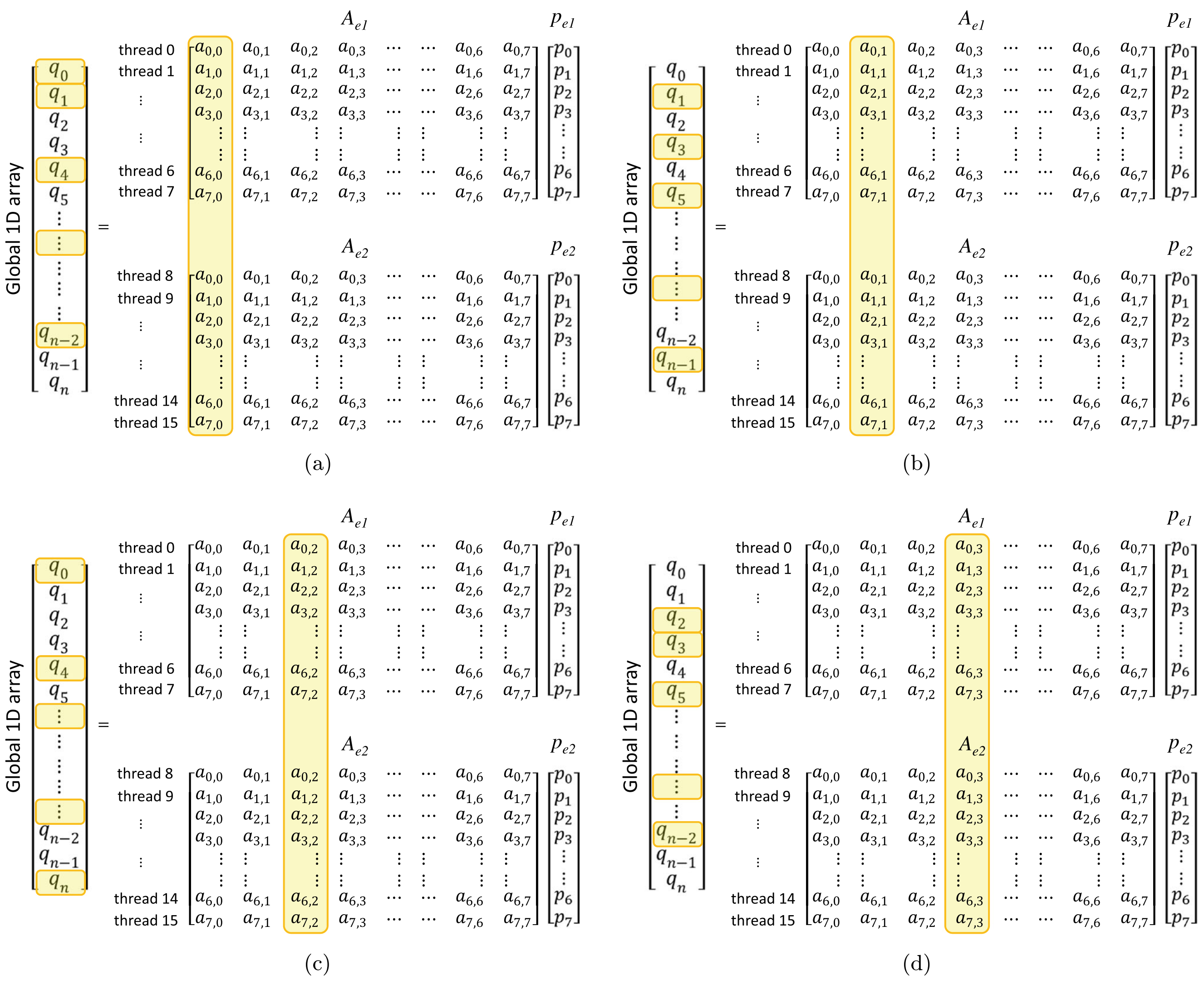


Fig. 7 The parallel block execution sequence. **a** All threads computing the first column. **b** All threads computing the second column. **c** All threads computing the third column. **d** All threads computing the fourth column

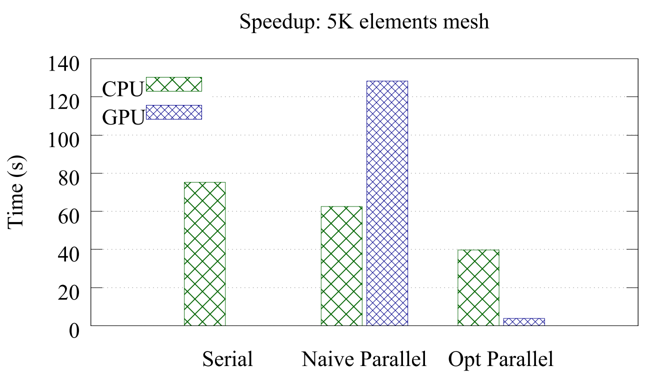


Fig. 8 Speedup values considering the serial, the naive parallel and the optimized parallel versions of the EbEPCG solver implemented in the CPU with OpenMP, and in the GPU with CUDA

4.1 Validation

We validated the code with the `PolyTop` results by comparing the compliance minimization curves and the final optimal topologies for the MBB beam problem defined in

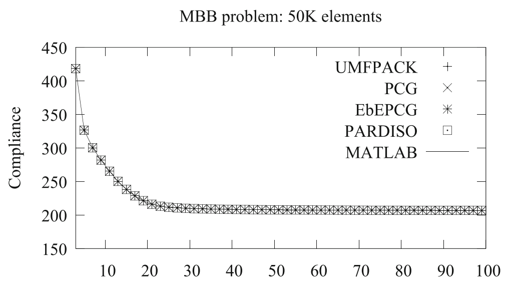
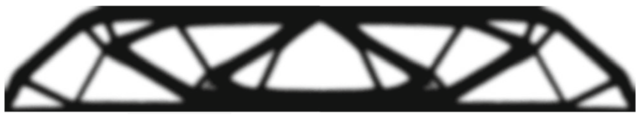


Fig. 9 Compliance minimization curves for the MBB beam problem by all PolyTop++ solvers and `PolyTop` (Talischi et al. 2012b)



(a)

n	*n-gon*	*number of n-gons*			
4		*143*	*0.3%*	*657*	*0.04%*
5		*10,280*	*20.6%*	*251,045*	*16.8%*
6		*32,542*	*65.0%*	*1,053,338*	*70.2%*
7		*6,967*	*13.9%*	*194,091*	*12.9%*
8		*68*	*0.2%*	*869*	*0.06%*
total		*50,000*		*1,500,000*	

(b)

Fig. 10 MBB beam problem results. **a** Final topologies using a mesh with 50K polygonal elements for both `PolyTop` and PolyTop++. **b** Mesh composition for 50K and 1.5M polygonal elements

Fig. 4. For this problem we use the same configuration parameters used by `PolyTop`, in the example described in Section 5 of Talischi et al. (2012b), except for the SIMP penalty parameter p and for the maximum number of iterations, set here as 3 and 200, respectively.

We used the MBB beam problem with a mesh composed by 50K polygonal elements using all the available solvers in the PolyTop++ code and the `PolyTop` code. The EbEPCG solver was implemented in parallel as described in Section 3.3 using the OpenMP library (OpenMP Architecture Review Board 2011). In Fig. 9 we observe that the compliance curves for all types of solvers are exactly the same as the `PolyTop` curve, as expected. In addition, the final topologies obtained by all PolyTop++ solvers are exactly alike if compared with `PolyTop`. Figure 10a shows the topology result, and Fig. 10b shows how heterogeneous our polygonal mesh is, composed mostly of 6-gons, 5-gons, and 7-gons. In Fig. 10b, we observe that the mesh composition distribution is almost equal, for the 50K element mesh and the 1.5M element mesh.

Table 5 Speedup between `PolyTop` and PolyTop++ (*PARDISO*)

Overall time (seconds) - 200 iterations

Mesh size	5 K	50 K	100 K	200 K
PolyTop++	8.8	176.9	466.9	1382.7
`PolyTop` (Talischi et al. 2012b)	50.7	628.1	1448.4	3413.6
Speedup	5.7	3.6	3.1	2.5

Code runtime for the MBB beam problem for 200 iterations.

4.2 Code profile

Analysing the performance results, we compared the runtime of the algorithm's main steps. This is important to identify the code bottleneck changes from the `PolyTop` runtime profile. Table 4 shows the PolyTop++ code runtime profile using the *PARDISO* solver for different mesh sizes. We chose this solver because it has similar properties to the Matlab solver, as both are parallel direct solvers.

Table 2 shows the `PolyTop` runtime code profile presented by Talischi et al. (2012b). We can observe that the main bottleneck of the `PolyTop` code is assembling the global matrix $\mathbf{K}$ (more than 35 % of the runtime). Comparing these results to Table 4, we notice that this is not the case in the C++ code due to the strategy adopted for computing auxiliary arrays to store the connectivity of the global stiffness matrix in a sparse level. Considering that the mesh connectivity remains intact during the entire simulation, we update the stiffness of the elements directly in the sparse matrix structure, avoiding rebuilding the global matrix in each iteration. As a result, the $\mathbf{K}$ matrix is assembled just once, together with the triplets computation, before the topology optimization loop.

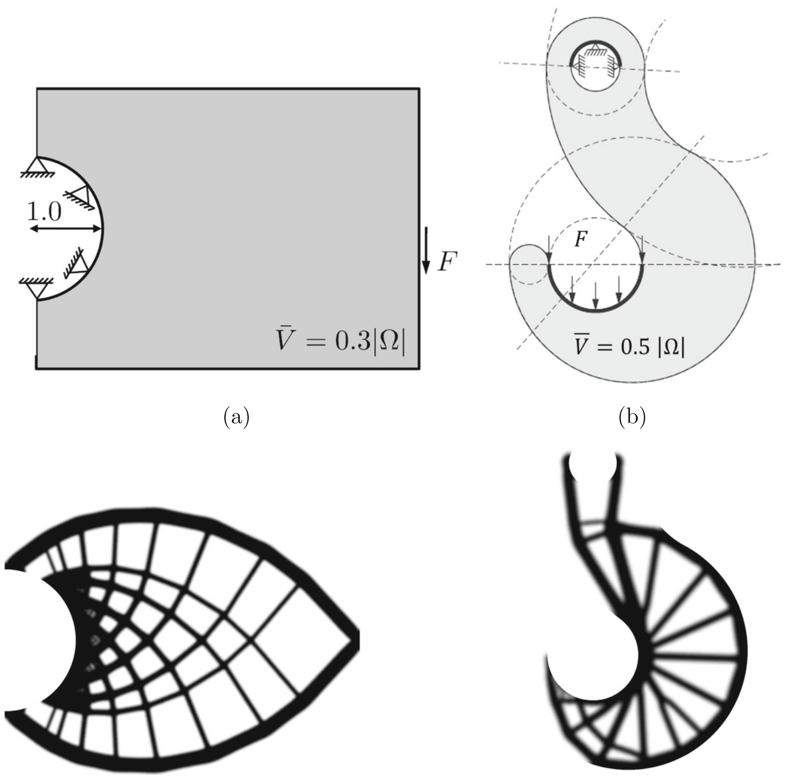
Table 4 Polytop++ code runtime profile for different number of polygonal elements using PARDISO (cf. Table 2)

Mesh size	5 K		50 K		100 K		200 K	
	time (s)	%	time (s)	%	time (s)	%	time (s)	%
Computing $\mathbf{P}$	0.1	1.2	10.5	5.9	42.4	9.1	191.1	13.8
Assembling $\mathbf{K}$	0.9	10.6	11.5	6.5	24.2	5.2	50.9	3.7
Solving $\mathbf{KU} = \mathbf{F}$	6.1	68.6	127.2	71.9	324.7	69.5	904.6	65.4
Update Design Vars.	1.1	12.3	11.1	6.3	21.6	4.6	45.7	3.3
Compliance sensitivity	0.5	5.5	12.0	6.8	38.4	8.2	131.5	9.5
Mapping $\mathbf{z}$, $\mathbf{E}$ and $\mathbf{V}$	0.07	0.8	3.88	2.2	14.0	3.0	55.8	4.0
Others	0.09	1.0	0.7	0.4	1.5	0.4	3.2	0.3
Total time	8.9		176.9		466.8		1382.8	

Table 6 Solver runtime for the MBB beam problem considering large meshes

Solver time (seconds)					
Mesh size	UMFPACK	PCG	EbEPCG	GPU EbEPCG	PARDISO
100K	117.3	124.0	94.3	10.7	4.1
200K	370.7	362.2	316.8	27.7	8.4
400K	1191.7	1022.5	901.2	75.0	20.3
500K	2106.2	1416.8	1260.0	114.6	26.5
1,000K	5760.3	4680.1	5760.0	372.6	**234.2**
1,500K	–	–	6840.4	**534.5**	–

The parallel direct solver (*PARDISO*) is much faster than the others. The direct solvers do not have enough memory for solving the linear system and the iterative PCG solver could not allocate the global stiffness matrix. The EbEPCG solver is the only option for large-scale problems.)



n	n-gon	number of n-gons			
4		487	0.11%	4,046	0.04%
5		89,207	17.8%	1,738,639	17.3%
6		343,735	68.7%	6,915,814	69.2%
7		66,215	13.3%	1,334,161	13.4%
8		356	0.09%	7,340	0.06%
total		500,000		10,000,000	

(e)

n	n-gon	number of n-gons	
4		494	0.11%
5		89,618	18.0%
6		342,942	68.5%
7		66,593	13.3%
8		353	0.09%
total		500,000	

(f)

Fig. 11 PolyTop++ solving the Michell cantilever problem (Talischi et al. 2010) and the Hook problem (Talischi et al. 2012b) with their final optimal layouts and meshes composition. **a** Michell cantilever problem. **b** Hook problem. **c** Michell optimal layout. **d** Hook optimal layout. **e** Michell mesh composition. **f** Hook mesh composition

Table 7 Solver runtime for the Michell cantilever problem (Talischi et al. 2010) considering large meshes

Solver time (seconds)			
Mesh size	EbEPCG	GPU EbEPCG	PARDISO
1M (4M dofs)	2676.1	**204.0**	306.3
5M (20M dofs)	29880.2	**2274.7**	–
10M (40M dofs)	72000.5	–	–

The GPU EbEPCG is faster than the PARDISO solver for a mesh with 1M of polygonal elements. The parallel direct solver (*PARDISO*) does not have enough memory for solving the linear system for larger mesh sizes.

4.3 Overall performance

Considering the total time to solve the entire topology optimization, we compare the total runtime using both PolyTop++ (*PARDISO* solver) and `PolyTop` codes for the MBB beam problem during 200 iterations for various mesh sizes (Table 5). All the 8 CPU cores were used for both applications. The increased speed achieved by the PolyTop++ code is more than 3 times that for the `PolyTop` model.

4.4 Solver performance

The main PolyTop++ bottleneck is the linear solver, as shown in Table 4. This step was already a bottleneck in the Matlab code, together with assembling the matrix **K**, as mentioned before. To analyze this step in more detail, we compare only the runtime of the linear solvers available in the C++ code. Table 6 presents the runtime for solving the linear system of equations, considering the very first iteration of the MBB beam problem. We used all 8 available cores, with 1 thread per core, for the parallel CPU solvers (EbEPCG and *PARDISO*). The *UMFPACK* library was used with its default parameters.

All the linear solvers give the same results for the node displacements. From Table 6, we notice that the parallel direct solver (*PARDISO*) is faster than the others. *PARDISO* includes a very efficient parallel implementation of a direct solver, requiring just a few seconds to solve meshes with hundreds of thousands of elements while the other solvers require several minutes to solve the same problem. We used the serial implementation of the *UMFPACK* solver, and we observed that the parallelism of the *PARDISO* solver makes a big difference compared to the *UMFPACK*. The iterative solvers, implemented in a serial way in the PCG and in a parallel way in the EbEPCG, have performances comparable to the *UMFPACK*.

We realize that comparing direct and iterative solvers is not fair, but the goal here is to show how different solvers can dramatically influence the feasibility of solving large-scale problems in topology optimization. Although direct solvers are faster and scalable, their memory consumption is usually bigger, so that they can be used only up to a certain mesh size. In this work, using 16.0GBs of RAM memory, the mesh size limit was 1M for polygonal elements (with just 8.0GBs of RAM the limit is 400K polygonal elements). Table 6 shows that the system does not have enough memory to solve for larger meshes using direct solvers. Notice that not even the iterative PCG solver could allocate the required memory; however, in this case, this was due to the memory required to store the global stiffness matrix.

In Table 6, we also demonstrate that for very large meshes of over 1 million polygonal elements, the ’matrix-free’ iterative solver is the only option. This type of solver requires only the local stiffness matrices, which makes a great difference in the memory footprint required. In contrast, our EbEPCG algorithm takes 1.6 hours to solve a linear system with 4 million dofs (1M elements) and almost 2 hours to solve a linear system with 6 million dofs (1.5M elements). It is important to mention that the results reported here in terms of computational time already takes into account the parallel solution developed in this work and described in Section 3.3. Our results

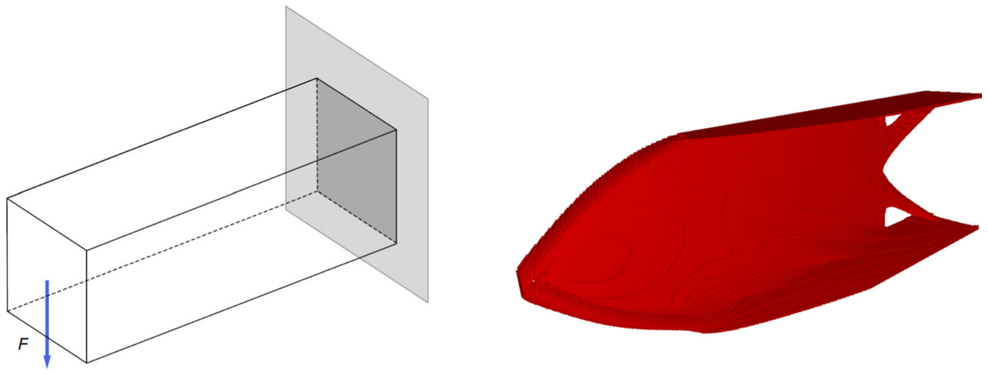


Fig. 12 Final topology layout for the 3D Cantilever Beam problem solved using PolyTop++. Due to the symmetry of the problem, only half of the domain is used. **a** 3D Cantilever Beam problem. **b** Optimal layout using a hexahedral mesh with 12M elements (36M dofs)

show the high computational cost for solving such large meshes and how important it is to use massive parallel machines like the GPU. The speedup achieved by our GPU version of the EbEPCG solver is so high that its runtime can be compared with the PARDISO parallel direct solver, when considering the 1M and 1.5M element meshes. By using a Nvidia GTX Titan with 6.0GBytes of RAM, the GPU EbEPCG solution is around 12 times faster than its respective CPU EbEPCG, which is also in parallel using OpenMP. These results show that our 'matrix-free' iterative solver is not just the only but also a feasible solution for solving very large polygonal meshes.

4.5 Non-cartesian domains

We also employ non-cartesian domains like the Michell cantilever problem (Talischi et al. 2010), and the Hook problem (Talischi et al. 2012b), as shown in Fig. 11. We use a mesh with 500K polygonal elements, a filter radius $R = 0.04$, a SIMP penalty parameter p from 1.0 to 4.0 with 0.25 of step, and 100 as the maximum number of iterations. The final topology layouts can be seen in Fig. 11c and d.

Considering only EbEPCG and PARDISO to solve the Michell cantilever problem with very large meshes, we obtained the very interesting result shown in Table 7: the iterative solver was faster than the direct solver. Our GPU EbEPCG solver achieved a speedup of 1.5 in relation to the PARDISO solver and a 13.1 increase in speed in relation to the CPU EbEPCG for a mesh with 1M of polygonal elements. Moreover, the PARDISO solver was not capable of solving a mesh with 5M polygonal elements using a CPU with 16.0GBs of RAM, while the GPU EbEPCG solver spent 37.9 minutes using a GPU with 6.0 GBs of memory.

4.6 3D examples

We extended the PolyTop++ framework to handle 3D examples. The Cantilever Beam problem was solved with our GPU EbEPCG solution using two meshes: one with polyhedron elements and other one with hexahedral. For the polyhedral case, we used Wachspress coordinates as defined in Floater et al. (2014) and analyzed in Warren et al. (2007). These coordinates are valid for elements that are not only convex but also simple. For the hexahedral case, we used only one local stiffness matrix shared by all elements, since they were all identical.

Figure 12 shows the geometry and boundary conditions of the Cantilever Beam problem and the results obtained for half of the domain using a corresponding hexahedral mesh with 12M elements (36M dofs). The symmetric Cantilever Beam using polyhedral mesh with 20K elements needs special techniques for its interpretation/visualization – this is work in progress. We used no filter, employed a SIMP penalty parameter p from 1 to 3 with 0.5 of increment, had a maximum number of 50 iterations and a volume fraction equal to 0.1.

To the best of our knowledge, the problem with 36M dofs is the largest ever solved for topology optimization using a single GPU. This result shows the ability of our element-by-element approach to handle large-scale problems, further when it is possible to share the local stiffness matrix between elements. The parallel computing in GPU solved this problem in 6 days and 6 hours. It is important to remember that this result was achieved by using the platform shown in Table 3, which is a single desktop with a single GPU. The evolution of the code to use a supercomputer cluster of GPUs could enable the solution of problems with larger sizes.

5 Conclusions

In this work, we have presented a topology optimization implementation that uses polygonal finite elements implemented in C++ language and in CUDA. It provides several features to handle the computational cost associated with the use of large-scale polygonal meshes. The modular feature provides an easy way to modify and extend the code to use different analysis routines without changing the topology optimization formulation.

We presented four types of linear solvers that can be used in the finite element analysis. The PolyTop++ (*PARDISO*) was approximately 3 times faster compared to the reference `PolyTop`. The *PARDISO* solver was the most efficient solver option in the code, solving topology optimization problems with 1M element meshes. However, the linear solver *EbEPCG* presented in this work is the only option when solving problems with even larger meshes. It was used with meshes composed of 1.5M polygonal elements (6 million dofs) and 10M polygonal elements (40 million dofs), with an increase in speed of 13 times between the GPU and the CPU versions. For the Mitchell cantilever problem with 1M elements, the *GPU EbEPCG* solver was 1.5 times faster than PARDISO.

The *GPU EbEPCG*, as a 'matrix-free' parallel linear solver, makes the use of very large polygonal meshes feasible in topology optimization problems. It presents a low memory footprint, different from the direct solvers, which were not capable of solving meshes with more than 1M elements. The parallelism solution appears to be a good way of speeding up the *EbEPCG* solver and the results indicate that larger meshes could be handled with more computational power.

We extended the PolyTop++ framework to handle 3D examples. The Cantilever Beam problem was solved with our GPU EbEPCG solution using a polyhedral mesh composed of 20K elements and a hexahedral mesh with 12M elements (36Mdofs). For the hexahedral case, we used only one local stiffness matrix shared for all elements, since they are all identical, and to the best of our knowledge, this is the largest problem ever solved for topology optimization using a single GPU.

The EbEPCG solver still needs improvements with regard to the high number of iterations required to achieve convergence. A better pre-conditioner may substantially reduce the computing time of the solver, like the multi-grid pre-conditioned conjugate gradients solver presented recently by Amir et al. (2013). Another potential for future work is a distributed version of the code to be used in a cluster of GPUs, taking advantage of the high computational power and available memory to provide an extension for solving even larger problems.

We extended the code by using iterative and direct solvers, and with serial and parallel implementations, like our parallel EbEPCG solver solution, to show its ability to easily change the analysis routines. We show that the linear solvers and even the entire finite element solution can be modified without any change in the topology optimization formulation. Moreover, the PolyTop++ handles large-scale problems in a much better fashion than the Matlab version of the code. Considering these improvements, we believe that PolyTop++ is a good alternative to be explored by the engineering community for practical problems and computational issues in topology optimization.

Acknowledgments We are thankful to the support from the US National Science Foundation under grant CMMI #1321661. We also acknowledge the support from the Donald B. and Elizabeth M. Willett endowment at the University of Illinois at Urbana–Champaign (UIUC). Any opinion, finding, conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the sponsors.

References

Aage N, Lazarov BS (2013) Parallel framework for topology optimization using the method of moving asymptotes. Structural and multidisciplinary optimization, pp 493–505

Andreassen E, Clausen A, Schevenels M, Lazarov BS, Sigmund O (2010) Efficient topology optimization in MATLAB using 88 lines of code. Struct Multidiscip Optim 43(1):1–16

Amir O, Aage N, Lazarov BS (2013) On multigrid-CG for efficient topology optimization. Struct Multidiscip Optim. doi:10.1007/s00158-013-1015-5

Augarde C, Ramage A, Staudacher J (2006) An element-based displacement preconditioner for linear elasticity problems. Comput Struct 84(31-32):2306–2315

Bendsoe MP (1989) Optimal shape design as a material distribution problem. Structural Optimization 202:193–202

Bourdin B, Chambolle A (2003) Design-dependent loads in topology optimization 1. Statement of the problem and of the main results. vol 9, no. January, p 19–48

Davis TA (2006) Direct methods for sparse linear systems. Society for Industrial and Applied Mathematic, Philadelphia

Davis TA, Duff IS (1997) An unsymmetric-pattern multifrontal method for sparse LU factorization. SIAM J Matrix Anal Appl 18(1):140–158

Deaton JD, Grandhi RV (2013) A survey of structural and multi-disciplinary continuum topology optimization: post 2000. Struct Multidiscip Optim

Duff IS, Grimes RG, Lewis JG (1989) Sparse matrix test problems. ACM Trans Math Softw 15(1):1–14

Floater M, Gillette A, Sukumar N (2014) Gradient bounds for Wachspress coordinates on polytopes. SIAM J Numer Anal 52:515–532

Gain AL, Paulino GH (2013) A critical comparative assessment of differential equation-driven methods for structural topology optimization. Struct Multidiscip Optim 48:685–710

Gebremedhin AH, Manne F, Pothen A (2005) What color is your Jacobian? Graph coloring for computing derivatives. J Soc Ind Appl Math 47(4):629–705

Nvidia (2013) Cuda programming guide 5.5. https://developer.nvidia.com/cuda-downloads

Olhoff N, Bendsoe MP, Rasmussen J (1991) On CAD-integrated structural topology design optimization. Comput Methods Appl Mech Eng 89:259–279

OpenMP Architecture Review Board (2011) OpenMP application program interface version 3.1. http://www.openmp.org/

Osher S, Fedkiw RP (2001) Level set methods: an overview and some recent results. J Comput Phys 169:463–502

Osher S, Sethian Ja (1988) Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. J Comput Phys 79:12–49

Papadrakakis M, Stavroulakis G, Karatarakis A (2011) A new era in scientific computing: Domain decomposition methods in hybrid CPUGPU architectures. Comput Methods Appl Mech Eng 200(13-16):1490–1508

Pereira A, Menezes IFM, Talischi C, Paulino GH (2011) An efficient and compact matlab implementation of topology optimization: application to compliant mechanism. In: Proceedings of the 32nd Iberian latin American congress on computational methods in engineering, Ouro Preto

Rozvany G, Querin O, Gaspar Z, Pomezanski V (2003) Weight-increasing effect of topology simplification. Struct Multidiscip Optim 25(5-6):459–465

Saad Y (2003) Iterative methods for sparse linear systems, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia

Schenk O, Gärtner K (2004) Solving unsymmetric sparse systems of linear equations with PARDISO. Futur Gener Comput Syst 20(3):475–487

Schmidt S, Schulz V (2012) A 2589 line topology optimization code written for the graphics card. Comput Vis Sci 14(6):249–256

Sigmund O (2001) A 99 line topology optimization code written in Matlab. Struct Multidiscip Optim 21(2):120–127

Suresh K (2010) A 199-line Matlab code for Pareto-optimal tracing in topology optimization. Struct Multidiscip Optim 42(5):665–679

Suresh K (2012) Efficient generation of large-scale pareto-optimal topologies. Struct Multidiscip Optim 47(1):49–61

Suresh K (2013) Generating 3D topologies with multiple constraints on the GPU. In: 10th World congress on structural and multidisciplinary optimization, pp 1–9

Talischi C, Paulino GH, Pereira A, Menezes IFM (2010) Polygonal finite elements for topology optimization : a unifying paradigm. Int J Numer Methods Eng 82:671–698

Talischi C, Paulino GH, Pereira A, Menezes IFM (2012a) PolyMesher: a general-purpose mesh generator for polygonal elements written in Matlab. Struct Multidiscip Optim 45(3):309–328

Talischi C, Paulino GH, Pereira A, Menezes IFM (2012b) PolyTop: a Matlab implementation of a general topology optimization framework using unstructured polygonal finite element meshes. Struct Multidiscip Optim 45(3):329–357

Talischi C, Pereira A, Paulino GH, Menezes IFM, Carvalho MS (2014) Polygonal finite elements for incompressible fluid flow. Int J Numer Methods Fluids 74:134–151

Wang S, de Sturler E, Paulino GH (2007) Large-scale topology optimization using preconditioned Krylov subspace methods with recycling. International journal for numerical methods in engineering, pp 2441–2468

Warren J, Schaefer S, Hirani AN, Desbrun M (2007) Barycentric coordinates for convex sets. Adv Comput Math 27:319–338

Zegard T, Paulino GH (2013) Toward GPU accelerated topology optimization on unstructured meshes. Structural and multidisciplinary optimization