Patient-Specific Cardiac Parametrization from Eikonal Simulations

Daniel Ganellari[1], Gundolf Haase[1], Gerhard Zumbusch[2], Johannes Lotz[3], Patrick Peltzer[3], Klaus Leppkes[3], and Uwe Naumann[3](✉)

[1] Institute for Mathematics and Scientific Computing, University of Graz, 8010 Graz, Austria
[2] Institut für Angewandte Mathematik, Friedrich-Schiller-Universität Jena, 07743 Jena, Germany
[3] Software and Tools for Computational Engineering, RWTH Aachen University, 52062 Aachen, Germany
naumann@stce.rwth-aachen.de

Abstract. Simulations in cardiac electrophysiology use the bidomain equations to describe the electrical potential in the heart. If only the electrical activation sequence in the heart is needed, then the full bidomain equations can be substituted by the Eikonal equation which allows much faster responses w.r.t. the changed material parameters in the equation. We use our Eikonal solver optimized for memory usage and parallelization. Patient-specific simulations in cardiac electrophysiology require patient-specific conductivity parameters which are not accurately available in vivo. One chance to improve the given conductivity parameters consists in comparing the computed activation sequence on the heart surface with the measured ECG on the torso mapped onto this surface. By minimizing the squared distance between the measured solution and the Eikonal computed solution we are able to determine the material parameters more accurately. To reduce the number of optimization parameters in this process, we group the material parameters and introduce a specific scaling parameter γ_k for each group. The minimization takes place w.r.t. the scaling γ. We solve the minimization problem by the BFGS method and adaptive step size control. The required gradient $\nabla_\gamma f(\gamma)$ is computed either via finite differences or algorithmic differentiation using `dco/c++` in tangent as well as in adjoint mode. We present convergence behavior as well as runtime and scaling results.

Keywords: Eikonal equation · Domain decomposition · Tetrahedral mesh · Parallel algorithm · Shared memory · Optimization · Algorithmic differentiation · Adjoints

Support from FWF project F32-N18, Erasmus Mundus JoinEUsee PENTA scholarship and Horizon 2020 Project-Nr.: 671697 MONT-BLANC 3. The computational results presented have been achieved in part using the Vienna Scientific Cluster (VSC).

© Springer Nature Switzerland AG 2020
V. V. Krzhizhanovskaya et al. (Eds.): ICCS 2020, LNCS 12137, pp. 290–303, 2020.
https://doi.org/10.1007/978-3-030-50371-0_21

1 Introduction

Simulations in cardiac electrophysiology (EP) use the bidomain equations consisting of two partial differential equations (PDEs) coupled nonlinearly by a set of ordinary differential equations which describe the intercellular and the extracellular electrical potential. Its difference, the transmembrane potential, is responsible for the excitation of the heart and its steepest gradients form an excitation wavefront propagating in time. The arrival time $\varphi(x)$ of this excitation wavefront at some point $x \in \Omega$ can be approximated by the simpler Eikonal equation with given heterogeneous, anisotropic velocity information $M(x)$. The domain $\Omega \subset \mathbb{R}^3$ is discretized by planar-sided tetrahedrons with a piecewise linear approximation of the solution $\varphi(x)$ inside each of them.

It is almost impossible to consider the bidomain equation for inverse problems such as determining the material parameters of a cardiac model. The Eikonal equation reduces the computational intensity of the bidomain equation significantly. It is much faster, and it can provide an activation sequence in seconds [18].

Recent work in [2,5,17] has shown that building an efficient 3D tetrahedral Eikonal solver for multi-core and SIMD (single instruction multiple data) architectures poses many challenges. It is important to keep the memory footprint low to reduce the costly memory accesses and achieve a good computational density on GPUs and other SIMD architectures with limited memory and register capabilities. In this paper we briefly address our algorithms for shared memory parallelization and global solution algorithms for a fast many-core Eikonal solver with a low memory footprint [3,4] which makes it suitable for the inverse problem we are considering.

Accurate patient specific conductivity measurements are still not possible. Current clinical EP models lack patient-specificity as they rely on generic data mostly obtained by generalized measurements done on dead tissues. One way to accurately determine these parameters is to compare the computed solution w.r.t. the Eikonal computed activation sequence on the heart surface with the measured ECG. By minimizing the squared distance between the measured solution ϕ^* and the Eikonal computed solution $\phi(\gamma, x)$ with material domains scaled by the parameter $\gamma \in \mathbb{R}^m$, we are able to determine the scaling parameters which may identify heart tissues with a low conductivity that might indicate a dead or ischemic tissue.

The minimization problem is solved by the steepest descent and the BFGS method. To compute accurate gradients up to machine accuracy, we use algorithmic differentiation (AD) [16] based on the AD tool `dco/c++` [12]. This paper provides mathematical and implementational details of the method followed by comparative results for the shared memory parallelization for the tangent as well as for the adjoint AD approach.

2 Eikonal Solver

We used the following variational formulation of the Eikonal equation [5]

$$\sqrt{(\nabla\varphi(x))^T M(x)\nabla\varphi(x)} = 1 \qquad x \in \Omega \tag{1}$$

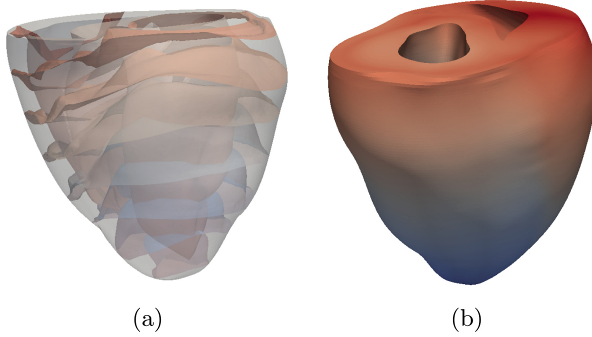
to model an activation sequence on the heart mesh. It is an elliptic non-linear PDE that does not depend on time and does not describe the shape of the wavefront but only its position in time. The solution of the Eikonal equation presents the arrival time of the wavefront at each point x in the discretized domain. In this context it is less accurate in comparison to the bidomain equations which describe both the shape and the position of the wavefront [18,20]. On the other hand, Eikonal equation provides a much lower computational intensity making it a good candidate for the inverse problem that we are considering in this paper.

Our Eikonal solver [3,4] builds on the fast iterative method (FIM) [1,10] by Fu, Kirby, and Whitaker [2] which is the state of the art for solving the Eikonal Eq. 1 on fully unstructured tetrahedral meshes with given heterogeneous, anisotropic velocity information M. We further improved the solver w.r.t. the algorithm, memory footprint and parallelization. A task based parallel algorithm for the Eikonal solver [5] is shown in Algorithm 1. The wavefront active list L is dynamically partitioned into sublists which are assigned for further processing to a number of processors. This strategy is particularly suitable for the OpenMP shared memory parallelization. On top of that we build our parallel automatic differentiation approach for computing the gradients needed to solve the inverse problem. The solution to Eikonal equation representing a wave front propagation is depicted in Fig. 1. Here we start with a single excitation point at the bottom of the domain and the isosurfaces of the solution of the Eikonal equation travel from the bottom to the top of the domain where are thereafter cut off.

Algorithm 1. Task based parallelism example

1: partition L dynamically into sub-sets L_i
2: launch kernel on GPU processors or start threads on thread i
3: **for all** vertices x in L_i **do**
4: $\quad \Phi_x = min(\Phi_x, Solver(t, \Phi))$
5: **end for**
6: wait for threads or processors to terminate

Detailed numerical results can be found in our previous work [5], showing that our code scales very well and provides an activation sequence in seconds. The low memory footprint [4] of the solver makes it well suitable for the inverse problem especially when considering an algorithmic differentiation approach where one of the main bottlenecks in the adjoint implementation is the increased memory footprint.



(a) (b)

Fig. 1. Arrival time $\varphi(x)$ ranging from 0 (bottom (a), blue (b)) to 1 (top (a), red (b)). (Color figure online)

3 Cardiac Parametrization

3.1 Eikonal Equation with Material Domains

According to physiology, the human heart consists of different tissues, such as the heart chamber, with different conductivity parameters. Let us denote these different material domains by $\overline{\Omega}_k$. Then our discretized domain can be expressed as

$$\Omega \quad \xrightarrow{tets} \quad \overline{\Omega}_h = \bigcup_{k=1}^{m} \overline{\Omega}_k, \qquad \Omega_l \bigcap_{k \neq l} \Omega_k = \emptyset \ . \tag{2}$$

The velocity information $M(x)$ is specific but constant in each tetrahedron $\tau \in \Omega_h$. An additional tag indicates to which material domain $\overline{\Omega}_k$ that element is assigned. It allows to scale the velocity information $M(x)$ in each material domain $\overline{\Omega}_k$ by some $\gamma_k \in \mathbb{R}$, i.e., the tetrahedron specific information is preserved but all tetrahedrons in the same material class have the same scaling parameter.

The decomposition of the domain into material domains leads to an Eikonal equation with material domains

$$\sqrt{(\nabla\phi)^T \gamma_k \cdot M \cdot (\nabla\phi)} = 1 \quad \forall x \in \Omega_k. \tag{3}$$

Now the activation time ϕ depends on the scaling parameters $\gamma \in \mathbb{R}^m$. Let us emphasize that this change does not affect the runtime of the Eikonal solver.

The domain Ω is statically partitioned into a number of non-overlapping sub-domains Ω_k, see Fig. 2, and a scaling parameter γ_k is assigned to each of them. We use two different ways to partition the domains into subdomains. The first approach uses ParMETIS to achieve an equal size subdomain partitioning and the second approach follows the physiology of the heart strictly. The latter produces currently worse optimization results, and therefore we focus on the first approach.

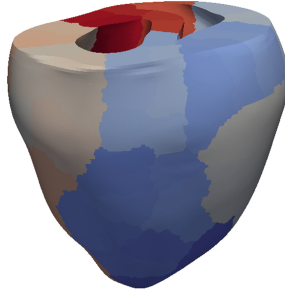


Fig. 2. Domain decomposition. Computational domain Ω and sub-domains Ω_i.

3.2 Optimization

Accurate patient specific conductivity measurements are still not possible. Current clinical EP models lack patient-specificity as they rely on generic data mostly obtained by generalized measurements done on dead tissues. One way to accurately determine these parameters consists in comparing the computed solution w.r.t. the Eikonal computed activation sequence on the heart surface with the measured ECG on the torso which is mapped onto the heart surface. Doing so one needs to solve a minimization problem with the objective functional as follows

$$f(\gamma) := \| \phi^*(x) - \phi(\gamma, x) \|^2_{\ell_2(\omega_h)}$$

with ω_h denoting the vertices in the discretization of Ω_h.

By minimizing the squared distance between the measured solution ϕ^* and the Eikonal computed solution $\phi(\gamma, x)$ with material domains scaled by the parameter $\gamma \in \mathbb{R}^m$, we are able to determine the scaling parameters which identify heart tissues with low conductivity that might indicate a dead or ischemic tissue. The scaling parameters γ do not only determine low conductivity tissues but also scale the generalized conductivity measurements to the accurate patient-specific conductivity parameters.

The mapping $\gamma \rightarrow \phi$ in (3) is nonlinear and nonconvex which results in a nonconvex optimization problem. It requires a good initial guess γ_0 and equal sized material domains as regularization. We consider generalized measurements done on dead tissues as an initial guess for the scaling.

The minimization problem is solved either by the steepest descent or the BFGS method. Both methods include an adaptive step size control and require the calculation of the gradient $\nabla_\gamma f(\gamma)$. The gradient calculation is done in two ways. First, we use finite differences as a verification method. The second approach uses AD that delivers gradients up to machine accuracy. We implemented a shared memory parallelization for both the tangent and the adjoint model. Adjoint shared memory optimizations are still ongoing work.

4 Implementation Details Using Algorithmic Differentiation

Algorithmic differentiation [7,16] is a semantic program transformation technique that yields robust and efficient derivative code. For a given implementation of a k-times continuously differentiable function $f : \mathbb{R}^n \rightarrow \mathbb{R}, y = f(\gamma)$ for $\gamma \in \mathbb{R}^n$ and $y \in \mathbb{R}$, AD generates implementations of corresponding *tangent* and *adjoint* models automatically. The tangent model computes directional derivatives $\dot{y} = \nabla_\gamma f \cdot \dot{\gamma}$, whereas the adjoint model computes gradients $\overline{\gamma} = [\nabla_\gamma f(\gamma)]^T \cdot \overline{y}$. The vectors $\dot{\gamma}, \overline{\gamma} \in \mathbb{R}^n$ and the scalars $\dot{y}, \overline{y} \in \mathbb{R}$ are first-order tangents and adjoints of input and output variables, respectively, see [16] for more details. The computational cost of evaluating either of both models is a constant multiple of the cost of the original function evaluation. This observation becomes particularly advantageous when computing the gradient of f for $n \gg 1$. In this case, instead of n evaluations of the tangent model to compute the gradient element by element, only one evaluation of the adjoint model is required. Nonetheless, it needs to be mentioned that the adjoint mode of AD usually requires a lot more memory, which makes further techniques like checkpointing [6], preaccumulation, or hybrid algorithmic/symbolic approaches necessary [15]. Higher derivatives can be obtained by recursive instantiations of the tangent and adjoint models. In the optimization problem defined in Sect. 3.2, only first derivatives are required.

In addition to the *scalar* modes presented above, *vector* modes can be used. The tangent scalar mode evaluates the tangent model once, i.e., performs one matrix-vector multiplication. The tangent vector mode, on the other hand, is able to perform multiple matrix-vector multiplications in one go, being more efficient than doing multiple scalar tangent evaluations by avoiding recomputation of temporary results. More details on scalar and vector mode can also be found in [16].

AD can be implemented either via source code transformation or operator overloading techniques. In this project, we use the AD overloading tool `dco/c++`. It has been successfully applied to a number of problems in, for example, computational fluid dynamics [19,23], optimal control [13], or computational finance [22]. `dco/c++` supports a wide range of features, two of which are used in this project: Checkpointing as well as advanced preaccumulation techniques on different levels (automatically on assignment-level, user-driven on higher level). Both techniques help to bring down memory requirements for the adjoint necessary to get a feasible code. The performance of AD tools is usually measured by the *run time factor*, which is the ratio of the runtime of one gradient computation w.r.t. one primal function evaluation, i.e.

$$\mathcal{F} = \frac{\text{Cost}(\nabla_\gamma f(\gamma))}{\text{Cost}(f(\gamma))} \ . \tag{4}$$

This factor and its scaling behavior is shown in Sect. 5.

As mentioned earlier, the Eikonal solver is OpenMP parallelized. Assuming that the original code is correctly parallelized, i.e., no data races, the (vector) tangent code can safely be run in parallel as well. Producing an adjoint of OpenMP

parallel code, on the other hand, is a major challenge since computing the adjoint requires a data flow reversal of the original code. Reversing the flow of data of parallel code introduces additional race conditions. Handling those automatically is not trivial and still actively researched [9,14]. We resolved this issue by splitting the code into active and passive segments, where only the active segment influences the derivative. As shown in Algorithm 1, in each parallel section, a minimum over many elements is computed. A simplified representation is given by $v = \min_i (S_i)$, where each S_i is calculated by an expensive solver call. Decomposing this into

$$k = \arg\min_i (S_i) \tag{5}$$

$$v = S_k \tag{6}$$

clearly shows, that we can compute Eq. 5 passively and only calculate Eq. 6 actively. This has the drawback that S_k is calculated twice. However, since only the data flow of the active segment needs to be reversed in adjoint mode AD, we can run the passive segment safely in parallel and keep the active segments sequential. This code version is referred to by the term *most-passive* in Sect. 5 where runtime results and scaling factors are analyzed. In contrast to this, the basic approach of running all segments actively is called *all-active*. The *all-active* version needs to run sequentially if computing adjoints. In case of running the tangent (vector) code, both, the *most-passive* and the *all-active* versions are executed fully in parallel.

5 Numerical Tests and Performance Analysis

5.1 Steepest Descent vs. BFGS

This section presents numerical results related to the `dco/c++` implementation for both, tangent and adjoint models, tested on Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60 GHz, 512 GB RAM. Initially we test with a small number of material domains and the coarsest mesh, Tbunny_C2, with 266,846 tetrahedrons in order to analyze the different behaviors of the steepest descent and BFGS. We test for $\gamma \in \mathbb{R}^6$, $\gamma_0 = (2\ 3\ 1\ 4\ 2\ 3)$ and convergence criteria $\epsilon = 0.5$ with an exact solution of $\gamma^* = (1\,1\,1\,1\,1\,1)$. The run time and the resulting γ are presented in Table 1. Tangent and adjoint are based on *all-active* version described in the previous section. It can be seen that the BFGS method is faster and converges in less iterations. Please note that the tangent (vector) model and the adjoint model have a similar run time for the small number of material domains. The adjoint model makes use of its advantage only for a larger number of parameters as shown later.

Table 1. Single-threaded run times in seconds and results.

Method	FD	Tangent	Adjoint	Result γ
Steepest descent	623	1570	1469	(1.003 0.980 1.003 0.958 1.002 1.227)
BFGS	194	684	696	(1.005 1.004 0.997 0.984 0.993 1.031)

Based on the actual physiology, the larger case has 21 material domains. The decomposition itself is currently not based on the physiology but using ParMETIS for an equally sized subdomain decomposition. Later on, we discuss what happens if we use a decomposition based on the physiology. The initial scaling was set to $\gamma_0 = \mathbf{1} \in \mathbb{R}^{21}$, i.e. the velocity information computed from the measurements done on dead tissues is not changing. Then we set the measured solution to be: $\gamma^* = (1\,1\,1\,1\,1\,1\,0.2\,0.2\,0.2\,0.2\,0.2\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1)$. The scaling parameters differ only in 5 subdomains in which we assume lower conductivity measured from the electrocardiogram.

The simulation in Fig. 3 shows in four steps how the material scaling changes throughout the optimization process. The whole optimization takes 19 iterations in total, but we only present four: the first, the last and two iterations in between to give an idea of the optimization process. Each subfigure represents the material scaling for each material domain of the Tbunny_C2 mesh. Here one can identify low conductivity tissue in the heart and in this case these material domains are exactly the ones in blue color as shown in the last step.

5.2 Gradient Verification and BFGS Convergence

To verify the AD implementation, we compare the gradient with finite differences results. The derivatives using tangent and adjoint AD are expected to be accurate up to machine precision. Since finite differences suffer from truncation and round-off errors a 'V'-shape is expected as error for varying perturbations h. As error, we compute the difference between finite difference and AD gradient, since we assume the AD gradient to be correct. This is indeed supported by observing the 'V'-shape as shown in Fig. 4a.

Results of running BFGS using the three different methods for derivative calculation is shown in Fig. 4. Though the gradients are expected to match up to machine precision for the two AD versions, BFGS shows different convergence behavior. Embedded into an iterative optimizer, the small differences in the gradients can lead to reaching a local stopping criterion earlier. This is the case here for iteration five. Nonetheless, both AD modes then fulfill the global convergence criteria of $\epsilon = 0.5$ after 16 and 18 iterations respectively. In contrast to that, the finite difference version fails to converge and requires a less strict convergence criteria.

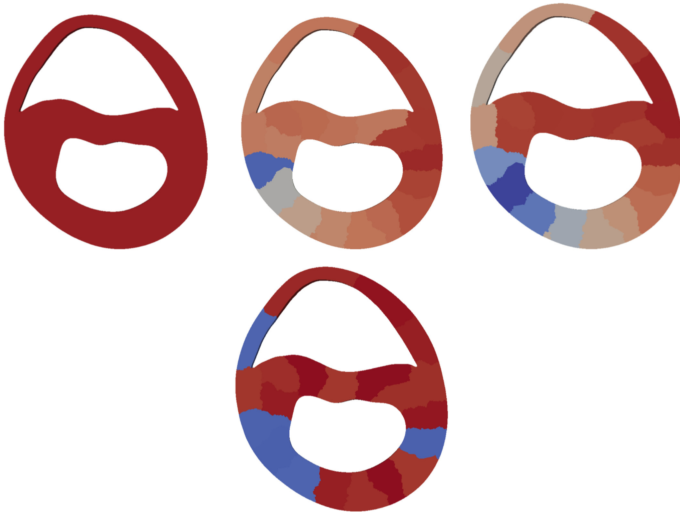


Fig. 3. Four steps from the simulation of the optimization using the BFGS algorithm with 21 material domains on the Tbunny_C2 mesh.

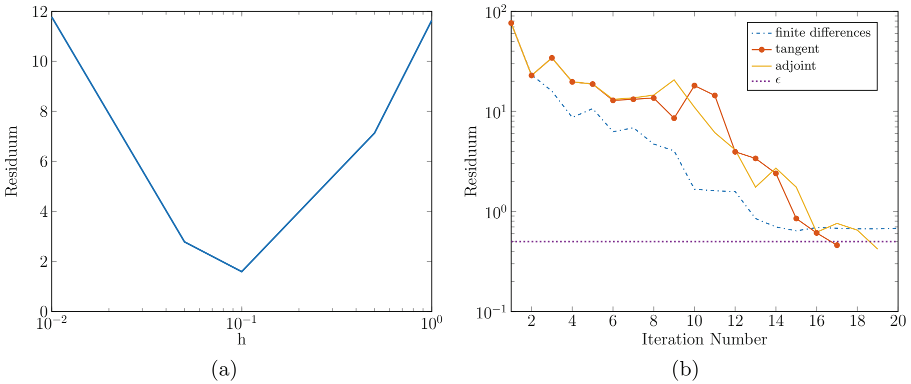


Fig. 4. Error of finite difference gradient (a) and convergence behavior of BFGS (b).

5.3 Sequential Timings for Gradient Computation

In the following, we measure the run time and the factor $\mathcal{F}$ (see Eq. 4) of a single gradient computation of the objective function. As introduced in Sect. 4, two different variants are implemented: *most-passive* and *all-active*. As shown in Fig. 5 tangent and adjoint codes give better run times for the *most-passive* version. In fact, the factor of the adjoint mode w.r.t. the primal run time was successfully improved from 23.8 to 2.7 in the *most-passive* variant which can be considered as very good. Besides this, the adjoint mode clearly wins over the tangent mode.

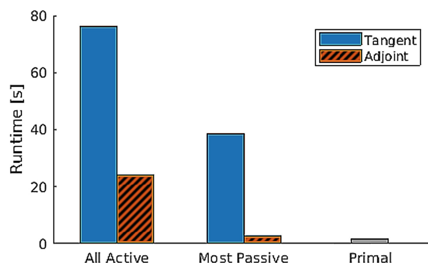


Fig. 5. Run time comparison for sequential gradient computations.

Table 2. Sequential run times for different tangent vector sizes.

Vector size	*all-active* [s]	*most-passive* [s]
7	50.9	9.6
11	54.0	8.6
21	50.6	6.5

While the above-shown run times correspond to the scalar tangent mode, measurements have also been made for the tangent vector mode. The gradient has been computed sequentially for both code variants for a vector size of 7, 11 and 21. To compute the full gradient, three tangent model evaluations are needed for a vector size of 7, two for 11, and only one for a vector size of 21. The runtime results are shown in Table 2. As the tangent vector mode by principle avoids some passive evaluations, a better runtime compared to the scalar mode can be expected. Therefore, the speedup compared to the previously shown tangent scalar run time of 76 s is reasonable. Since the problem size is a multiple of 7 and equals exactly 21, running with these two vector sizes perform the minimum amount of required tangent direction computations for the full gradient. Using a vector size of 11 ends up calculating one extra tangent direction, which results in slightly worse runtime. The *most-passive* variants on the other hand only run a small segment of the code actively. The slow-down due to additional directions only affect the active segment, which results in the run times shown in Table 2.

5.4 Scaling

For analyzing the OpenMP scaling behavior, we compare the AD versions to the scaling of the primal code, see Fig. 6a. Ideally, the AD code should show a similar scaling behavior as the primal. Since the *most-passive* implementation is the more efficient one, the following scaling analysis uses this version. Figure 6b contains scaling information for tangent and adjoint mode. One interesting aspect is the runtime factor for increasing number of threads. As one can see, the tangent and adjoint factors increase for more threads. This indicates a worse scaling behavior, which is reflected in the scaling factors. Recalling the introduction

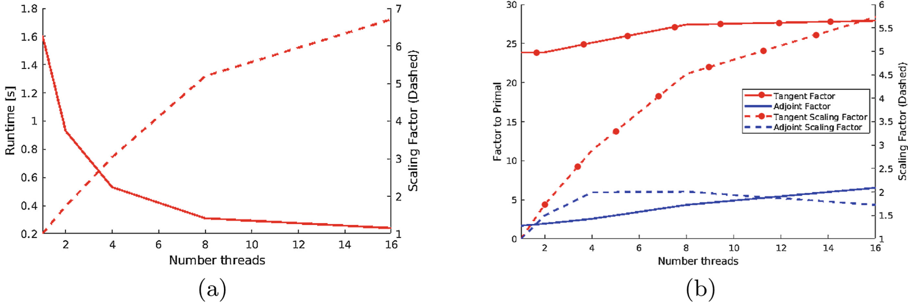


Fig. 6. Scaling behavior of the primal code (a) and scaling factors for the *most-passive* AD variant (b).

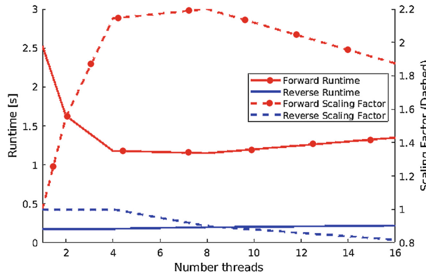


Fig. 7. Forward and reverse run time and scaling factor for the *most-passive* variant.

given in Sect. 4, the tangent (vector) mode can be fully parallelized. However, although the scaling is similar to the primal scaling, it is a little worse (6.7 compared to 5.7 with 16 threads). The adjoint code on the other hand can only be parallelized for the passive segment. For a further discussion, detailed adjoint measurements are shown in Fig. 7. An AD adjoint code by overloading consists of two sweeps, the forward and the reverse sweep. During the forward sweep, a data structure is build up (the *tape*) while executing the overloaded function. This data structure is then used during the reverse sweep to actually propagate adjoints. The reverse sweep can not be parallelized at all, i.e. the scaling factor is expected to be one. However, due to data allocated on different physical memory banks (first-touch policy during forward sweep), multithreading eventually slows down the reverse section. Since the run time of the reverse sweep is shorter than of the forward sweep, the impact on the overall scaling behavior is so as well. As stated before, the forward sweep timings include the execution of the passive segment, which explains a descent scaling. However, at some point (in this case already with 8 threads), the sequential active segment, running in an OpenMP critical section, leads to stalling of threads, which results in a decreasing scaling factor. Combining this insight for forward and reverse mode explains the adjoint scaling factor displayed in Fig. 6b.

6 Conclusions and Future Work

Patient-specific simulations in cardiac electrophysiology require patient-specific conductivity parameters which are not accurately available in vivo. In this article, we have shown algorithms and a proper software stack for improving the given conductivity parameters by minimizing the squared distance between the measured solution and the Eikonal computed solution. An efficient parallelized Eikonal solver has successfully been coupled with the AD tool `dco/c++` to not only speed up the optimization, but also to get more accurate answers. As demonstrated by carrying out numerical run time tests, the scaling behavior of the tangent model implementation is similar to the original scaling behavior. The adjoint model on the other hand only scales up to a small amount of cores with the current approach. Nonetheless, small to medium sized problems can be processed much faster using the adjoint model implementation.

Further work needs to be done towards a scalable adjoint model implementation not only for use with OpenMP but special care needs to be taken for getting a working GPU adjoint. Usually, adjoint GPU code cannot be generated automatically but has to be written by hand. Nonetheless, there exist tools that support the developer in writing adjoint GPU code, e.g., using source transformation [8] or overloading [11]. None of those tools are as automatic as state-of-the-art tools for CPU code currently are. In addition, we are considering an analytic approach to compute the gradient using the adjoint-state method [21]. The analytic approach shall not suffer from excessive memory requirements as does the adjoint model using AD tool [16]. From the problem definition perspective, the decomposition with equally sized subdomains where the regularization holds is not only much more accurate, but it also converges in fewer iterations. It is preferable to the pure physiological decomposition. Mapping the equal sizes subdomains to the physiology is ongoing work. Possible solution strategies could be:

- Refine the physiologically based material domain decomposition.
- Interpret the information $\frac{dy}{dM}$ returned without additional costs from the adjoint model.

The first idea relates to the equally sized subdomains. It means that we split the larger subdomains into smaller subdomains similar to the smallest one and preserve the same scaling parameter therein to end up with equally sized subdomains in which case the regularization holds. The second idea tells us what the problem is and leads the way to a more global regularization which shall apply in all cases. The information returned without additional costs from the adjoint model, $\frac{dy}{dM}$, shows how each component of the velocity information changes for each tetrahedron. This sensitivity analysis from the detailed element-wise information shall give us hints for an improved regularization.

References

1. Fu, Z., Jeong, W.K., Pan, Y., Kirby, R.M., Whitaker, R.T.: A fast iterative method for solving the Eikonal equation on triangulated surfaces. SIAM J. Sci. Comput. **33**, 2468–2488 (2011)
2. Fu, Z., Kirby, R.M., Whitaker, R.T.: Fast iterative method for solving the Eikonal equation on tetrahedral domains. SIAM J. Sci. Comput. **35**(5), C473–C494 (2013)
3. Ganellari, D., Haase, G.: Fast many-core solvers for the Eikonal equations in cardiovascular simulations. In: International Conference on High Performance Computing Simulation (HPCS), pp. 278–285. IEEE (2016). https://doi.org/10.1109/HPCSim.2016.7568347. peer-reviewed
4. Ganellari, D., Haase, G.: Reducing the memory footprint of an Eikonal solver. In: International Conference on High Performance Computing Simulation (HPCS). IEEE (2017, accepted)
5. Ganellari, D., Haase, G., Zumbusch, G.: A massively parallel Eikonal solver on unstructured meshes. Comput. Visual. Sci. **19**(5), 3–18 (2018). https://doi.org/10.1007/s00791-018-0288-z
6. Griewank, A., Walther, A.: Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. ACM Trans. Math. Softw. **26**(1), 19–45 (2000). https://doi.org/10.1145/347837.347846
7. Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Other Titles in Applied Mathematics, vol. 105, 2nd edn. SIAM, Philadelphia (2008)
8. Hascoët, L., Pascual, V.: The Tapenade automatic differentiation tool: principles, model, and specification. ACM Trans. Math. Softw. **39**(3), 20:1–20:43 (2013)
9. Hückelheim, J., Hovland, P., Strout, M.M., Müller, J.D.: Reverse-mode algorithmic differentiation of an OpenMPparallel compressible flow solver. Int. J. High Performan. Comput. Appl. **33**(1), 140–154 (2019). https://doi.org/10.1177/1094342017712060
10. Jeong, W.K., Whitaker, R.T.: A fast iterative method for Eikonal equations. SIAM J. Sci. Comput. **30**, 2512–2534 (2008)
11. Leppkes, K., Lotz, J., Naumann, U., du Toit, J.: Meta adjoint programming in C++. Technical report, AIB-2017-07, Department of Computer Science, RWTH Aachen University (2017)
12. Lotz, J.: Hybrid Approaches to Adjoint Code Generation with dco/C++. Dissertation, RWTH Aachen University (2016)
13. Lotz, J., Naumann, U., Hannemann-Tamas, R., Ploch, T., Mitsos, A.: Higher-order discrete adjoint ODE solver in C++ for dynamic optimization. Procedia Comput. Sci. **51**, 256–265 (2015)
14. Lotz, J., Naumann, U., Sagebaum, M., Schanen, M.: Discrete adjoints of PETSc through `dco/c++` and adjoint MPI. In: Wolf, F., Mohr, B., an Mey, D. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 497–507. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40047-6_51
15. Naumann, U., Lotz, J., Leppkes, K., Towara, M.: Algorithmic differentiation of numerical methods: Tangent and adjoint solvers for parameterized systems of nonlinear equations. ACM Trans. Math. Softw. **41**(4), 1–21 (2015)
16. Naumann, U.: The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation. Society for Industrial and Applied Mathematics, Philadelphia (2012)

17. Noack, M.: A two-scale method using a list of active sub-domains for a fully parallelized solution of wave equations. J. Comput. Sci. **11**, 91–101 (2015)
18. Pezzuto, S., Kal'avský, P., Potse, M., Prinzen, F.W., Auricchio, A., Krause, R.: Evaluation of a rapid anisotropic model for ECG simulation. Frontiers Physiol. **8**, 265 (2017). https://doi.org/10.3389/fphys.2017.00265
19. Sagebaum, M., Gauger, N.R., Naumann, U., Lotz, J., Leppkes, K.: Algorithmic differentiation of a complex C++ code with underlying libraries. Procedia Comput. Sci. **18**, 208–217 (2013)
20. Sali, A.: Coupling of monodomain and Eikonal models for cardiac electrophysiology. Master's thesis (2016). https://opus4.kobv.de/opus4-zib/frontdoor/index/index/docId/6051
21. Taillandier, C., Noble, M., Chauris, H., Calandra, H.: First-arrival travel time tomography based on the adjoint-state method. Geophysics 74(6)(2009)
22. du Toit, J., Lotz, J., Naumann, U.: Adjoint algorithmic differentiation of a GPU accelerated application. http://www.nag.co.uk/Market/articles/adjoint-algorithmic-differentiation-of-gpu-accelerated-app.pdf
23. Towara, M., Naumann, U.: A discrete adjoint model for OpenFOAM. Procedia Comput. Sci. **18**, 429–438 (2013)