A PARALLEL FINITE ELEMENT SOLVER FOR QUASI-STATIC MUSCLE DEFORMATION

JAVIER A. ALMONACID[*]

Abstract. Recent advances in understanding how muscles deform have led to a new, more informative mathematical model that views muscles as composite biomaterials, drawing a connection to the deformation of solids. This model is based on a three-dimensional, highly nonlinear system of partial differential equations. The current numerical approximation uses a second-order finite element method, in which the assembling process is done using the WorkStream design pattern in a shared-memory environment. Unfortunately, the degrees of freedom in the system can quickly grow to the point where the memory of a machine may not store such an amount of information, or that it may take too long for the linear system to solve. This is especially true when the mesh is constructed from magnetic resonance imaging (MRI) data. In this work, a parallel solver for quasi-static skeletal muscle deformation is discussed. As a stepping stone, a distributed version of a solver for Neo-Hookean solid deformation is constructed and scaling analyses are performed on this code. This new code is designed to run on a distributed architecture, building upon the existing deal.II implementation. The decomposition of the mesh is achieved through the p4est library, while the solution of the corresponding system of equations is coded using the Trilinos library. We also describe an updated script that can be used to install deal.II and link all the necessary dependencies.

Key words. muscle physiology, finite element method, deal.II, high-performance computing

AMS subject classifications. 65-04, 65Y05, 65M60, 74L15

1. Introduction.

The human musculoskeletal system consists of multiple soft tissues arranged in complex architectures. Some of these tissues are orders of magnitude stiffer than the others. Moreover, muscle fibres are capable of nonlinear dynamics (activation) prompted by electrochemical signalling from the central nervous system, in addition to the soft-tissue mechanical behaviour they exhibit. There is particular interest by physiologists in the question of the functional role that the architecture (arrangement of tissues) and regionalization of activation play in locomotion. Thus, *in silico* experiments offer an alternative to more traditional approaches (such as *in*

[*]Department of Mathematics, Simon Fraser University, Burnaby, BC, V5A 1S6, Canada (javiera@sfu.ca).

vivo and *in vitro* experiments) to answer these types of questions.

1.1. Muscles are not one-dimensional. Many models of muscle deformation that are in use today consider muscle as a one-dimensional spring containing a damping element (known as *contractile element*), where mass and inertial effects are assumed to be negligible. This approach dates back to the 1930s (when A. V. Hill [8] introduced it) and, due to its simplicity, it is still used today in popular biomechanics software such as OpenSim [7]. However, recent interdisciplinary developments have revealed that many of the aspects not considered in Hill-type models, such as mass distribution, size, history and three-dimensional structure have a significant effect in force, work, and power development during some types of muscle contractions [13, 14]. Therefore, not only there is a need to develop three-dimensional models that can address these issues but also to develop software that can efficiently and accurately solve the underlying equations.

1.2. Model in study and computational aspects. This work deals with the model developed in [18]. It represents a step forward from more case-specific approaches (such as [4] and [15]) in that the continuum-mechanics based approach allows to model the physical principles that relate muscle shape and force for muscles of any shape and size. The model considers muscle, aponeurosis, and tendon as anisotropic, fibre-reinforced, composite and nearly incompressible Neo-Hookean materials. The tissues that surround these components (such as water, blood, and collagen) are considered as a single nonlinear isotropic material: the *base material*. This is therefore a simple yet coherent representation of the complex structure of a muscle (see Figure 1).

Computationally speaking, the model in [18] is solved using a second-order finite element method, implemented in C++ using the library deal.II v8.5 [3]. The solver operates in a mostly serial fashion, with some *embarrasingly parallel* loops (such as the assemble of the global stiffness matrix and load vector) implemented using the design pattern WorkStream [17], thus allowing for multi-threading in a shared memory environment. Unfortunately, the multiplicities of length scales and highly heterogeneous

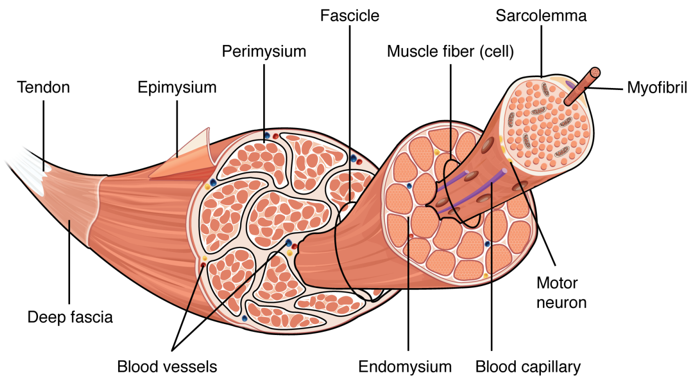

FIG. 1. *Typical structure of skeletal muscle.*

tissue material properties imply that the numerical simulations have to be performed (ideally) on highly refined meshes.

In [18], the authors only used about 128,000 quadrature points for the muscle block experiments and about 37,000 for the geometry of the medial gastrocnemius (MG) (hence, the number of degrees of freedom in the system is much lower). It can be seen in Figure 2 that the mesh for the MG had to be severely subsampled from the data collected using magnetic resonance imaging (MRI), an act that may lead to an important lose of accuracy. Therefore, there is a need for developing computational techniques that allow users to efficiently handle meshes with a large number of elements. Moreover, as the number of degrees of freedom in the system increases, the solution of the linear system becomes a major bottleneck (this part currently does not contain any parallelization). Because larger meshes may not fit in the memory of a single machine or the linear system may take too long to solve, an efficient algorithm must include a way to partition the mesh and the linear system into smaller pieces that can be solved faster.

1.3. Objectives. Motivated by the above, the aim of this project is to modify the current implementation so that it can run in a distributed memory environment, using features already existent in deal.II. We remark that this project does not seek to construct a new finite element method, but rather to build on the existing one, adapting typical steps like assemble of the stiffness matrix and mesh partitioning to the parallel setting. It is expected that this project will be completed in three major

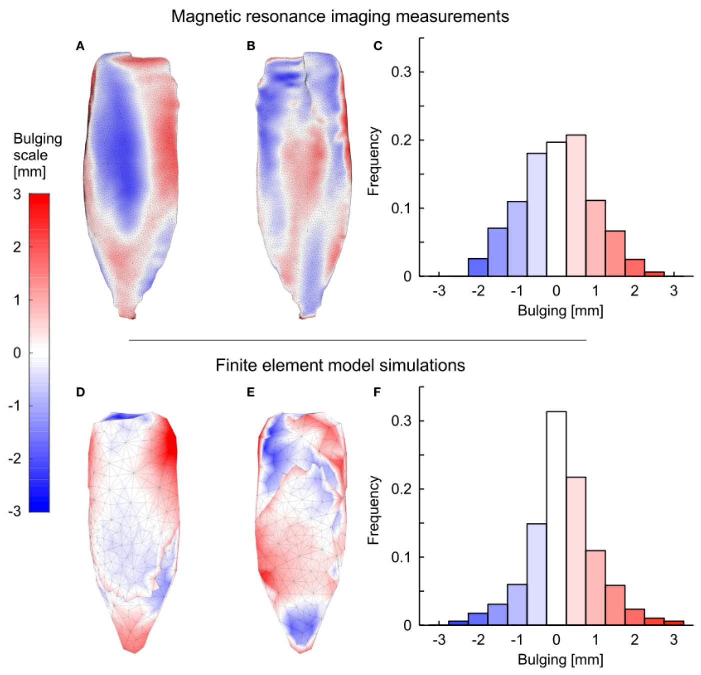

FIG. 2. *Muscle bulging in the medial gastrocnemius during fixed-end contraction.* ***(A-C)*** *show MRI data and* ***(D-F)*** *show data predicted by FEM model for the MRI-derived geometry. Red and blue shades indicate outwards and inwards bulging, respectively.* ***(A,D)*** *show the superficial surface of the muscle whereas* ***(B,E)*** *show the deep surface. The proportions (frequency) of the points on both surfaces that showed different magnitudes and directions of bulging are shown in* ***(C,F)****. Source: [18, Figure 12]. Used with permission.*

steps.

1.3.1. Software setup. The software (deal.II v8.5) must be recompiled for its use with the external libraries p4est [6] (a play on the expression parallel forest that describes the parallel storage of a hierarchically constructed mesh as a forest of quad- or oct-trees) and Trilinos [16] (that takes care of the linear algebra aspects). One way to do this is through the shell script deal.II/candi [11], however, since the script has been deprecated in favour of more recent versions of deal.II, some source files will need to be modified to account for changes in repository locations. This would result in an updated version of deal.II/candi that will be made available to the public. While this step is loosely related to the contents of the course, this new script is nevertheless an interesting byproduct of this project, specially when considering that deal.II *is not* backwards compatible.

1.3.2. Adaptation of "Step 44" for distributed objects. Two of the main advantages of using deal.II are its flexibility and availability of documentation, for which the series of tutorials (see [2]) play a crucial role. The programmer can be introduced to the software by following a series of tutorials, each one with a name of the form "Step N", $N \geq 1$.

The code developed in [18] (hereafter called the *NML code*) corresponds to an extension of Step 44, which deals with the quasi-static deformation of a Neo-Hookean solid, a model that is closely related to the one described in subsection 1.2. Hence, before proceeding with the much more complex NML code, the second objective of this project is to parallelize Step 44 for its use in a distributed memory environment. The main tools to be used here are Step 40 and Step 55. The first one corresponds to techniques for the massively parallel solution of the Laplace equation and sets the basis for distributing triangulations and matrices. In turn, the second one focuses on how to handle block matrices and block vectors (since the solution vector contains the unknowns of one vector field and two scalar fields) in the context of the Stokes equations.

First, the mesh will be partitioned in several subdomains and distributed among several processes. The idea is that each processor knows only part of the mesh and some adjacent cells (ghost cells). deal.II provides the class `parallel:distributed::Triangulation` in which the link to the p4est library is established. Communication between processes is done through MPI messaging, which is implemented in the `Utilities::MPI` namespace.

Next, the assemble process (i.e., the process in which contributions of each cell are transferred to the global stiffness matrix) is addressed. Care must be taken in that each process loops only over the cells that it "owns" (i.e., are stored in its memory), and that at the end, it reduces the contributions. The linear algebra in this problem will be dealt via the Trilinos library, for which deal.II provides wrappers in the `TrilinosWrappers` namespace. Indeed, distributed matrices and vector types are implemented here, as well as solvers and preconditioners. Finally, each processor will export the results in `.pvtu` Paraview format. The method

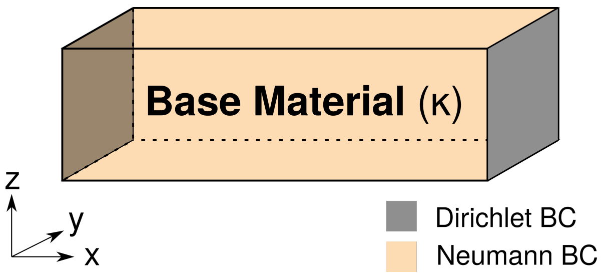

FIG. 3. *Sketch of the computational domain Ω and its boundaries. The block of muscle is filled with the base material (of bulk modulus κ), representing the different soft tissues that may be included in the model.*

`DataOutBase::write_pvtu_record` provides a way to write a "master record" that will allow Paraview to glue the subdomains into a single figure.

To assess the performance of the new code, we compute the result to a benchmark problem and perform a strong scaling analysis using the cluster `plato`. Then, we compare the results to those obtained with the original code.

1.3.3. Basic considerations when converting the NML code into distributed memory. The conversion of Step 44 into a distributed-memory setting will serve as a template to perform the same action on the NML code for quasi-static deformation of skeletal muscle. It is worth noting that some parts of the code, like those related to parameter processing, will remain sequential. In turn, terminal outputs will have to be mapped to one of the processes (say, rank 0) using the class `ConditionalOStream`. In this part of the project, we analyze what needs to be modified in the code for its use with distributed elements, using the information gathered from the conversion of Step 44.

1.4. Outline. In Section 2, the mathematical model used in the NML code is presented, along with some details of the current computational implementation. Then, in Section 3, we describe the update of the deal.II/candi script. Next, in Section 4 we present the conversion of Step 44 using the Message Passing Interface, what changed from the original code, and a strong scaling analysis of the code. Finally, in Section 5, we briefly talk about the considerations when converting the NML code into distributed memory, to then in Section 6 draw some conclusions about this project.

2. Mathematical and algorithmical background.

Let $\Omega = \Omega_t$ be a region in $\mathbb{R}^3$ representing the state of the muscle at time $t \geq 0$, whose boundary has been partitioned as $\partial\Omega = \Gamma_N \cup \Gamma_D$ (see Figure 3). The model for a quasi-static deformation of the tissues seeks a displacement $\mathbf{u}(\mathbf{x}, t)$, a pressure $p(\mathbf{x}, t)$, and a dilation $J(\mathbf{x}, t)$ such that

$$-\nabla \cdot \boldsymbol{\sigma}(\mathbf{u}, p, J) = \mathbf{f} \quad \text{in } \Omega_t \times (0, T], \tag{2.1a}$$

$$J - \det(\mathbf{I} + \nabla\mathbf{u}) = 0 \quad \text{in } \Omega_t \times (0, T], \tag{2.1b}$$

$$p - \frac{\kappa}{2}\left(J - \frac{1}{J}\right) = 0 \quad \text{in } \Omega_t \times (0, T], \tag{2.1c}$$

subject to boundary conditions:

$$\begin{aligned} \boldsymbol{\sigma}\mathbf{n} &= \mathbf{t} \quad \text{on } \Gamma_{N,t} \times (0, T], \\ \mathbf{u} &= \mathbf{d} \quad \text{on } \Gamma_{D,t} \times (0, T], \end{aligned}$$

and initial conditions:

$$\mathbf{u} = \mathbf{u}_0, \quad \mathbf{v} = \mathbf{v}_0, \quad p = p_0, \quad J = J_0 \quad \text{in } \Omega_0. \times \{t = 0\}.$$

Here, $T > 0$, $\mathbf{I}$ denotes the identity tensor, $\kappa > 0$ is the bulk modulus (which may be piecewise-constant), $\mathbf{t}$ and $\mathbf{d}$ are a prescribed traction and displacement, respectively and the stress tensor $\boldsymbol{\sigma}$ takes the form:

$$\boldsymbol{\sigma}(\mathbf{u}, p, J) = \boldsymbol{\sigma}_{vol}(p, J) + \boldsymbol{\sigma}_{iso}(\mathbf{u}). \tag{2.2}$$

Here the volumetric and isometric stresses take the form

$$\boldsymbol{\sigma}_{vol}(p, J) := p\mathbf{I}, \quad \boldsymbol{\sigma}_{iso} = \boldsymbol{\sigma}_{base} + \boldsymbol{\sigma}_{tissue},$$

where

$$\boldsymbol{\sigma}_{base}(\mathbf{u}) := 2Js_{base}\sum_{k=1}^{3} kc_k(\operatorname{tr}\overline{\mathbf{B}} - 3)^{k-1}\overline{\mathbf{B}},$$
$$\overline{\mathbf{B}} := J^{-2/3}(\mathbf{I} + \nabla\mathbf{u})(\mathbf{I} + \nabla\mathbf{u})^{\mathtt{t}},$$

and

$$\boldsymbol{\sigma}_{tissue}(\mathbf{u}) := 2J\,\overline{\mathbf{B}}\,\partial_{\overline{\mathbf{B}}}E_{tissue}\,(\overline{\mathbf{B}}),$$
$$E_{tissue}(\overline{\mathbf{B}}) := s_{tissue}\sum_{k=1}^{3} c_k(\operatorname{tr}\overline{\mathbf{B}} - 3)^{k}.$$

The constants c_k, s_{base}, s_{tissue} are known and depend on the base material properties of the muscle. Hence, by looking at how $\boldsymbol{\sigma}$ is defined in (2.2), the nonlinearity of the model becomes evident.

2.1. Solution algorithm and numerical methods. First, the finite element used to solve the system (2.1) corresponds to a classic choice in finite element analysis (see, e.g., [9]). The displacement is approximated using continuous piecewise-quadratic polynomials, while the pressure and dilation are approximated by discontinuous piecewise-linear polynomials. We denote the solution and this space by

$$\boldsymbol{\Xi} := (\mathbf{u}, p, J) \in Q_2 \times DGPM_1 \times DGPM_1.$$

In three-dimensions, this finite element contains 68 degrees of freedom per hexahedral cell: 60 for the displacement (20 per component), 4 for the pressure and 4 for the dilation. This yields a second-order method.

The variational formulation corresponding to the system (2.1) is linearized using Newton's method at a continuous level, i.e., it is not the nonlinear but the sequence of linear systems that are solved numerically. Next, to solve each one of the linear systems, a static condensation is first performed (discussed right after) and then the system is solved using a conjugate gradient method along with an symmetric successive over-relaxation (SSOR) preconditioner. All integrals are computed using a Gaussian quadrature rule of order 3.

2.1.1. Static condensation. After performing a Newton discretization of the nonlinear system, the following system needs to be solved for the increment $d\boldsymbol{\Xi}$:

$$K(\boldsymbol{\Xi}_i)\, d\boldsymbol{\Xi} = F(\boldsymbol{\Xi}_i)$$

where

$$K(\boldsymbol{\Xi}_i) := \begin{bmatrix} K_{uu} & K_{up} & 0 \\ K_{pu} & 0 & K_{pJ} \\ 0 & K_{Jp} & K_{JJ} \end{bmatrix}, \quad d\boldsymbol{\Xi} := \begin{pmatrix} du \\ dp \\ dJ \end{pmatrix}, \quad F(\boldsymbol{\Xi}_i) := \begin{pmatrix} F_u(\mathbf{u}_i) \\ F_p(p_i) \\ F_J(J_i). \end{pmatrix}$$

In this case, the discontinuous approximations for p and J yield matrices K_{pJ}, K_{Jp}, and K_{JJ} that are block diagonal. Therefore, the fields p and J can be easily expressed on each cell simply by inverting a local matrix and multiplying it by the local right hand side. Because all these operations are local, they are a good candidate for parallel computation. To condense the results and recover a classical displacement-based method, the following operations need to be done at a local level

$$\begin{aligned} dp &= K_{Jp}^{-1}(F_J - K_{JJ}\, dJ), \\ dJ &= K_{pJ}^{-1}(F_p - K_{pu}\, du), \\ \Rightarrow dp &= K_{Jp}^{-1} F_J - \bar{K}(F_p - K_{pu}\, du), \end{aligned}$$

where $\bar{K} := K_{Jp}^{-1} K_{JJ} K_{pJ}^{-1}$, and thus we solve the system

$$K_{con}\, du = F_u - K_{up}(K_{Jp}^{-1} F_J - \bar{K} F_p),$$

where $K_{con} := K_{uu} + \bar{\bar{K}}$ and $\bar{\bar{K}} := K_{up} \bar{K} K_{pu}$. Notice that this represents a decrease in the computational cost of the method since it avoids the inversion of the entire block matrix K.

2.2. Structure of the NML code. The NML has developed a C++ code based on the library deal.II v8.5 [3] that can be used in a shared-memory (SM) environment.

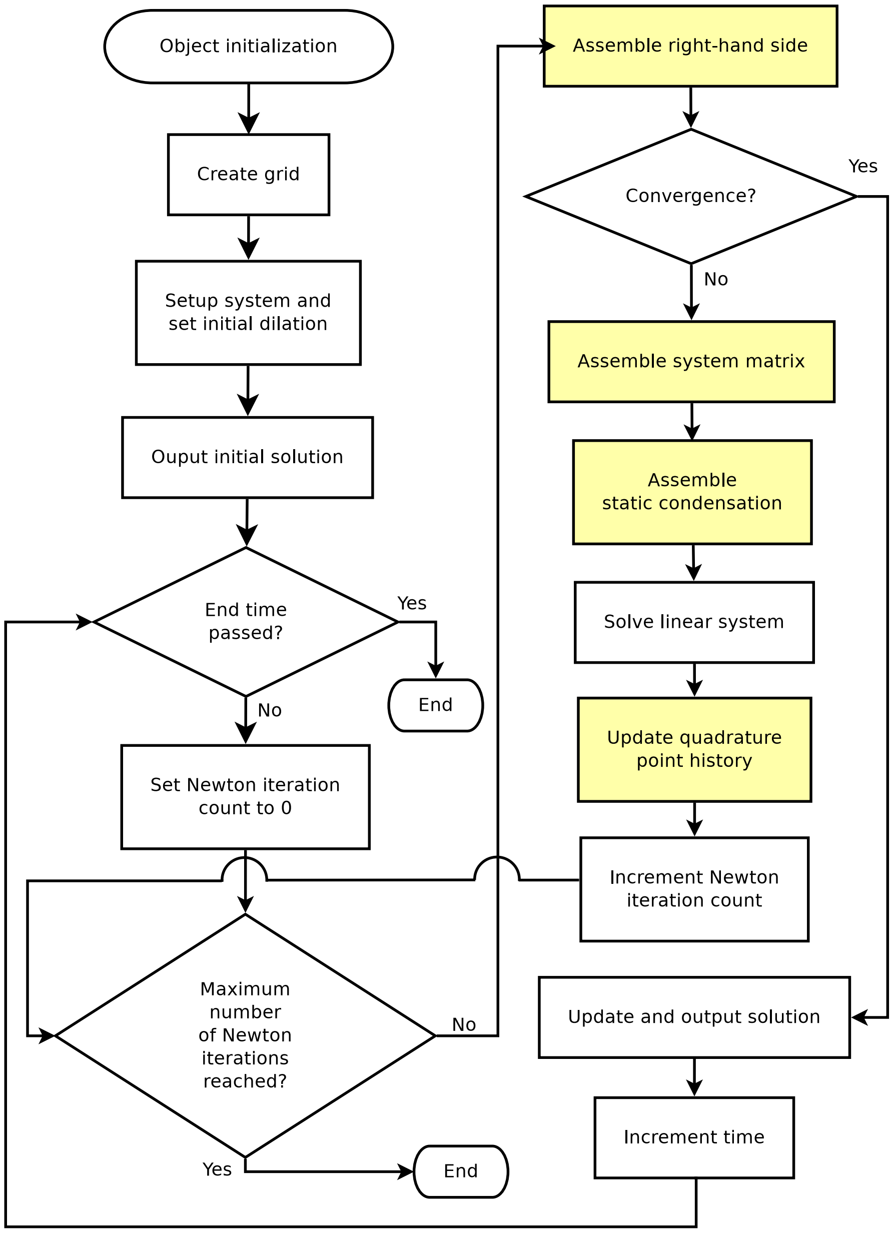
Object initialization
Create grid
Setup system and set initial dilation
Ouput initial solution
End time passed?
Yes
End
No
Set Newton iteration count to 0
Maximum number of Newton iterations reached?
No
Yes
End
Assemble right-hand side
Convergence?
Yes
No
Assemble system matrix
Assemble static condensation
Solve linear system
Update quadrature point history
Increment Newton iteration count
Update and output solution
Increment time

FIG. 4. *Execution flowchart of the code developed by the NML (it follows the same structure as Step 44). The methods shown here belong to the* `Solid` *class. Coloured boxes correspond to functions that are run in parallel using WorkStream.*

This code is as an extension of the deal.II tutorial (cf. [2]) for quasi-static deformation of a Neo-Hookean solid: the “Step 44” tutorial. It is written using object-oriented programming where three main classes can be distinguished:

1. `Muscle_Tissues_Three_Field` which declares and defines all the material

properties of the muscle, aponeurosis, and tendon. In addition, this class defines getters and setters to compute functional properties, such as the stress tensors defined in (2.2).

2. `PointHistory` which stores data at quadrature points. Each quadrature point holds a pointer to a material property. It also creates and initializes an object of type `Muscle_Tissues_Three_Field`.
3. `Solid` which is the main driver of the code. Methods for grid generation, assembling of structures and graphics output (among others) belong to this class.

In addition, the code defines a series of structures to read data (such as material properties and numerical constraints) from a parameters file using the deal.II `ParameterHandler` class. We portray the overall execution of the code in Figure 4.

2.3. Multithreading in the current code. The current implementation uses multithreading in four parts of the code: the assemble of the right-hand side vector and system matrix, the assemble of (locally) statically-condensed matrices, and the update of quadrature points once the new solution has been computed.

The parallel execution found in the code relies on the design pattern *WorkStream* [17]. In modern finite element codes, it is common to find an operation that needs to be done on every cell, followed by a reduction of these local computations into a global structure. Consider, for example, the numerical computation of a very traditional integral in finite element codes for elliptic problems. Indeed, let $\mathcal{T}_h$ be a triangulation of the domain Ω where each cell is denoted by K. Then, we have

$$A_{i,j} = \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j = \sum_{K\in\mathcal{T}_h} \int_K \nabla\varphi_i \cdot \nabla\varphi_j .$$

Here φ_i, φ_j are basis functions of the finite element space and $A_{i,j}$ is an entry of the global matrix. Thus, we can compute the integrals over each K separately (using a quadrature rule, for example) and then reduce all the contributions. In pseudo-code, this general structure can be thought as:

```
GlobalObject  global_object;
```

```
PerTaskData   local_contribtution;
ScratchData   scratch_object;

for each cell in the triangulation{
  local_contribution = compute_local_contribution(cell, scratch_object);
  global_object += local_contribution;
}
```

It is precisely this design pattern that is referred to as *WorkStream*. In deal.II, this is implemented using the function `WorkStream::run` with input arguments that include: the cell and information about how to iterate over them, the functors used to compute the local contribution and to add these to the global object, an object `local_contribution` of type `PerTaskData` where local contributions will be temporarily stored, and an object `scratch_data` of type `ScratchData` where data needed to compute the local contributions is stored. The structures `PerTaskData` and `ScratchData` and their members have to be manually declared before *WorkStream* is called. It is worth mentioning that this pattern "is not intended for distributed memory (DM) use", as explicitly stated in [17]. However, we will see later how we can recycle some of the created structures to convert a SM code into a DM code.

3. Setting up the software for its use with Message Passing Interface.

Message Passing Interface (MPI) is a standardized and portable message-passing standard designed to function on parallel computing architectures (both of SM and DM types). While it is possible to manually link each one of the libraries needed to run MPI codes to an existing installation of deal.II, this is better achieved using the script deal.II/candi [11]. This code provides a batch file which can be executed to install and link all (or some of) the dependencies needed to run MPI deal.II codes, such as mesh generators and partitioners (e.g., p4est [6], METIS [10]) and linear algebra libraries (e.g., PETSc [5], Trilinos [16]).

Because the candi script is no longer maintained for the 8.5 version of deal.II, we had to modify some of their configuration files to fix repository changes, particularly in the HYPRE library [1] needed by PETSc v3.7.6. To do this, we downloaded the 3.7.6 version of PETSc from the original website [5] and updated the file that requests the HYPRE tarball files. Then, we repackaged the software, uploaded it to this author's website[1], updated checksums, and modified the corresponding sources in the configuration files of the candi script. The work was performed on the "dealii-8.5" branch of a forked version of the original repository and it is now located in this author's GitHub repository[2]. The modified candi script run successfully on a virtual machine running Ubuntu 18.04 with 10 GB or RAM and 4 cores.

Remark 3.1. In an earlier version of this work, we had the intention of using the library PETSc to handle the linear algebra aspects of the program. However, we noticed that preconditioners were only implemented in deal.II for use in SM codes (not only in the 8.5 version of the software but also in the latest 9.2 version). Hence, we had to switch to the Trilinos library. The deal.II/candi installer did not require any modifications to link Trilinos and deal.II.

4. A stepping stone: an MPI-version of Step 44.

Due to the complexity of the NML code, we have decided to first convert Step 44 (from which the NML code derives) into a code that can run in a DM environment. Because the structure of Step 44 is exactly the one presented in Figure 4, we can use this code as a template for the modifications that need to be made in the NML muscle code.

4.1. Major changes.

With the code (Step 44) being rewritten, the following changes can be distinguished (unless otherwise stated, the classes that we will refer to belong to the namespace `dealii`):

1. New members have been added to the `Step44::Solid` class. We define new MPI-related variables, such as an MPI communicator (given by the default communicator `MPI_COMM_WORLD`), the number of MPI processes, and a parallel equivalent of `std::cout`. The latter is achieved by defining an object of type `ConditionalOStream` that can be put in place of `std::cout` whenever needed and without any additional considerations.

[1]At the time of writing this report, the modified PETSc package is hosted at: http://www.sfu.ca/~javiera/files/petsc-lite-3.7.6.tar.gz.
[2]https://github.com/javieralmonacid/candi

2. To set the basis for a distributed triangulation, we create objects that describe the degrees of freedom (DOFs) that are locally owned by the current processor and the DOFs that are relevant to the current processor (this includes ghost nodes). We also create objects to describe the partition owned by the local processor and the partition that is relevant to the processor (this includes ghost cells in the vicinity of the locally owned partition). These are all objects of type `IndexSet`.
3. The triangulation is of type `parallel::distributed::Triangulation` (this class is a wrapper for the p4est library [6]). Therefore, the triangulation is neither generated nor stored in whole by any of the processors. They only have access to the cells they own and to some ghost cells.
4. All objects of type `BlockVector` (such as the solution of the system) are converted into `TrilinosWrappers::MPI::BlockVector`.
5. All objects of type `BlockSparseMatrix` (such as the matrix of the system) are converted into `TrilinosWrappers::BlockSparseMatrix`.
6. The initial condition for the dilation is $J_0 = 1$. This means that we have to project the constant function into the finite element space. In Step 44, this is easily achieved using the method `VectorTools::project`, however, this method does not work for distributed elements. We therefore code a new function called `Solid::set_initial_dilation` that solves the L^2-projection problem using Trilinos objects. The variational formulation for this problem reads: find $J \in DGPM_1$ such that

$$\int_\Omega J \cdot H = \int_\Omega 1 \cdot H \quad \forall\, H \in DGPM_1.$$

   Thus, we follow Step 40 (deal.II tutorial for the massively-parallel solution of the Laplace equation) to solve this problem.
7. We combine the methods for assembling the system matrix and right-hand side vectors into a single one called `assemble_system`. This is particularly beneficial to be able to distribute the local contributions to the global ele-

ments using `ConstraintMatrix::distribute_local_to_global` (this function requires both the local matrix and local right-hand side to be present in the same scope, something that was not possible in Step 44 as these two elements were assembled separately).

8. Regarding the static condensation described in subsection 2.1.1, we notice that to compute (for instance) K_{Jp}^{-1}, we have to first manually gather the entries from the global matrix to construct the local version of K_{Jp}. This can become a bottleneck in a distributed code because the elements may not be stored close to each other, and in some cases, they might be stored in different processors. Therefore, instead of performing the static condensation at a local level and then assembling everything (as it is done in Step 44 and the NML code), we perform this process at a global level and we let Trilinos to take care of the inversion of the distributed matrices.
9. We use an algebraic multigrid (AMG) preconditioner to solve the statically-condensed system, instead of the SSOR preconditioner used in Step 44. The AMG preconditioner is used in several distributed codes in the deal.II tutorials, such as Step 40 and Step 55 (distributed solution of a Stokes system).
10. For loops that iterate over all cells of the domain must be re-examined. We do this primarily using the boolean result of `cell->is_locally_owned()` as a flag. In some parts of the code we use the class `FilteredIterators` that provides the beginning and end of an iterator that only considers cells in the current processor. We perform reduction in these loops using two methods: `Utilities::MPI::sum` for simple floating-point numbers and `compress` (available in both the `BlockVector` and `BlockSparseMatrix` classes of the namespace `TrilinosWrappers`) to gather the local contributions to the system matrix and right-hand side vector from other processors.
11. We finally adapt the method `Solid::output_results` to the distributed case. This function is responsible for handling the graphical output to Paraview. Here, each processor writes a `.vtu` file containing the solution and other relevant information (such as residual and partitioning of the mesh). Then,

we let the processor with rank 0 to iterate over each processor collecting the `.pvtu` files containing information about how `.vtu` files relate to each other. This step also writes a master record of extension `.pvd`, which is the file that is called in Paraview for visualization. All these outputs are handled with the methods `write_vtu`, `write_pvtu_record` and `write_pvd_record` from the `dealii::DataOutBase` class.

4.2. Unchanged sections. We were able to recover several pieces of code from the original Step 44. This will reduce the amount of code to be written in the distributed version of the NML code. Here are some of the most noticeable parts that were introduced in the code without modifications.

1. The use of *WorkStream* (as described in subsection 2.3) requires a separate function responsible to compute the local contributions. We reuse these functions and we only take caution when parsing the cell we are working on. For example, in Step 44, the update of the quadrature point history is handled as:

```
WorkStream::run(dof_handler_ref.begin_active(),
                dof_handler_ref.end(),
                *this,
                &Solid::update_qph_one_cell,
                &Solid::copy_local_to_global_UQPH,
                scratch_data_UQPH,
                per_task_data_UQPH);
```

   In turn, we rewrite this call in the new code as:

```
for (const auto &cell : dof_handler_ref.active_cell_iterators())
    if (cell->is_locally_owned())
        update_qph_one_cell(cell, scratch_data_UQPH);
```

   Notice that the function `Solid::copy_local_to_global_UQPH` is empty (in our case), as well as the `per_task_data_UQPH` object. They are not needed because the update of quadrature points (i.e., "moving the mesh") does not need to store additional quantities, but these objects are required to call

`WorkStream::run`. In this case, it is the function `update_qph_one_cell` that can be transferred to the new code (almost) without modifications.

2. The namespace `Parameters` contains several structures that are responsible for parsing parameters from a text file into the program. Because all processors need this information, this class was used in the new code as provided in Step 44.
3. The class `Material_Compressible_Neo_Hook_Three_Field` is responsible for declaring the physical properties of the material and defining the constitutive laws involved in the system (such as that those that describe the stress $\boldsymbol{\sigma}$ in (2.2)). This part of the code is heavily used in the assemble of the matrix and update of the quadrature point history. Because all calls to the member methods are done point-wise (i.e., at each quadrature point), this class can easily be reused in the new code.
4. The function responsible to handle some of the boundary conditions, i.e., `make_constraints`, uses throughout its definition a constraint object of type `ConstraintMatrix`. We found that this class worked well in a DM setting, hence, we reused this function without modifications.
5. Finally, all functions related to error computation were preserved from Step 44. Only minor modifications were required, such as changing the input from shared to distributed (Trilinos) type.

4.3. A numerical example. We run the new implementation for Step 44 in the cluster `plato.usask.ca` using Penguin nodes (2 x Intel Xeon Gold 6148 @ 2.40GHz "Skylake"), each one containing 40 cores. The compilation is done using CMake and the following modules need to be loaded (in this order): `arch/avx2`, `StdEnv/2016.4` and `dealii/8.5.0`.

In this experiment, we recreate the deformation of a nearly-incompressible block under compression [12]. Here, the material is Neo-Hookean with shear modulus $\mu = 80.194 \cdot 10^6\ N/m^2$ and $\nu = 0.4999\ N/m^2$. The domain is a cube of side 1 mm where one of the upper quarters is subject to a load $p_0 = 4\ N/mm^2$ (see Figure 5).

We discretize the domain using 4,096 cells. This results in a system having 140,579

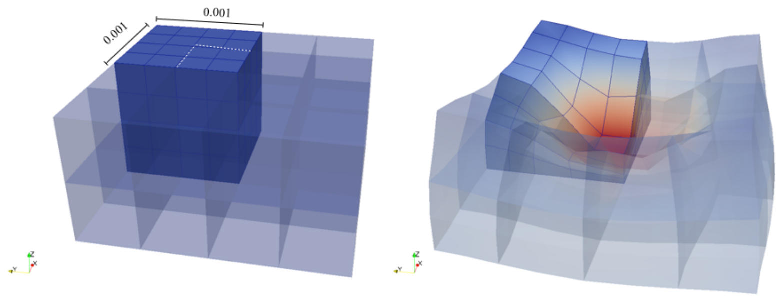

FIG. 5. *Initial and (expected) final configurations for the solid. We only simulate one quarter of the block. Source: [2].*

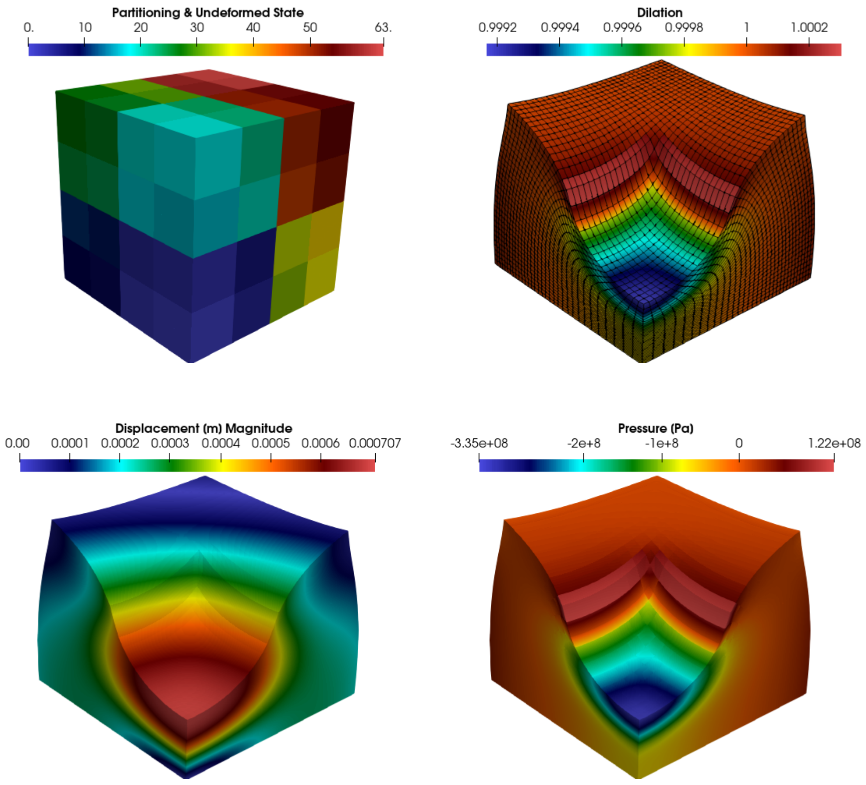

FIG. 6. *Initial and final state of the computed solution using the MPI-version of Step 44 using 64 cores.*

DOFs (107,811 for the displacement and 16,384 for the displacement and dilation). We run the code with a time step of 0.1 s until the end time 0.5 s. To test the parallel capabilities of the new code, we run it in 64 cores, equally distributed in two nodes. We see in Figure 6 that the algorithm provides the expected result. The total computation time was 23 minutes and 4 seconds.

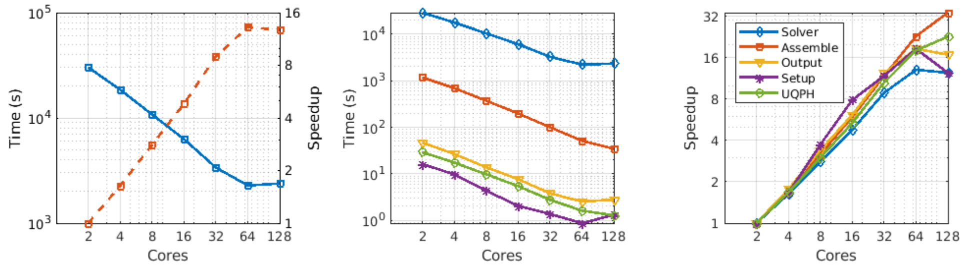

FIG. 7. *Performance and strong scaling of the new DM code. The legend shown in the right graph is the same one used in the center graph. Legend description: "Solver" refers to the linear solver, "Assemble" refers to the assembling of the system matrix and right-hand side, "Output" corresponds to the parallel output of files to* `.vtu` *format, and "UQPH" stands for "update quadrature point history".*

4.4. Scaling analyses for the SM and DM implementations. We submit the newly created code to a scaling analysis using a fixed size mesh containing 32,768 cells. This results in a system containing 1,086,019 DOFs. The experiment is run using 2, 4, 8, 16, 32, 64 (equally distributed in two nodes) and 128 (equally distributed in four nodes) cores.

Remark 4.1. In what follows, all the speedups are computed with respect to the execution time using 2 cores.

We portray the results in Figure 7. Here, we observe linear scaling with some loss of speedup when using 128 cores due to intensive MPI communication (given the size of the problem). Notice that the use of a triangulation of type `parallel::distributed` forces virtually every piece of the code to run in parallel, as we see from the speedup graph in Figure 7.

To put these results in contrast with the existing SM implementation for Step 44, we perform a scaling analysis on this code as well. However, we were not able to run it with the same amount of cells as for the scaling done with the DM code (not a single Newton iteration was computed in under one hour when using 32 cores). Hence, we reduced the size (but still kept it fixed) of the experiment to 4,096 cells (similar to the example presented in subsection 4.3). The results are shown in Figure 8. Here, we see how there is virtually no speedup in the code as the number of cores increase. Indeed, even when the assemble process, the update of quadrature points and assemble of

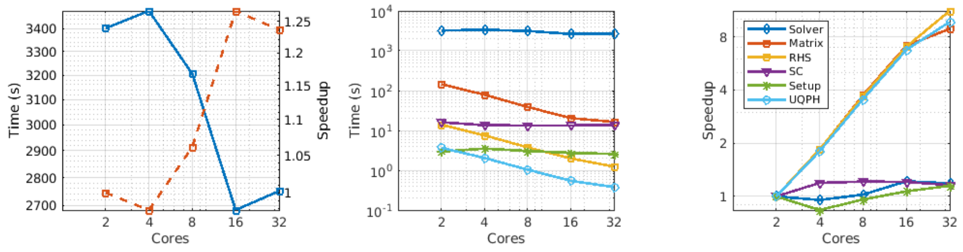

FIG. 8. *Performance and strong scaling of the original SM code (Step 44). The legend shown in the right graph is the same one used in the center graph. Legend description: "Solver" refers to the linear solver, "Matrix" and "RHS" represent the assembling of the system matrix and right-hand side, respectively, "SC" refers to the static condensation, "Setup" refers to the initial setup of the system, and "UQPH" stands for "update quadrature point history".*

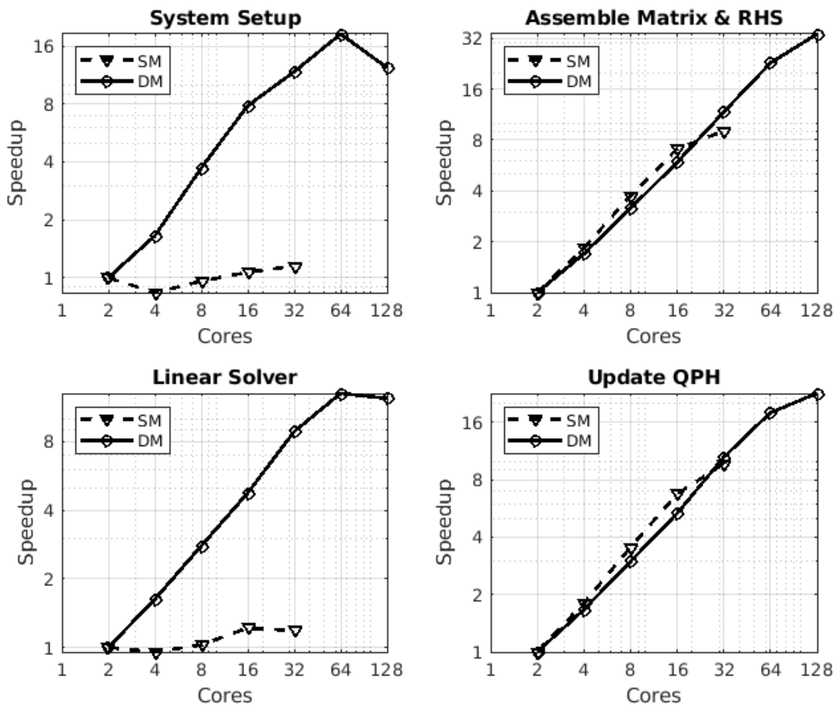

FIG. 9. *Speedup comparison between the SM and DM codes.*

the static condensation are all performed in parallel (see Figure 4), the sequential solution in the linear solver (which is the most time-consuming part of the code according to Figure 7 and Figure 8) diminishes all other parallel components of the code. This comparison becomes more evident when we compare them side-by-side with the distributed code, as we show in Figure 9.

5. The NML code for muscle deformation: basic considerations about its conversion into DM. The NML code works in a very similar way to Step 44, but this time the equivalent to the class `Material_Compressible_Neo_Hook_Three_Field` contains many more members. Nevertheless. the DM version of Step 44 should make the work of converting the NML code into a DM setting much easier. To visualize what can be obtained with the current code, we run a small experiment where we deform a muscle (which could be considered as a simplistic representation of the human medial gastrocnemius) containing a layer of aponeurosis in the top and bottom. The deformation takes place for one second and the mesh contains 315 cells. We show displacement, pressure and dilation fields in Figure 10, Figure 11, and Figure 12, respectively.

In addition to the changes that have been detailed in subsection 4.1, we observe from the NML code that the call to the local methods:

1. `get_active_muscle_fibre_force`,
2. `get_passive_muscle_fibre_force`,
3. `get_basematerial_force`,
4. `get_volume_force`,
5. `get_isochoric_force`,

need be modified in accordance with the distributed triangulation and the result must be reduced using the `Utilities::MPI::sum` function. This is performed during during a post-processing stage where information is being written into XML and binary files. Hence, care must be taken when combining the parallel output of data and the reduction of the quantities being written to disk.

6. Conclusion. In this project, we focused on figuring out which steps need to be taken to convert the existing NML code for skeletal muscle deformation into a distributed code. To do this, we performed this action on the Step 44 deal.II tutorial which deals with the quasi-static deformation of a Neo-Hookean solid. The mesh was generated and partitioned using p4est, while the linear algebra aspects of the code were managed using the Trilinos library. As a byproduct, we compiled a new version of the script deal.II/candi to link these libraries to the deal.II v8.5 main library.

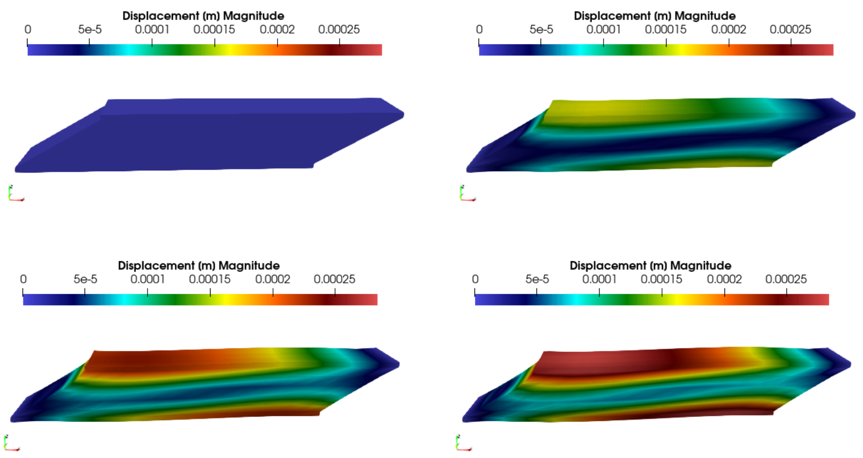

FIG. 10. *Displacement field obtained using the NML code at times (from left to right, from top to bottom): 0.0 s, 0.4 s, 0.7 s and 1.0 s.*

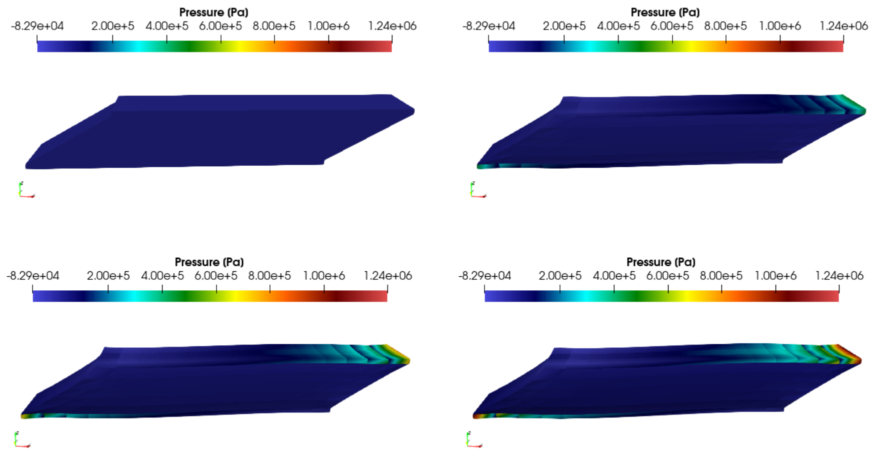

FIG. 11. *Pressure field obtained using the NML code at times (from left to right, from top to bottom): 0.0 s, 0.4 s, 0.7 s and 1.0 s.*

Although Step 44 performs some computations in parallel using WorkStream, the bulk of the work performed by the linear solver is not parallelized. When the number of DOFs increase in the system, this code performs almost in a serial fashion. In turn, the new MPI implementation of Step 44 yields a scalable code that can handle meshes with hundreds of thousands of cells.

Because all parts of the new code run in parallel and the program scales appropriately, we could consider this code a "massively parallel solver for Neo-Hookean solid deformation" (although further testing is required to support this claim). This work

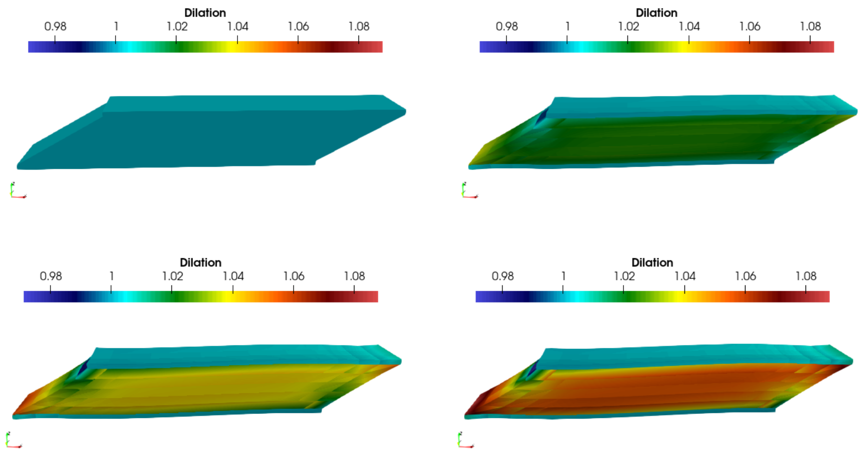

FIG. 12. *Dilation field obtained using the NML code at times (from left to right, from top to bottom): 0.0 s, 0.4 s, 0.7 s and 1.0 s.*

gives great hope in being able to construct a version of the NML muscle code that can handle meshes with a large number of cells (such as those obtained from MRI data) and compute highly-nonlinear solutions in a reasonable amount of time. Future work will precisely continue in this direction.

REFERENCES

[1] *HYPRE: Scalable Linear Solvers and Multigrid Methods.* https://computing.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods. Accessed: 2021-04-28.

[2] *The deal.II Library: Tutorial programs.* https://dealii.org/8.5.0/doxygen/deal.II/Tutorial.html. Accessed: 2021-02-01.

[3] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, and D. Wells, *The deal. ii library, version 8.5*, J. Numer. Math., 25 (2017), pp. 137–145.

[4] E. Azizi, E. L. Brainerd, and T. J. Roberts, *Variable gearing in pennate muscles*, P. Natl. Acad. Sci. USA, 105 (2008), pp. 1745–1750.

[5] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, *PETSc Web page.* https://www.mcs.anl.gov/petsc, 2021, https://www.mcs.anl.gov/petsc.

[6] C. Burstedde, L. C. Wilcox, and O. Ghattas, *p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees*, SIAM Journal on Scientific Computing, 33

(2011), pp. 1103–1133, https://doi.org/10.1137/100791634.

[7] S. L. Delp, F. C. Anderson, A. S. Arnold, P. Loan, A. Habib, C. T. John, E. Guendelman, and D. G. Thelen, *Opensim: open-source software to create and analyze dynamic simulations of movement*, IEEE T. Bio-med. Eng., 54 (2007), pp. 1940–1950.

[8] A. V. Hill, *The heat of shortening and the dynamic constants of muscle*, Proc. R. Soc. Ser. B-Bio., 126 (1938), pp. 136–195.

[9] T. J. Hughes, *The finite element method: linear static and dynamic finite element analysis*, Courier Corporation, 2012.

[10] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM Journal on scientific Computing, 20 (1998), pp. 359–392.

[11] U. Koecher, B. Turcksin, and T. Heister, *candi (compile & install) for deal.II*. https://github.com/dealii/candi. Accessed: 2021-04-01.

[12] S. Reese, P. Wriggers, and B. Reddy, *A new locking-free brick element technique for large deformation problems in elasticity*, Comput. Struct., 75 (2000), pp. 291–304.

[13] S. A. Ross, D. S. Ryan, S. Dominguez, N. Nigam, and J. M. Wakeling, *Size, history-dependent, activation and three-dimensional effects on the work and power produced during cyclic muscle contractions*, Integr. Comp. Biol., 58 (2018), pp. 232–250.

[14] S. A. Ross and J. M. Wakeling, *Muscle shortening velocity depends on tissue inertia and level of activation during submaximal contractions*, Biol. Letters, 12 (2016), p. 20151041.

[15] D. A. Sleboda and T. J. Roberts, *Incompressible fluid plays a mechanical role in the development of passive muscle tension*, Biol. Letters, 13 (2017), p. 20160630.

[16] T. Trilinos Project Team, *The Trilinos Project Website*. https://trilinos.github.io, 2020. Accessed: 2020-05-22.

[17] B. Turcksin, M. Kronbichler, and W. Bangerth, *Workstream–a design pattern for multicore-enabled finite element computations*, ACM T. Math. Software, 43 (2016), pp. 1–29.

[18] J. M. Wakeling, S. A. Ross, D. S. Ryan, B. Bolsterlee, R. Konno, S. Domínguez, and N. Nigam, *The energy of muscle contraction. i. tissue force and deformation during fixed-end contractions*, Front. Physiol., 11 (2020).