Int J CARS (2016) 11:695–705
DOI 10.1007/s11548-015-1241-y

ORIGINAL ARTICLE

Coupling strategies for multi-resolution deformable meshes: expanding the pyramid approach beyond its one-way nature

Matthias Becker[1] · Niels Nijdam[1] · Nadia Magnenat-Thalmann[1]

Received: 12 January 2015 / Accepted: 4 June 2015 / Published online: 20 June 2015

Abstract
Purpose With higher resolutions, medical image processing operations like segmentation take more time to calculate per step. The pyramid technique is a common approach to solving this problem. Starting with a low resolution, a stepwise refinement is applied until the original resolution is reached.
Methods Our work proposes a method for deformable model segmentation that generally utilizes the common pyramid technique with our improvement, to calculate and keep synchronized all mesh resolution levels in parallel. The models are coupled to propagate their changes. It presents coupling techniques and shows approaches for synchronization. The interaction with the models is realized using springs and volcanoes, and it is evaluated for the semantics of the operation to share them across the different levels.
Results The locking overhead has been evaluated for different synchronization techniques with meshes of individual resolutions. The partial update strategy has been found to have the least locking overhead.
Conclusion Running multiple models with individual resolutions in parallel is feasible. The synchronization approach has to be chosen carefully, so that an interactive modification of the segmentation remains possible. The proposed technique is aimed at making medical image segmentation more usable while delivering high performance.

Keywords Deformable models · Multi-resolution · Synchronization · Mesh coupling

✉ Matthias Becker
becker@miralab.ch

[1] MIRALab, CUI, University of Geneva, Battelle, Building A, 7, route de Drize, 1227 Carouge, Switzerland

Motivation

Medical imaging, like computed tomography (CT), magnetic resonance imaging (MRI) and ultrasonography (US), is commonly used in clinical environments and under constant improvement. Higher resolutions result in larger data sets that require specialized processing. A common technique is the pyramid approach [1]. Several levels of details are created by subsampling the input. Using a factor of two, a 2D image would quadruple at each level. Using these levels, processing can be performed from coarse to fine, getting the main structure right before working details. This technique is commonly used in image registration [1]. In this case, the parameters are adjusted for each levels.

To segment anatomical structures, model-based approaches like deformable models [22] have proven to be very powerful and robust. They deform a three-dimensional (3D) polygonal mesh, of the desired anatomical structure, based on internal (shape preservation) and external (image based) forces. This deformation is performed iteratively until an equilibrium has been reached. The calculation effort per step [23] depends on the mesh resolution. Therefore, using the pyramid approach, a low-resolution mesh is used first and configured to allow larger changes. In the refinement (going down the pyramid), the mesh resolution is increased while limiting its range of motion. The structure of the pyramid approach and distribution of steps per level is illustrated in Fig. 1. This way, the coarse model first aligns to the anatomical structure and later the high-resolution model is able to capture fine details [21]. This process is always performed in a one-way direction from coarse to fine.

In our work, we propose a deformable model segmentation approach that works on multiple levels of detail in parallel. The meshes for each layer are synchronized with each other, breaking the linear structure of the pyramid approach. This

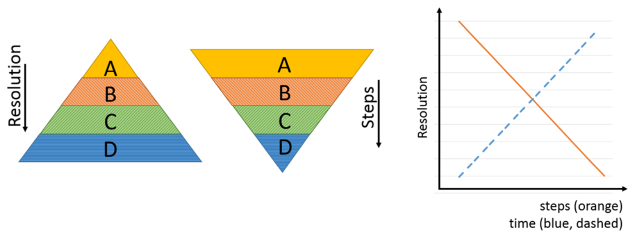

Fig. 1 Using the pyramid approach, the resolution of the model increases with every level (*left*). At the same time, the number of steps per level decreases due to the higher computational costs

changes the interaction pattern. The linear approach requires a scale-based correction—influence points and corrections have to be set (and the mesh evolved) before switching to the next level of detail. In the approach presented here, the user can work location based, e.g., when segmenting the femur, focus on the femoral head first and work on the condyles later. This way, the user can switch between multiple levels of details and can go to a coarser model without losing fine details previously created.

The different meshes run independently from each other (except for some form of synchronization) and can be distributed across multiple cores or machines. We are therefore able to exploit multi-core machines even with algorithms that are not optimized for this. Our methodology can be applied to a large variety of 3D medical images, including CT, MRI and US. The image loading is using the Insight Segmentation and Registration Toolkit (ITK) [34]; therefore, image data sets are required to be ITK compatible. However, image loading can be easily extended, and additional image forces can be introduced for new modalities. Iso-voxel spacing is currently required. Filtering (noise reduction, bias field correction, etc.) can be performed to improve the effectiveness of the image forces.

Related work

Deformable models, introduced by Terzopoulos et al. [31,32] and extended by Cohen [10], are used for a broad field of applications like computer graphics [25], cloth simulation [33], cell tracking in time-lapse microscopy [12] and medical image analysis, as shown in the survey of McInerney et al. [22]. Bredno et al. [7] describe a general deformable model for multiple dimensions. Lehmann et al. [20] give a good overview of different techniques and their fields of application. Deformable models in medical segmentation are discussed in [3].

Deformable models can be divided by their representation. Continuous representations can be explicit, e.g., with snakes [18], or implicit, e.g., with level sets like in [26]. Discrete representations are either based on particle systems, e.g., by Szeliski and Tonnesen [29], or based on either triangular [7] or simplex meshes [11].

Bogovic et al. [5] describe a framework that allows the parallel segmentation of multiple coupled objects. They demonstrate the feasibility of their level set-based approach with brain structures. In contrast, Changizi and Hamarneh [9] show a probabilistic multi-shape presentation. Their approach handles multiple related meshes and can be used to improve statistical models.

Popular interaction modes for snakes, and later deformable models like springs and volcanoes, have been introduced by Kass et al. [18] in 1988. Han et al. [14] have used these interaction modes for multi-user collaborative interactive segmentation. Henriques and Wünsche [15] show improved interaction through meshless deformation.

Edwards et al. [13] discuss mesh consistency during parallel modifications. Lachat et al. [19] present a method for parallel re-meshing of distributed meshes. The work by Sumengen et al. [28] demonstrates a multi-resolution representation of a FEM model that can be deformed in a distributed environments. They subdivide the mesh and every client is simulating a part, and the results are then shared with the other clients. On the client side, it is then possible to generate representations at multiple resolutions. Also Tang et al. [30] demonstrate the distribution of mesh distributions through collaborative environments.

We have chosen deformable models for their flexibility and their well established use in medical image segmentation. Furthermore, interaction methods have been defined for them. Some work has been done on parallel mesh access and distribution and on the generation of multiple resolution representations. Nonetheless, an approach for coupled multi-resolution meshes that allows interaction is still missing.

Methods

The coupling of multiple meshes is a complex process, and several approaches are discussed in detail on their advantages and disadvantages. Furthermore, since the meshes have different resolutions and are continuously processed independently from each other, some form of synchronization is needed in order to propagate the deviations. Lastly, we look at approaches for interactive segmentation and how they can be implemented in the multi-resolution approach.

For a given mesh M, its subversions for n levels are $M_1, M_2, \ldots, M_n$. It is given that always $M_1 \supseteq M_2 \supseteq \cdots \supseteq M_n$. A vertex V may be called V_i within the mesh M_i. This relation is illustrated in Fig. 2 with initial model a and sub-levels b and c.

Coupling

In order to propagate the changes between the coupled meshes $M_1, \ldots, M_n$, we discuss in the following section

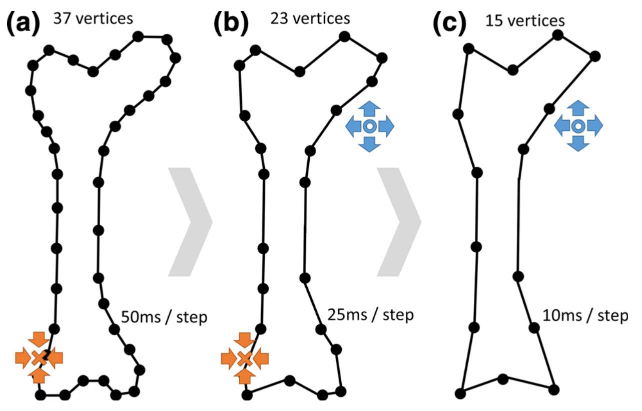

Fig. 2 Concept of the coupled models. An initial model (**a**) is decimated (**b**, **c**) to models with fewer vertices. This reduces the calculation time per simulation step. Interaction is done through magnets and antimagnets. Magnets (x) are passed up to finer levels, and antimagnets (o) are only propagated to more coarse levels

three different techniques: naive direct vertex copying, position averaging and finally spring forces between meshes.

Direct vertex position copying

A simple approach to coupling the meshes is direct vertex copying. It can be done from coarse to fine or from fine to coarse. In either case, either M_1 or M_n are favored, since $M_1 \supseteq \cdots \supseteq M_n$. Therefore, any changes in the meshes $M_2, \ldots M_{n-1}$ are discarded. This renders them useless and reduces the actual number of meshes to two levels: M_1 and M_n.

Vertex position averaging

A more practical approach is to average the vertices across the models. The new average vertex V_{new} is calculated using weights ω_i:

$$V_{\text{new}} = \frac{1}{n} \sum_{i=1}^{n} \omega_i V_i$$

with weights ω_i so that

$$\sum_{i=1}^{n} \omega_i = 1$$

Using this approach, the position of the new vertex is the same for all meshes. Manipulating the weighting factor on a per-mesh basis can be used to increase the impact of certain meshes, e.g., higher resolutions. In extension, it is possible to adjust the weight dynamically per mesh, e.g., starting from a coarse model with a high weight at the beginning, and transfer the weight continuously to the finer models. Another use of this approach is to give the user control over the weights, either directly or through a generalized interface.

In some use cases, it is useful to set the weights depending on the region. An example of this could be the femoral shaft, which has few details of importance so the coarse mesh could be favored. The femoral head on the other hand has a lot of detailed structures of high importance, so the finer meshes would be given more weight.

Spring force coupling

The spring force coupling uses the (weighted) average of the position of the vertices on the different meshes as described above and places a force between the actual vertices and the average position. This Hookean spring force is defined as follows:

$$F_{\text{spring}}(V_j) = \frac{1}{n-1} \sum_{i=1, i \neq j}^{n} \alpha_i (V_j - V_i)$$

Similar to the previous approach, the weights α_i and its control mechanisms can be applied here as well.

Synchronization

Each of the deformation simulations is running in its own thread and at different speeds due to the mesh resolutions. In order to synchronize between the different resolution models, it must be avoided that these meshes are modified while forces are being calculated on their basis. We have developed different strategies (Fig. 3), which we will present here. All strategies are based on a central force (the sync force) function that handles the synchronization. After each simulation step, the models provide a new copy of their changed vertices. The general segmentation process only has to be extended with the call to that function. During the calculation of the forces, the sync force function is called. The execution will be delayed internally until all models are in sync. The actual segmentation thread therefore does not have to be aware of the other threads.

Per step updates strategy

The per step update strategy (PSU), see Fig. 3a, ensures that all threads are always at the same step. This is achieved by blocking the threads at the beginning of the force calculation until all of them have updated their meshes. In this case, the overall performance is restricted by the thread that takes longest time for a segmentation step.

Scaled updates strategy

The scaled update strategy (SU), shown in Fig. 3b, is based on a per-thread cost estimation. While in the PSU strategy, all

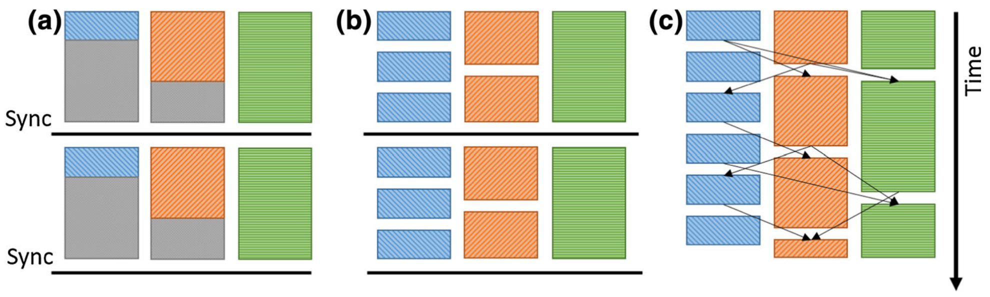

Fig. 3 Comparison of the different synchronization approaches with three models over time. The PSU strategy (**a**) causes a high amount of waiting time (*gray*) for the lower-resolution models for synchronization. The SU strategy (**b**), when carefully tuned, reduces these waiting times. The PU strategy (**c**) performs updates (*arrows*) in a unsynchronized manner, fully removing waiting times

models were synchronized with each other after every step, this strategy has the number of steps per thread depending on the complexity of the model. With a good estimation, waiting times can be reduced in comparison to the PSU strategy. A possible configuration for three threads would be the following: The first thread with a full-resolution model is allowed to do one step, the second thread with a half-resolution model can do two steps, and the third with one third-resolution model can do three steps. When the calculation scales linearly with resolution, that configuration reduces waiting times. In addition, the cost estimation can be dynamically derived and adjusted from the resolution of the model and the time it takes for the simulation to perform a single step.

Partial updates strategy

Figure 3c illustrates the partial updates strategy (PU), which strongly reduces the synchronization overhead with its relaxed locking mechanism. For each resolution model, updates are stored immediately and forces are calculated directly upon request. Locking is only used to ensure that no forces are being calculated while the meshes are changing. This approach should allow the best performance but depending on the thread scheduling models can run massively apart. However, due to the synchronization, this effect is kept minimal.

Interactive segmentation

There exist many styles of interaction for deforming meshes. After careful consideration, we have selected to implement springs and volcanoes, also known as magnets and antimagnets. Magnets are designed to attract the mesh, and antimagnets are supposed to push away the mesh.

A magnet at the position P only interacts with the closest vertex V_i of the mesh. Using the scaling factor α_{magnet}, they are defined as:

$$F_{\text{magnet}}(V_i) = \alpha_{\text{magnet}}(P - V)$$

Antimagnets are designed to affect only near proximity, given by a radius. A vertex V inside this radius, having the distance r to the antimagnet, receives the following force, scaled by $\alpha_{\text{antimagnet}}$:

$$F_{\text{antimagnet}}(V) = \alpha_{\text{antimagnet}} \frac{r}{\|r\|^3}$$

Magnets and antimagnets are set/defined by the user on a specific levels. Nonetheless, they should also influence the meshes at other resolutions. To reduce side effects and unwanted behavior, we have evaluated the semantics of the operations and defined the following rules:

1. Magnets are only passed down to more detailed levels. Magnets pull the closest vertices toward them. While this is desirable at all levels, on more coarse levels, they will attach to a different vertex and can cause major, unwanted corrections.
2. Opposite to this, antimagnets are only passed up to more coarse levels. When using antimagnets, the user usually wants to correct a segmentation mistake and to push the model away from certain structures. This is a larger operation that works more efficiently on coarse mesh. Nevertheless, the more detailed models also benefit from this, since they are coupled to the coarse models.

Implementation

In this section, we will describe the segmentation framework architecture and its implementation. A general overview is given for segmentation and the architectural changes for the implementation of concurrency. Figure 4 shows the structure of both frameworks.

Our segmentation framework is based on existing frameworks and structured into layers. We use VTK [2] and ITK [34] and OpenMesh [6] and the Bullet physics library[1]

[1] http://bulletphysics.org/.

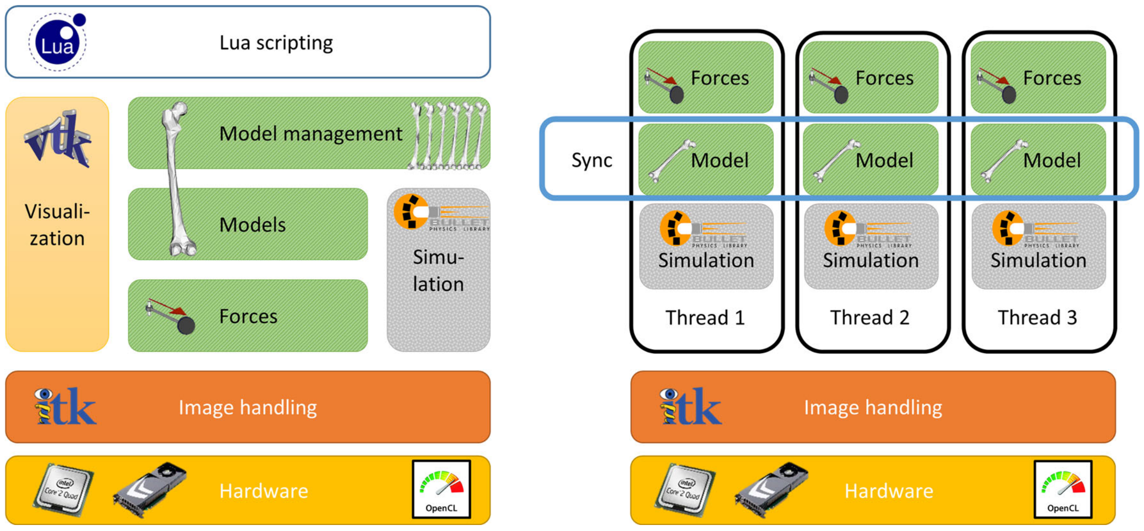

Fig. 4 Deformable model framework used in this work has a layered architecture. Its single-process version (*left*) has several components (scripting, visualization, model manager) which have been omitted in the presentation of the multi-threaded version (*right side*)

and is scriptable by using Lua [16]. We have selected to be capable of using the underlying hardware at its best. The physics simulation library Bullet physics has be chosen since its pipeline also is available as OpenCL[2] version which can exploit not only multi-core CPUs but also many-core GPUs. Similarly, ITK has been chosen not only because the large number of supported file formats and filters but also because of its OpenCL support.

While the core of the framework is developed in C and C++, we provide Lua bindings for fast iterations and easy testing of new concepts. The main loop of the segmentation process is shown below. Before that code, the images and models have been loaded and the simulation has been initialized. In the loop, we first calculate the forces that are to be applied to the model, and we integrate them using the simulation environment. Finally, the visualization is updated.

Listing 1 Main loop for deformable model calculation

```
for i=1,1000 do
  if i% 60 == 0 then
    print((i/60), "seconds␣simulated")
  end
  modelMan:CalcForces()
  sim:performSimulationStep(0.016666)
  visu:addModel(model,1.0,0.0,0.0)
end
```

Image data sets can largely vary in size and can become very big, especially in multi-channel images. Therefore, we have designed the multi-threaded version of our framework to share the image between the threads. Since all image operations are read-only, we do not need advanced locking or access control. So the image manager forms the shared foundation for the threads as shown in Fig. 4. Each thread receives a different model and therefore needs its own model manager. To avoid interference, every model also uses its own simulation environment. Each thread, that is created and started, is assigned a unique ID accessible from the per-thread Lua scripting environment.

Synchronization

The task of running multiple meshes in parallel and coupling their vertices requires careful synchronization. It falls into two parts, the actual synchronization using primitives like mutexes and the integration into the scripts.

We have modified the main loop of the deformable model simulation to include calls to our force that performs the coupling (modelSyncForce). This force is implemented as a singleton and is therefore accessible from all threads. Every thread initially has to register to this force, as seen on this first line of the source below. Threads register with their ID, their model and the index mapping they use. The different resolutions of the meshes are created by decimation of the initial high-resolution mesh. All meshes share some vertices. To identify them, all vertices are mapped to the indices of the initial model. Since the modelSyncForce is calculated across threads, it gets called apart from the model manager. It has the threadID and the model as parameters. After simulation step is done, the new vertices model are updated in the modelSyncForce. Depending on the synchronization strategy, execution can be interrupted in either the calculation or the update phase.

[2] https://www.khronos.org/opencl/.

Listing 2 Modified main loop for multi-threaded deformable model calculation.

```
modelSyncForce:registerThread
 (threadID,model,indices)
for i=1,1000 do
  if i% 60*2 == 0 then
    print(threadID, " : ",(i/60)*2,
          " seconds simulated")
  end
  modelMan:CalcForces()
  modelSyncForce:calcForces
       (threadID,femur)
  sim:performSimulationStep(0.016666)
  modelSyncForce:updateThread
       (threadID,femur)
  visu:addModel(femur,threadID)
end
```

We have carefully selected the components of our single-threaded version framework to support different platforms, including Microsoft Windows, Apple OS X and Linux. For the multi-threaded version, we have selected boost's threads and synchronization primitives because it allowed us to keep the cross-platform compatibility. In the next section, we will describe how we have used threads mutexes to implement the three synchronization strategies described in "Synchronization" section under "Methods".

Partial update strategy (PU)

The partial update strategy provides locking for concurrent access to the vertices, but it does not perform any synchronization. If threads are stalled, it is possible that the other threads continue to run and scatter. The basic locking of access to the vertices is shown in the following listing. The access to the vertices is also controlled in that fashion in the other strategies.

Listing 3 Usage of mutexes for locking in the partial update strategy.

```
mutex vertexLock;

void update(threadID, model) {
 vertexLock.lock();
 //copy vertices here
 vertexLock.unlock();
}

void calculate(threadID, model) {
 vertexLock.lock();
 //calculate force here
 vertexLock.unlock();
}
```

Per step update strategy (PSU)

The per step strategy requires synchronization at two points: The forces can only be calculated once all models have been updated and the updates can only be done once all forces have been calculated.

Listing 4 Usage of mutexes for locking in the per step update strategy.

```
mutex canUpdate;
mutex canCalculate;
std::map < Key,bool> threadUpdated;
std::map < Key,bool> threadCalculated;

void update(threadID, model) {
 while (!canUpdate.try_lock()) {
   this_thread->yield();
 }
 canUpdate.unlock();
 threadCalculated[threadID] = false;
 //copy vertices here
 threadUpdated[threadID] = true;
 if (all_updated) {
   canUpdate.lock();
   canCalculate.unlock();
 }
}

void calculate(threadID, model) {
 while (!canCalculate.try_lock()) {
   this_thread->yield();
 }
 canCalculate.unlock();
 threadUpdated[threadID] = false;
 //calculate force here
 threadCalculated[threadID] = true;
 if (all_calculated) {
   canCalculate.lock();
   canUpdate.unlock();
 }
}
```

Scaled updates strategy (SU)

The scaled updates strategies resemble the per step update strategy, but some threads are entitled to more steps before sync (between sync it follows the partial update strategy). It is important that the last update is not called, before all threads have reached it. This ensures that at least the original model still refers to its initial state. In our implementation, we use a mutex per thread to block the (last) update until all threads have performed the required number of iterations.

Listing 5 Usage of mutexes for locking in the scaled update strategy.

```
std::map < Key,int> scalingFactor;
std::map < Key,int> threadCalculated;
std::map < Key,mutex> threadUpdated;

void updateThread(threadID, model) {
 if ((threadCalculated[threadID] >=
   (scaling_factor[threadID]))) {
   while (!updateLocks[threadID]
     ->try_lock()) {
     this_thread->yield();
   }
 }
 //copy vertices here
}

void calcForces(Key threadID) {
 //calculate force here
 threadCalculated[threadID]++;
 if (all_calculated) {
   //reset update counts
   for (Key k: threadCalculated) {
     threadCalculated[k] = 0;
     updateLocks[k]->unlock();
   }
 }
}
```

Results

We have tested our approach in different scenarios. We describe the data set and medical field used for evaluation and evaluate the effect of the synchronization overhead.

Data sets

For the evaluation data set, we have chosen image data that covers the complete leg. It has been acquired in the context of the MultiScaleHuman project [24]. The image has a dimension of 654 × 655 × 604 voxel, with a spacing of 0.78, 0.78 and 1.5 mm. Details of the device used and the mDixon protocol that was used can be found in [17]. The MR scans were stitched and filtered, and an initial labeling by tissue types has been performed. The preprocessing steps are described in [4], and they include stitching of separate stacks, a bias field correction using a non-uniform intensity normalization (N3) approach by Sled et al. [27] and the reduction of noise using a non-local means algorithm as described by Buades et al. [8]. The data set is kept at its initial size, and all processes operate on the same copy.

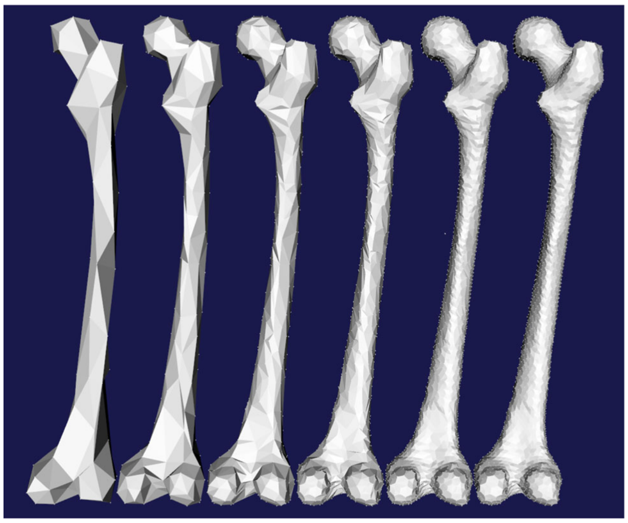

Fig. 5 Effect of the decimation on the model of the femur. The original model (*right*) is reduced to fewer vertices while preserving shape features like the femoral head and the condyles

The goal is to segment the femur in this data set. We have evaluated three femur models, from low to high resolution. The low-resolution model had 344, the medium-resolution model had 1850, and the high-resolution model had 37,010 vertices, and these models are shown in Fig. 5. Having these large variations in size allows us to estimate the share of the time used for locking as well as the overall robustness.

Deformable model configuration

We have used our deformable framework as described in “Implementation” section. It has been configured with several forces. We use Laplacian smoothing using adjacent vertices within a one-edge neighborhood and a constant growing force. We exploit the image data by using gradients. For the coupling force, we have chosen the spring force, since it best fits our requirements.

Starting with the loaded (high-, medium- and low-resolution) model, we have created reduced versions of them. With n models, model M_i, $i = 1, \ldots, n$, only has $\frac{1}{i}$ times the vertices of the initial model. We have used the built-in decimators of OpenMesh, in this case with the collapse priority based on error quadrics. The decimation at different levels of a femur model is shown in Fig. 5.

Synchronization evaluation

The synchronization is the bottleneck of the whole calculation. The independent segmentation processes have to be coordinated to all meet, after a specific strategy, as described in “Synchronization” section under “Implementation”. In this section, we evaluate the three approaches to seeing the

Table 1 Timing measurement of the low-resolution model for total duration of update, the time waiting for locks during the update, the duration of the calculation of the force and the time waiting for updates during force calculation

Sync	Thr.	Update (µs)	Lock µs	Calc	Lock
PSU	1	5	0	72 µs	1 µs
PSU	2	65	62	10 ms	9.8 ms
PSU	3	1287	1279	10.7 ms	10.5 ms
PSU	4	2405	2391	10.9 ms	10.7 ms
PSU	5	2421	2410	11 ms	10.8 ms
PSU	6	2140	2129	11.1 ms	10.9 ms
PU	1	5	1	71 µs	1 µs
PU	2	4	3	56 µs	1 µs
PU	3	7	3	62 µs	2 µs
PU	4	9	9	73 µs	9 µs
PU	5	6	4	66 µs	7 µs
PU	6	9	5	53 µs	9 µs
SU	1	15	7	70 µs	0 µs
SU	2	2918	2813	66 µs	3 µs
SU	3	3646	3633	69 µs	4 µs
SU	4	2655	2644	58 µs	3 µs
SU	5	1739	1730	59 µs	4 µs
SU	6	1701	1694	56 µs	3 µs

Table 2 Timing measurement of the medium-resolution model for total duration of update, the time waiting for locks during the update, the duration of the calculation of the force and the time waiting for updates during force calculation

Sync	Thr.	Update	Lock	Calc	Lock
PSU	1	76.9 ms	76.9 ms	906 µs	440 µs
PSU	2	79.9 ms	79.9 ms	387 ms	386 ms
PSU	3	79.7 ms	79.7 ms	454 ms	453 ms
PSU	4	79.8 ms	79.8 ms	474 ms	473 ms
PSU	5	79.9 ms	79.9 ms	484 ms	483 ms
PSU	6	79.9 ms	79.9 ms	489 ms	488 ms
PU	1	0 µs	0 µs	459 µs	0 µs
PU	2	13 µs	0 µs	395 µs	26 µs
PU	3	12 µs	6 µs	375 µs	12 µs
PU	4	15 µs	15 µs	349 µs	0 µs
PU	5	16 µs	3 µs	245 µs	3 µs
PU	6	9 µs	6 µs	270 µs	3 µs
SU	1	141 ms	141 ms	500 µs	0 µs
SU	2	5.2 ms	5.2 ms	334 µs	15 µs
SU	3	25.3 ms	25.3 ms	324 µs	9 µs
SU	4	32.1 ms	32.1 ms	351 µs	44 µs
SU	5	30.2 ms	30.2 ms	268 µs	5 µs
SU	6	27.6 ms	27.6 ms	280 µs	20 µs

impact of locking. We have used the three models with different resolutions and have run our framework with six threads per model. During the segmentation, we have measured the average times for updating the mesh and calculating the forces. Furthermore, we have measured the time that was spent waiting for locks in these parts. The results are listed below for the low-resolution (Table 1), medium-resolution (Table 2) and high-resolution (Table 3) models. A visual representation of the results can be found in Fig. 6 (PSU), Fig. 7 (PU) and Fig. 8 (SU).

The first result we found is that the partial update strategy always is the fastest. In the calculation time, it can be seen that the time linearly grows with model size (as indicated by the different threads). For the per step update strategy, the first thread has the shortest lock time. This can be explained by the fact that it requires the longest calculation and therefore the other threads already have reached the synchronization point. With growing model size, the differences between the strategies become more visible. While in the low-resolution model, the difference between PU and SU and PSU is in the area of ms, it grows at medium resolution. At high resolution, PU takes 7–11 ms, PSU between 30 and 115 ms and SU between 29 and 70 s. The locking time per thread increases with the resolution for PSU strategy. It can be seen that PU causes the least time waiting for locks.

Table 3 Timing measurement of the high-resolution model for total duration of update, the time waiting for locks during the update, the duration of the calculation of the force and the time waiting for updates during force calculation

Sync	Thr.	Update	Lock	Calc	Lock
PSU	1	26.8 ms	25.4 ms	8216 µs	0 µs
PSU	2	69.5 ms	69.1 ms	6470 µs	23 µs
PSU	3	75.7 ms	75.2 ms	6849 µs	382 µs
PSU	4	89.3 ms	88.8 ms	7922 µs	424 µs
PSU	5	98.8 ms	98.3 ms	7714 µs	517 µs
PSU	6	107.9 ms	107.5 ms	6494 µs	487 µs
PU	1	2255 µs	1754 µs	9020 µs	0 µs
PU	2	167 µs	0 µs	8686 µs	1503 µs
PU	3	167 µs	0 µs	7851 µs	1837 µs
PU	4	167 µs	0 µs	6681 µs	0 µs
PU	5	963 µs	869 µs	6734 µs	818 µs
PU	6	987 µs	889 µs	6376 µs	310 µs
SU	1	70.1 s	70.1 s	10.1 ms	1.1 ms
SU	2	70.0 ms	70.0 s	8.2 ms	1.0 ms
SU	3	19.4 s	19.4 s	6.1 ms	60 µs
SU	4	42.4 s	42.4 s	7.2 ms	1.2 ms
SU	5	24.1 s	25.1 s	6.1 ms	189 µs
SU	6	28.5 s	28.5 s	6.8 ms	1.1 ms

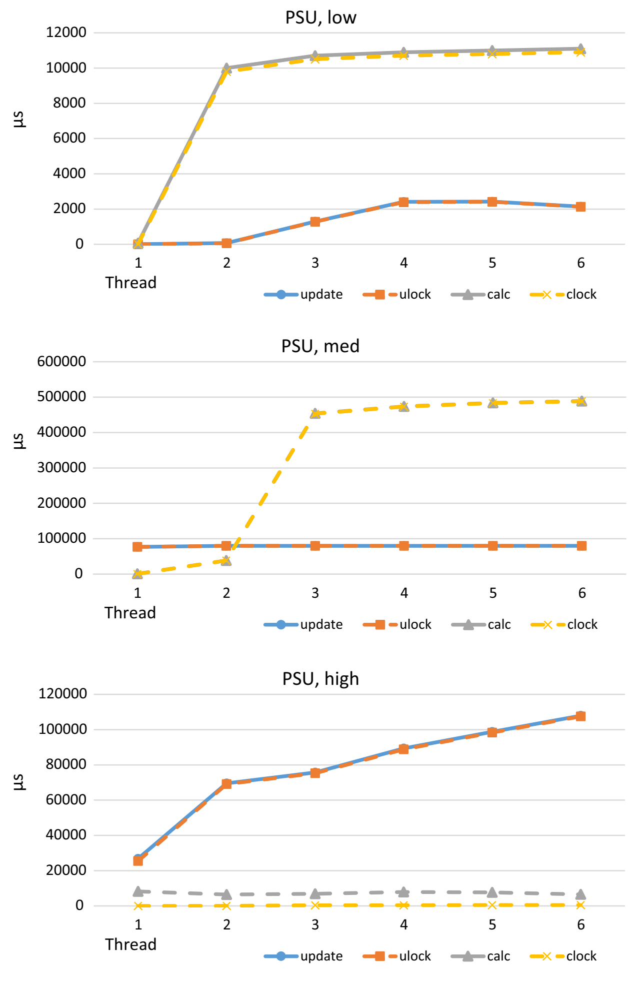

Fig. 6 Timing measurement of the per step update approach for total duration of update, the time waiting for locks during the update, the duration of the calculation of the force and the time waiting for updates during force calculation. The locking times are *dashed*

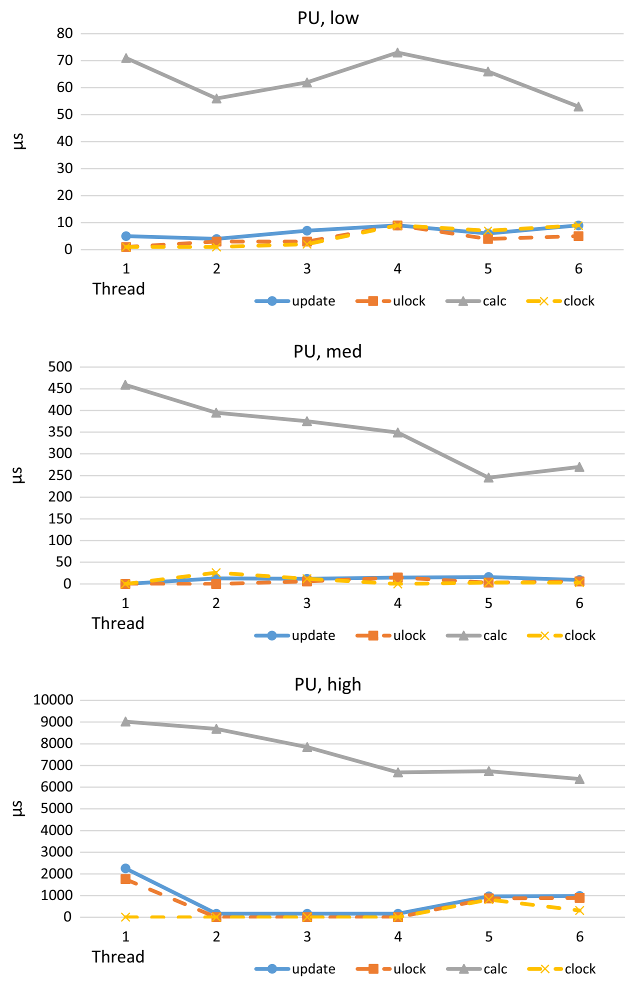

Fig. 7 Timing measurement of the partial update approach for total duration of update, the time waiting for locks during the update, the duration of the calculation of the force and the time waiting for updates during force calculation. The locking times are *dashed*

Discussion

Total calculation duration is sensitive to model resolution. This can be seen from the timing data independent of the synchronization strategy. The overall computation time (ignoring the locks) goes down with the resolution as expected.

By looking closely at these strategies, we found the partial update strategy to be the fastest. This is achieved by reducing the synchronization alignment dependencies between the threads. The only locking is around updating and force calculation for memory safety, but there are no global synchronization approaches. This is supported by the measured data that show that the PU strategy has the shortest lock times. For the lower-resolution threads, we can see a higher share of locking time. Since these threads are simulated faster, they reach the locks more often. Furthermore, the higher-resolution threads take longer for updating and force calculation and therefore block the locks longer.

The PSU strategy shows that the thread with the highest resolution spends the shortest time waiting for locks. Since this thread, by pure resolution, always takes the longest time to simulate, all other threads are waiting already when the locking point is reached. In the high resolution, it can be seen that the total force calculation times no longer consists of locking. This propagates to the update locks and causes to be stalled for more time than in the low-resolution models.

The SU strategy is very sensitive to proper selection of the scaling. We can see the expected timing results, all in the same region, for low- and medium-resolution model, but not for the high-resolution model. The large variation in values (19–70 s) indicates that the scaling factors are not chosen well.

Overall, we found the PU strategy to be the most beneficial. It enables a fast coupling with only very little synchronization overhead. The behavior follows user expec-

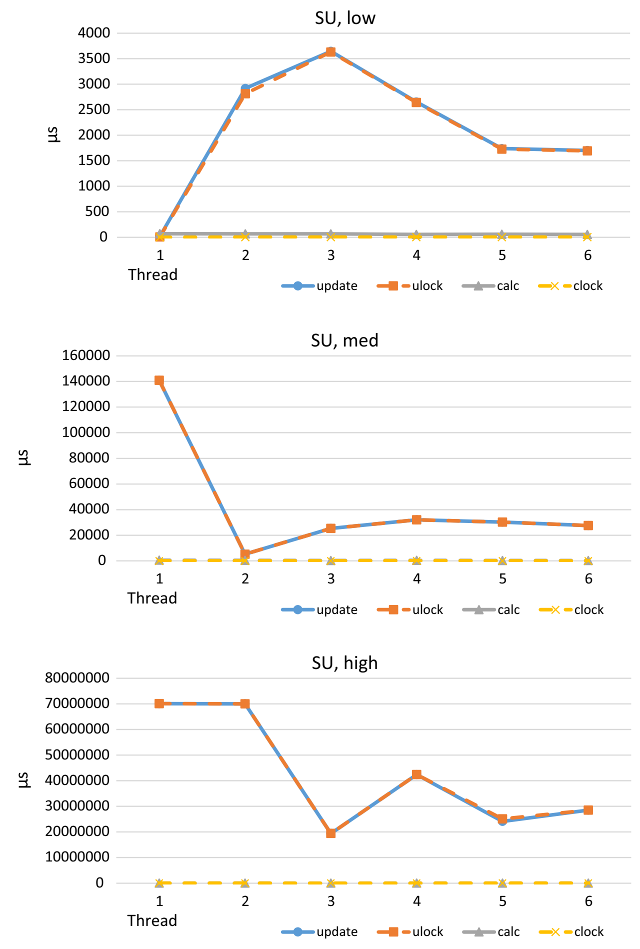

Fig. 8 Timing measurement of the scaled update approach for total duration of update, the time waiting for locks during the update, the duration of the calculation of the force and the time waiting for updates during force calculation. The locking times are *dashed*

tation and does not block the user's interaction with model through long locking times.

Limitations of the study and outlook

For now, we have only tested our proposed method on a single machine. An interesting outlook would to extend it to a multi-machine approach. Possible scenarios would be off-site versus on-site: A coarse simulation running locally allows smooth interaction, whereas a more powerful remote server runs the fine simulation on the same data. Similarly, the coarse simulation could be run on a phone or tablet computer while the fine model is simulated in the hospital data center or the cloud.

With model synchronization already in place, another outlook would be the collaborative interaction with the model. A distributed group of user could work in parallel on the model, if desired at different scales, and could collaboratively place interaction points. That way, additional knowledge could be included for critical parts of the segmentation process, e.g., in image region with artifacts or unclear model boundaries.

Conclusion

In this work, we have presented a framework to perform multi-resolution mesh coupling for medical image segmentation. We have discussed approaches for the coupling through vertex propagation, averaging and springs. We have investigated the synchronization of the parallel models and introduced three locking approaches. On the implementation side, we have shown how a single-model framework can be transformed into a multi-model framework. To better understand the impact of locking, we have evaluated the different models with different resolution with all locking strategies. The results show the partial update strategy as a clear winner. For the future, calculation across multiple machines and collaborative interactions seem promising. To better understand the impact on users, a comprehensive user study should follow.

Acknowledgments This work has been funded by the EU FP7 Marie Curie Initial Training Network project Multi-scale Biological Modalities for Physiological Human Articulation (MultiScaleHuman) under Grant Number 289897.

Conflict of interest Matthias Becker, Niels Nijdam and Nadia Magnenat-Thalmann declare that they have no conflict of interest.

Informed consent For the image data that have been used in this work, informed consent was obtained from all patients. The Ethical Committee for Research On Humans (CEREH) of the Geneva University Hospitals and the Swiss Agency for Therapeutic Products granted their approval for this study.

References

1. Adelson E, Anderson C (1984) Pyramid methods in image processing. RCA Eng 29(6):33–41
2. Avila LS, Barré S, Blue R, Cole D, Geveci B, Hoffman WA, King B, Law CC, Martin KM, Schroeder WJ, Squillacote AH (2001) The VTK user's guide. Kitware Inc, New York
3. Becker M, Magnenat-Thalmann N (2014) Deformable models in medical image segmentation. In: Magnenat-Thalmann N, Ratib O, Choi HF (eds) 3D multiscale physiological human, chap. 4, 1st edn. Springer, London, pp 81–106. doi:10.1007/978-1-4471-6275-9_4
4. Becker M, Magnenat-Thalmann N (2014) Muscle tissue labeling of human lower extremities in multi-channel mDixon MR imaging: concepts and applications. In: Bioinformatics and biomedicine (BIBM), 2014 IEEE international conference on, pp 279–284. doi:10.1109/BIBM.2014.6999168
5. Bogovic JA, Prince JL, Bazin PL (2013) A multiple object geometric deformable model for image segmentation. Comput Vis Image Underst 117(2):145–157. doi:10.1016/j.cviu.2012.10.006
6. Botsch M, Steinberg S, Bischoff S, Kobbelt L (2002) OpenMesh–a generic and efficient polygon mesh data structure. In: OpenSG symposium, Citeseer

7. Bredno J, Lehmann TMT, Spitzer K (2003) A general discrete contour model in two, three, and four dimensions for topology-adaptive multichannel segmentation. IEEE Trans Pattern Anal Mach Intell 25(5):550–563
8. Buades A, Coll B, Morel J (2005) A non-local algorithm for image denoising. In: Computer vision and pattern recognition. CVPR 2005. IEEE computer society conference on, vol 2, pp 60–65. doi:10.1109/CVPR.2005.38
9. Changizi N, Hamarneh G (2010) Probabilistic multi-shape representation using an isometric log-ratio mapping. In: Medical image computing and computer-assisted intervention: MICCAI … International conference on medical image computing and computer-assisted intervention, vol 13, Pt 3, pp 563–570
10. Cohen LDL (1991) On active contour models and balloons. CVGIP Image Underst 53(2):211–218
11. Delingette H (1999) General object reconstruction based on simplex meshes. Int J Comput Vision 32(2):111–146. doi:10.1023/A:1008157432188
12. Dufour A, Thibeaux R, Labruyère E, Guillén N, Olivo-Marin JC (2011) 3-D active meshes: fast discrete deformable models for cell tracking in 3-D time-lapse microscopy. IEEE Trans Image Process 20(7):1925–1937. doi:10.1109/TIP.2010.2099125
13. Edwards HC, Williams AB, Sjaardema GD, Baur DG, Cochran WK (2010) SIERRA toolkit computational mesh conceptual model. Tech. Rep, March, Sandia National Laboratories
14. Han S, Nijdam NA, Schmid J, Kim J, Magnenat-Thalmann N (2010) Collaborative telemedicine for interactive multiuser segmentation of volumetric medical images. Vis Comput 26(6–8):639–648. doi:10.1007/s00371-010-0445-y
15. Henriques A, Wünsche B (2008) Improved meshless deformation techniques for plausible interactive soft object simulations. In: Computer vision and computer graphics. Theory and applications
16. Ierusalimschy R, de Figueiredo LH, Filho WC, de Figueiredo LH (1996) Lua–an extensible extension language. Softw Pract Exp 26(6):635–652
17. Juan DG, Trombella S, Delattre BM, Seimbille Y, Ratib O (2014) Study of skeletal muscle behavior by PET/MRI. In: IEEE-EMBS international conference on biomedical and health informatics (BHI), pp 61–64. doi:10.1109/BHI.2014.6864304
18. Kass M, Witkin A, Terzopoulos D (1988) Snakes: active contour models. Int J Comput Vis 331(4):321–331
19. Lachat C, Dobrzynski C, Pellegrini F (2014) Parallel mesh adaptation using parallel graph partitioning. In: Proceedings of the 11th World Congress on computational mechanics, pp 1–12
20. Lehmann TM, Bredno J, Spitzer K (2003) On the design of active contours for medical image segmentation. A scheme for classification and construction. Methods Inf Med 42(1):89–98. doi:10.1267/METH03010089
21. Lötjönen J, Reissman PJ, Magnin IE, Katila T (1999) Model extraction from magnetic resonance volume data using the deformable pyramid. Med Image Anal 3(4):387–406
22. McInerney T, Terzopoulos D (1996) Deformable models in medical image analysis: a survey. In: Mathematical methods in biomedical image analysis, 1996, Proceedings of the workshop on, vol 1, pp 171–180. Elsevier. doi:10.1109/MMBIA.1996.534069
23. Meier U, López O, Monserrat C, Juan MC, Alcañiz M (2005) Real-time deformable models for surgery simulation: a survey. Comput Methods Programs Biomed 77(3):183–197
24. MultiScaleHuman project
25. Nealen A, Müller M, Keiser R, Boxerman E, Carlson M (2006) Physically based deformable models in computer graphics. Comput Graph Forum 25(4):809–836. doi:10.1111/j.1467-8659.2006.01000.x
26. Osher S, Sethian JA (1988) Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. J Comput Phys 79(1):12–49. doi:10.1016/0021-9991(88)90002-2
27. Sled JG, Zijdenbos AP, Evans AC (1998) A nonparametric method for automatic correction of intensity nonuniformity in MRI data. IEEE Trans Med Imaging 17(1):87–97. doi:10.1109/42.668698
28. Sumengen S, Eren MT, Yesilyurt S, Balcisoy S (2008) A multi-resolution mesh representation for deformable objects in collaborative virtual environments. In: Communications in computer and information science, vol 21 CCIS, pp 75–87
29. Szeliski R, Tonnesen D (1992) Surface modeling with oriented particle systems. ACM SIGGRAPH Comput Graph 26(2):185–194. doi:10.1145/142920.134037
30. Tang Z, Rong G, Guo X, Prabhakaran B (2010) Streaming 3D shape deformations in collaborative virtual environment. In: 2010 IEEE virtual reality conference (VR) pp 183–186. doi:10.1109/VR.2010.5444793
31. Terzopoulos D, Platt J, Barr A, Fleischer K (1987) Elastically deformable models. SIGGRAPH. Comput Graph 21(4):205–214. doi:10.1145/37402.37427
32. Terzopoulos D, Witkin A (1988) Physically based models with rigid and deformable components. IEEE Comput Graph Appl 8:41–51. doi:10.1109/38.20317
33. Volino P, Magnenat-Thalmann N (2005) Implicit midpoint integration and adaptive damping for efficient cloth simulation. Comput Animat Virtual Worlds 16:163–175
34. Yoo TS, Ackerman MJ, Lorensen WE, Schroeder W, Chalana V, Aylward S, Metaxas D, Whitaker R (2002) Engineering and algorithm design for an image processing API: a technical report on ITK–the insight toolkit. Stud Health Technol Inform 85:586–592