Parallelized Topological Relaxation Algorithm

Guangchen Ruan
Indiana University
Bloomington, USA
gruan@iu.edu

Hui Zhang
University of Louisville
Louisville, USA
hui.zhang@louisville.edu

***Abstract*—Geometric problems of interest to mathematical visualization applications involve changing structures, such as the moves that transform one knot into an equivalent knot. In this paper, we describe mathematical entities (curves and surfaces) as link-node graphs, and make use of energy-driven relaxation algorithms to optimize their geometric shapes by moving knots and surfaces to their simplified equivalence. Furthermore, we design and configure parallel functional units in the relaxation algorithms to accelerate the computation these mathematical deformations require. Results show that we can achieve significant performance optimization via the proposed threading model and level of parallelization.**

***Index Terms*—mathematical visualization, topological relaxation, parallel computing, performance tuning, python**

I. Introduction

The first central idea of this paper is to model self-deformable mathematical objects in 3- and 4-dimensional Euclidean Space. Our objects of fundamental interest are mathematical curves (1D objects embedded in 3-space) and mathematical surfaces (2D objects embedded in 4-space). In topology, both mathematical curves and surfaces are changing structures — they can deform into their equivalence which can appear very different but indeed represent the same embedding in mathematical space.

A. Relaxing 1D Mathematical Knots and 2D Surfaces

The idea of knot or surface equivalence is to give a precise definition of when two objects should be considered the same even when positioned quite differently in space. A formal mathematical definition is that two objects are equivalent if there is an orientation-preserving homeomorphism. For example, topologically, sphere and cube are one and the same object since one can be transformed continuously (i.e. with no cutting nor tearing) into another. In this section we briefly describe the basic algorithms for topologically relaxing 1D mathematical knots and 2D surfaces with a simulation where moves to change their structures are proposed and generated during each iteration.

In mathematics, knots are closed loops (they do not have ends) like a circle (or ring). In fact, a circle is a knot, known as an unknot or a trivial knot because it is so simple. The

This work was partly funded by NSF grants #1651581 and #1726532. Hui Zhang is corresponding author of this work.

978-1-7281-0858-2/19/$31.00 ©2019 IEEE

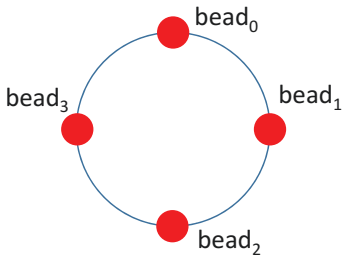

Fig. 1. Illustration of 1D mathematical knot.

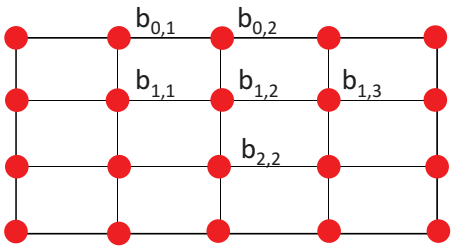

Fig. 2. Illustration of 2D mathematical surface.

creation of mathematical 3D curves and knots (closed 3D curves) can often be facilitated with a 2D drawing interface. One often constructs an initial configuration for an object while neglecting most issues of geometric placement. The next task that comes naturally is to topologically refine these initial embeddings, not only to make the geometry look more pleasant, but also to simplify the geometric representation. Ideally most of the work can be done automatically. Our approach is largely based on force-directed algorithms, also known as spring embedders, that calculate the layout of a graph (in our case, 1D linked nodes for knots as in Fig. 1, and 2D linked nodes for surfaces as in Fig. 2) using only information contained within the structure itself, without relying on domain-specific guidance (see e.g., the 1984 algorithm of Eades [1]).

Force Laws for Topological Refinement. To embed a graph we replace the vertices with electro-statically charged masses and replace each edge with a spring to form a me-

chanical system. The vertices are placed in some initial layout and let go so that the spring systems and electrical forces on the masses move the system to a minimal energy state. Two basic forces are used, an ***attractive mechanical*** force applied between adjacent masses on the same spring and a ***repulsive electrical*** force applied between all other pairs of masses. The mechanical force is a generalization of Hooke's law, allowing for an arbitrary power of the distance r between masses, $F_m = Hr^{1+\beta}$, where H is a constant. The electrical force also allows for a general power of the distance, $F_e = Kr^{-(2+\alpha)}$, where r again is the distance between the two masses, and K is a constant. The electrical force is applied to all pairs of masses excluding those consisting of adjacent masses on the same link. In most of the preliminary results [2]–[4] shown in this proposal, we used $\beta = 1$ and $\alpha = 2$.

Collision Avoidance for Topology Preservation. For this force-directed algorithm to be applicable to our principal test case of mathematical curves positioned in $\mathbb{R}^3$, it is imperative that any proposed evolution should respect topological constrains: it does not involve cutting the curve or passing the curve through itself. Parallel to the force laws previously specified, the self-intersection problem is solved in our approach by requiring that the position of each mass be updated one at a time, and collision avoidance is strictly performed to determine if one is heading towards one of the following two potential collisions:

- **point-segment collision** — a vertex of a 3D curve is going towards a link of the curve and the distance is less than a predefined threshold distance,
- **segment-segment collision** — a link of a 3D curve is going towards another link and their distance is less than a predefined threshold distance.

If either of these two states exists, the pair of closest points on the colliding components are identified to define a 3D vector passing through them. An equal (but opposite) displacement along the 3D vector is then applied to each component to take the component out of collision range. The collision avoidance process modifies masses' positions whenever necessary to ensure the entire evolution is under topological constraint [5], [6]. Fig. 3 shows the dynamic model works to prevent unwanted intersections that might change the closed curve's topological features.

B. Related Work

The idea of making computer-generated mathematical pictures of curves and surfaces has developed in many directions with the recent advances in computational algorithm and mathematical visualizaiton. Carter generates nicely rendered figures for many most complicated yet beautiful examples in modern topology, such as the 2-manifolds embedded and evolved in 4-dimensional space [7]. Scharein's *Knotplot* has been widely used to construct and deform mathematical knots physically in 3D [8]. Weeks' SnapPea software displays and manipulates the over/under crossings of mathematical knots [9]. Some combine haptic interfaces and 3D graphics to simulate the dynamics of 3D curves and 4D knotted surfaces (see, e.g, Phillips [10], Spillmann [11], and Zhang [12], [13]). Zhang's work concerns Reidemeister-move based interfaces to deform mathematical 3D knots and 4D surfaces with mathematically valid moves, i.e., the Reidemeister moves [14], [15]. While such "expert's interfaces" have provided many interesting 4D visualization results, the entire interaction process can be much tedious and error prone.

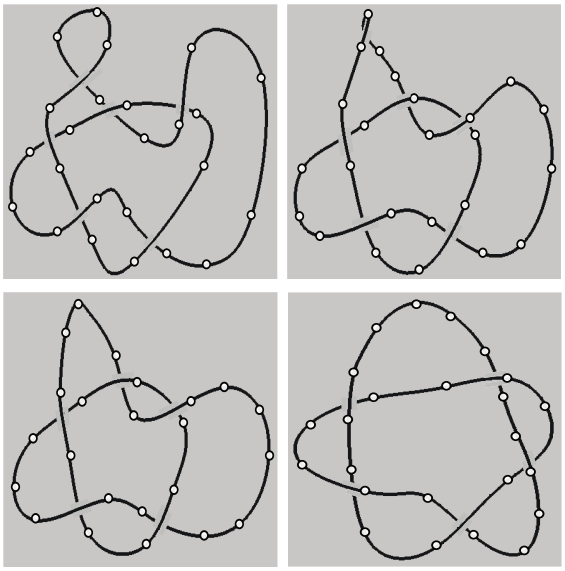

Fig. 3. Typical screen images of the self-deformation. The simple closed curve (a knot 5^1) relaxes, with the proposed force laws and collision avoidance mechanism. During the relaxation, the knotted string preserves the its topological structure.

In many application domains, parallel algorithms are developed to speed up computation of their serial counterparts. In [16], Ruan *et al.* transform points in density based clustering to graphs and leverage distributed iterative graph processing framework to parallelize the clustering. In [17], Ruan *et al.* propose parallel algorithms to mining quantitative timing information from event based temporal data. The algorithms leverage iterative MapReduce [18] framework for parallelization. To enable quantitative and qualitative measurement of caries lesion, in [19], Ruan *et al.* propose a parallel, MapReduce [18] based framework where 2D contours are extracted from dental CT scans in Map phase and 3D geometry of tooth is reconstructed in Reduce phase.

II. Parallel Functional Unit Algorithms for One-dimensional Mathematical Knot

As we can see from Section I-A, each iteration of the simulation process consists of two steps: (1) calculating forces for each bead, followed by (2) performing collision avoidance where positions of involved beads are adjusted whenever the collision is detected.

First of all, we have an important observation that the force calculation procedure of one bead is independent of that of another, which makes the force calculation of beads can be effectively parallelized with multi-threading technique.

Furthermore, we note that the force calculation procedure can be broken down into following functional units for each bead. (1) calculating repulsive forces, (2) calculating attractive forces, (3) calculating the total force by summing up received repulsive and attractive forces, and (4) Moving the bead into a new position based on the total force and the maximum magnitude. This fine-grained breakdown opens the door of of designing a parallel algorithm with particular level of parallelization (*i.e.*, # of threads) for each functional unit in terms of its specific time complexity analysis and therefore offering better performance compared to handling the force calculation as a whole.

Below we detail the parallel algorithm design for each functional unit along with time complexity analysis using big O notation.

A. Parallel Algorithm for Repulsive Force Calculation

Time Complexity Analysis. For each bead, it receives repulsive forces from all other beads. In addition, we note that repulsive force is symmetric, meaning that the forces that bead A receives from B and B receives from A have the same magnitude with opposite direction. Therefore, the time complexity for calculating repulsive forces is $O(N*(N-1)/2) = O(N^2)$ where N is the number of beads in the knot in question.

Algorithm 1: Parallel Repulsive Force Calculation - 1D Knot

Input : $Beads$: set of beads, $NumWorkers$: number of worker threads
Output: $distMatrix$: distance matrix, $repulsiveForceMatrix$: repulsive force matrix

```
1  #Initialization
2  distMatrix ← zeros();
3  repulsiveForceMatrix ← zeros();
4  #Get cell indices in form of row and
     column numbers of the lower left
     triangle of distMatrix
5  indexList ← cell indices of the lower left triangle of
     distMatrix;
6  #Generate list of thread ids, ranging
     from 0 to NumWorkers − 1
7  threadIdList ← range (NumWorkers);
8  threadList ← [ ];
9  foreach threadId ∈ threadIdList do
10   task ← repulsiveForceTask (Beads,
       NumWorkers, threadId, indexList,
       distMatrix, repulsiveForceMatrix);
11   thread ← newThread();
12   thread.run(task);
13   threadList.add(thread);
14 #Wait until all threads finish
     execution
15 threadList.join()
```

Algorithm 1 outlines the logic of parallelizing the calculation of repulsive forces, with blue colored lines as comments. The algorithm input are $Beads$ which is a N by 3 matrix where each row represents a bead in the form of a three-dimensional vector, and $NumWorkers$ which specifies the # of threads to speed up the computation. The output are calculated distance matrix N by N $distMatrix$ as well as N by N by 3 $repulsiveForceMatrix$. We note that due to symmetry for both matrices only a half triangle needs to be calculated.

As preparation, we initialize $distMatrix$ and $repulsiveForceMatrix$ (lines 3 and 4) and generate list of cell indices (*i.e.*, each entry in the list is a cell's index in the matrix in the form of row and column number pair) of the lower left triangle of the distance matrix (line 5). The cell indices are used to indicate the workload of the calculation. Moreover, we generate a list of thread ids ranging from 0 to $NumWorkers - 1$, which are used by worker threads to determine individual workload. Furthermore, we initialize an empty thread list to hold threads to run (line 8). In the *for* loop, for each worker thread, we compose task (line 10), launch thread and invoke the task (lines 11 and 12), and add the thread to the list (line 13). Finally, we wait until all worker threads complete theirs tasks before exit (line 15).

The function *repulsiveForceTask()* used to compose worker thread task is sketched in Algorithm 2. We can see that we pass $distMatrix$ and $repulsiveForceMatrix$ by reference so the two states are shared across all worker threads. To dispatch workload evenly amongst workers, we employ a round robin based approach. To fulfill this, we firstly get indices of the entries in $indexList$, where each entry is a pair of row and column numbers identifying the position of a cell in the distance matrix. Then in the *for* loop that iterates over the indices, we simply check whether $i \bmod NumWorkers$ equals to the assigned thread id. If the answer is yes, then the worker proceeds to handle the entry, and skip it otherwise. In this way, each worker handles exactly the same amount of workload with at most one entry difference. In the meantime, this workload dispatching strategy effectively avoids race conditional and threads can update the shared states (*i.e.*, $distMatrix$ and $repulsiveForceMatrix$) concurrently without requiring locking mechanism.

B. Parallel Algorithm for Attractive Force Calculation

Time Complexity Analysis. For each bead, it receives attractive forces from two adjacent beads. Similar to repulsive force, calculation of attractive force is symmetric as well. Therefore, the time complexity for calculating attractive forces is $O(N * 2/2) = O(N)$ where N is the number of beads in the knot in question. Furthermore, we are able to reuse the distance matrix derived from the previous step of calculating repulsive forces.

Algorithm 3 highlights the algorithm used to parallelize the calculation of attractive forces. The algorithm input are original input $Beads$ that represents the knot data and $NumWorkers$ which specifies the # of threads to speed

Algorithm 2: Repulsive Force Task - 1D Knot

Input : $Beads$: set of beads, $NumWorkers$: number of worker threads, $threadId$: worker thread Id, $indexList$: cell indices, $distMatrix$: distance matrix, $repulsiveForceMatrix$: repulsive force matrix

Output: None

```
1  #Get indices of the entries in indexList
2  entryIdxList ← range (length (indexList));
3  foreach i ∈ entryIdxList do
4      #Use threadId to determine whether the
        cell in question is its
        responsibility
5      if i mod NumWorkers ! = threadId then
6          continue;
7      #Row index of bead_i
8      indexI ← indexList[i].row;
9      #Row index of bead_j
10     indexJ ← indexList[i].col;
11     #Get vectors for bead_i and bead_j
12     bead_i ← beads[indexI, :]; bead_j ← beads[indexJ, :];
13     #Calculate L2 distance between bead_i
        and bead_j
14     v ← bead_i − bead_j;
15     distMatrix[indexI, indexJ] ← L2 (v);
16     #Calculate repulsive force that bead_j
        imposes on bead_i
17     magnitude ← K ∗ dist^−(2+α);
18     repulsiveForceMatrix[indexI, indexJ, :] ←
        v/dist ∗ magnitude;
```

up the computation. In addition, $distMatrix$ is the distance matrix which is the output from prior repulsive force calculation. The output of the algorithm is a N by 3 $attractiveForceMatrix$. Likewise, because of symmetry only a half triangle needs to be calculated for the attractive force matrix.

The algorithm starts by initializing $attractiveForceMatrix$, $threadIdList$ and $threadList$ (lines 1 to 5), followed by the *for* loop which starts workers for actual processing. In each iteration of the loop, it composes a task (line 7) and launches a worker thread which in turn runs the task (lines 8 and 9), and bookkeeps the thread by appending it to the thread list (line 10). Once all workers are launched, we wait until their completion before exiting the main thread (line 12).

Algorithm 4 gives the pseudocode of the task that is assigned to a worker. Here we employ a similar round-robin based strategy to dispatch workload evenly across all worker threads as the one described in Algorithm 2. If the $bead_i$ in question is on the worker's duty list, it finds $bead_j$ that is next

Algorithm 3: Parallel Attractive Force Calculation - 1D Knot

Input : $Beads$: set of beads, $NumWorkers$: number of worker threads, $distMatrix$: distance matrix

Output: $attractiveForceMatrix$: attractive force matrix

```
1  #Initialization
2  attractiveForceMatrix ← zeros();
3  #Generate list of thread ids, ranging
    from 0 to NumWorkers − 1
4  threadIdList ← range (NumWorkers);
5  threadList ← [ ];
6  foreach threadId ∈ threadIdList do
7      task ← attractiveForceTask (Beads,
        NumWorkers, threadId, distMatrix,
        attractiveForceMatrix);
8      thread ← newThread();
9      thread.run(task);
10     threadList.add(thread);
11 #Wait until all threads finish
    execution
12 threadList.join()
```

to $bead_i$ and calculates the attractive force that $bead_j$ imposes on $bead_i$. By using modulo operation to determine the row index of $bead_j$ we handle the special case when $bead_i$ is the last row in the input matrix $beads$ (line 10). We note that due to symmetry we only need to do one way calculation, *i.e.*, the attractive force that $bead_j$ receives from $bead_i$ can be inferred from the calculated one (*i.e.*, force that $bead_j$ imposes on $bead_i$), by just negating the direction of the force. Therefore we effectively cut the computation workload by half.

Moreover, recall that we can reuse the distance matrix calculated from repulsive force calculation (see Algorithms 1 and 2) and only lower left triangle of the distance matrix has been filled because of symmetry. In lines 14 to 17, we get the distance between $bead_i$ and $bead_j$ by positioning the indices (*i.e.*, $indexI$ and $indexJ$) carefully so the accessed cell falls in the lower left triangle of the distance matrix. Finally, we calculate the attractive force based on retrieved distance (lines 16 to 18).

C. Parallel Algorithm for Total Force Calculation

Time Complexity Analysis. For each bead, to get its total force we sum up repulsive forces from all beads except the two most adjacent ones and two attractive forces from the two adjacent ones. Therefore, the time complexity for calculating total forces is $O(N * (N - 1 - 2 + 2) = O(N^2)$ where N is the number of beads in the knot in question.

Algorithm 5 sketches the algorithm that parallelizes the calculation of total forces. Its input are dataset $Beads$, number of worker threads to use $NumWorkers$, as well as repulsive force matrix $repulsiveForceMatrix$ and attractive force matrix $attractiveForceMatrix$ that are from prior calculation

Algorithm 4: Attractive Force Task - 1D Knot

Input : $Beads$: set of beads, $NumWorkers$: number of worker threads, $threadId$: worker thread Id, $distMatrix$: distance matrix, $attractiveForceMatrix$: attractive force matrix

Output: None

```
1  #Get number of beads in the dataset
2  numBeads ← length (Beads);
3  foreach i ∈ range (numBeads) do
4      #Use threadId to determine whether the
        bead in question is its
        responsibility
5      if i mod NumWorkers != threadId then
6          continue;
7      #Row index of bead_i
8      indexI ← i;
9      #Row index of bead_j next to bead_i
10     indexJ ← (i + 1) mod numBeads;
11     #Get vectors for bead_i and bead_j
12     bead_i ← beads[indexI, :]; bead_j ← beads[indexJ, :];
13     #Get L2 distance between bead_i and
        bead_j from distance matrix
14     if indexI < indexJ then
15         dist ← distMatrix[indexJ, indexI];
16     else
17         dist ← distMatrix[indexI, indexJ];
18     #Calculate attractive force that
        bead_j imposes on bead_i
19     v ← bead_i − bead_j;
20     magnitude ← H * dist^(1+β);
21     attractiveForceMatrix[indexI, :] ←
        v/dist * magnitude;
```

Algorithm 5: Parallel Total Force Calculation - 1D Knot

Input : $Beads$: set of beads, $NumWorkers$: number of worker threads, $repulsiveForceMatrix$: repulsive force matrix, $attractiveForceMatrix$: attractive force matrix

Output: $totalForceMatrix$: total force matrix

```
1  #Initialization
2  totalForceMatrix ← zeros();
3  #Generate list of thread ids, ranging
    from 0 to NumWorkers − 1
4  threadIdList ← range (NumWorkers);
5  threadList ← [ ];
6  foreach threadId ∈ threadIdList do
7      task ← totalForceTask (Beads,
        NumWorkers, threadId,
        repulsiveForceMatrix, attractiveForceMatrix,
        totalForceMatrix);
8      thread ← newThread();
9      thread.run(task);
10     threadList.add(thread);
11 #Wait until all threads finish
    execution
12 threadList.join()
```

(see Algorithms 1 and 3). The output of the algorithm is a N by 3 matrix $totalForceMatrix$, which stores the resulting total forces.

The algorithm initializes $totalForceMatrix$, $threadIdList$ and $threadList$ (lines 1 to 5), and in the following *for* loop it launches workers with assigned tasks. The logic in the loop covers: (1) composing a task (line 7); (2) launching a worker thread to run the task (lines 8 and 9); and (3) bookkeeps the thread for later join (line 10). Once all workers are launched, we wait until their completion before exiting the main thread (line 12).

The payload function $totalForceTask()$ is outlined in Algorithm 6. Again, we employ a round-robin based dispatching strategy. In line 8, we sum up attractive forces from two most adjacent beads. Recall that when calculating attractive forces we leverage the symmetry property therefore the attractive force $bead_i$ receives from $bead_{i-1}$ should be the negative one that $bead_{i-1}$ obtains from $bead_i$ (hence the minus operator rather and add). To handle the special case of $bead_i$ being the first in the dataset, we uses the trick of $(i - 1 + numBeads)\ mod\ numBeads$ to get the index of $bead_{i-1}$. In lines 7 to 20, we adopt a two-step approach by first summing up repulsive forces from all other beads (lines 7 to 13) and then subtracting the ones from the two adjacent neighbors (lines 14 - 20). By this means, we have a performance boost by eliminating the need of checking whether a bead is $bead_i$'s adjacent neighbor and needs be skipped consequently. Once again, we use modulo operation to get $bead_i$'s two adjacent neighbors correctly when $bead_i$ is either the first or the last in the input dataset (line 15). In addition, for $bead_x$ that enforces repulsive force on $bead_i$, we check the relative order of indices of $bead_x$ and $bead_i$ (*i.e.*, x and i) to make sure that the accessed cell lies in the lower left triangle of the distance matrix and use negation as needed (lines 10 to 13 and 16 to 20).

D. Parallel Algorithm for New Position Calculation

Time Complexity Analysis. For each bead, we adjust its position based on the total force it received. Therefore, the time complexity for calculating new position is simply $O(N)$ where N is the number of beads in the knot in question.

Algorithm 7 outlines the algorithm used to prarallelize the calculation of new position of beads. Note that there is no output since input $beads$ is passed by reference and the adjusted positions are updated upon it directly. The worker

Algorithm 6: Total Force Task - 1D Knot

Input : $NumWorkers$: number of worker threads, $threadId$: worker thread Id, $repulsiveForceMatrix$: repulsive force matrix, $attractiveForceMatrix$: attractive force matrix

Output: None

```
1  #Get number of beads in the dataset
2  numBeads ← length (repulsiveForceMatrix);
3  foreach i ∈ range (numBeads) do
4      #Use threadId to determine whether the bead in question is its responsibility
5      if i mod NumWorkers != threadId then
6          continue;
7      #Sum up attractive forces from two most adjacent beads
8      v = attractiveForceMatrix[i, :] − attractiveForceMatrix[(i − 1 + numBeads) mod numBeads, :];
9      #Sum up repulsive forces from all beads
10     foreach j ∈ range (i) do
11         v += repulsiveForceMatrix[i, j, :];
12     foreach j ∈ range (i + 1, numBeads) do
13         v −= repulsiveForceMatrix[j, i, :];
14     #Subtract repulsive forces from two adjacent neighbors
15     idxList ← [(i + 1) mod numBeads, (i − 1 + numBeads) mod numBeads];
16     foreach idx ∈ idxList do
17         if i < idx then
18             v += repulsiveForceMatrix[idx, i, :];
19         else
20             v −= repulsiveForceMatrix[i, idx, :];
21     totalForceMatrix[i, :] ← v;
```

thread launching and task assignment are similar to prior algorithms and hence we skip the details. Note however that we need to first find the maximum magnitude of all forces (line 2), which serves a the normalizer in the position adjustment process. This can be achieved by either a straightforward linear scan of $totalForceMatrix$ or a parallel approach based on divide-and-conquer technique. Since the operation (*i.e.*, magnitude comparison) is lightweight, a linear scan should suffice and yield better performance.

Algorithm 8 sketches the position adjustment task. In line 8, we come up with a new position for the bead in question based on the total force it received, the maximum magnitude of all forces as well as a user configured move threshold parameter.

Algorithm 7: Parallel New Position Calculation - 1D Knot

Input : $Beads$: set of beads, $NumWorkers$: number of worker threads, $totalForceMatrix$: total force matrix

Output: None

```
1  #Find the maximum magnitude of all total forces
2  maxMagnitude ← findMaxMagnitude (totalForceMatrix);
3  #Generate list of thread ids, ranging from 0 to NumWorkers − 1
4  threadIdList ← range (NumWorkers);
5  threadList ← [ ];
6  foreach threadId ∈ threadIdList do
7      task ← adjustPositionTask (Beads, NumWorkers, threadId, totalForceMatrix, maxMagnitude);
8      thread ← newThread();
9      thread.run(task);
10     threadList.add(thread);
11 #Wait until all threads finish execution
12 threadList.join()
```

Algorithm 8: Adjust Position Task - 1D Knot

Input : $Beads$: set of beads, $NumWorkers$: number of worker threads, $threadId$: worker thread Id, $totalForceMatrix$: total force matrix, $maxMagnitude$: maximum magnitude

Output: None

```
1  #Get number of beads in the dataset
2  numBeads ← length (Beads);
3  foreach i ∈ range (numBeads) do
4      #Use threadId to determine whether the bead in question is its responsibility
5      if i mod NumWorkers != threadId then
6          continue;
7      #Adjust position
8      Beads[i, :] ← Beads[i, :] + totalForceMatrix[i, :]/maxMagnitud ∗ moveThreshold;
```

III. Parallel Functional Unit Algorithms for Two-dimensional Mathematical Surface

The simulation of 2D mathematical surface shares a lot of similar characteristics with 1D knot in terms of computation. For instance, the force calculation can be broken down into the calculation of repulsive force, attractive force, total force and position move as well. In addition, force is symmetry. Moreover, one important observation is that 2D surface employs the same logical representation as 1D knot after linear transformation, based on which many parallel algorithms presented in Sec. III shall apply for 2D surface. In this section we describe the rationale of the transformation and for parallel algorithms we highlight the pieces that need special attention for 2D surface scenario.

A. 2D Surface to 1D Linear Representation

In 1D knot scenario, the beads on knot can be viewed as a 1D array where each element in the array represents a bead in the form a 3D point. In 2D surface scenario, we view the surface as a 2D matrix where each element is again a bead of the form 3D point. To be able to reuse the frameworks of the parallel algorithms discussed in Sec. III, it is motivated to transform 2D surface's matrix into 1D knot's array so logically they use the same representation. The way to transform a cell in 2D matrix with row index row_{idx} and column index col_{idx} is shown in Eq. 1, where $NumCols$ is the # of columns in the matrix.

$$linear\_index = col_{idx} + row_{idx} * NumCols \quad (1)$$

The reverse transformation from a linear index to original row, column indices pair is also straightforward, as shown in Eq. 2 (where // means integer division) and Eq. 3.

$$row_{idx} = linear\_index \;//\; NumCols \quad (2)$$

$$col_{idx} = linear\_index \bmod NumCols \quad (3)$$

B. Parallel Algorithm for Repulsive Force Calculation

Time Complexity Analysis. For each bead, it receives repulsive forces from all other beads. Hence the analysis for 1D knot case applies here, *i.e.*, $O(N^2)$ where $N = NumCols * NumRows$ is the number of beads in the 2D matrix in question.

With linear transformation, Algorithm 1 and Algorithm 2 shall directly work for the 2D surface scenario and we skip the repetition. We simply note that the input $Beads$ stores the data of transformed 2D matrix, with linear index (*i.e.*, row # of $Beads$) based on Eq. 1 ranging from 0 to $NumCols * NumRows - 1$.

C. Parallel Algorithm for Attractive Force Calculation

Time Complexity Analysis. For each bead, it receives attractive forces from four adjacent beads, up, down, right and left and the force is symmetry. Therefore, the time complexity

Algorithm 9: Attractive Force Task - 2D Surface

Input : $Beads$: set of beads, $NumWorkers$: number of worker threads, $threadId$: worker thread Id, $numRows$: # rows of 2D matrix, $numCols$: # columns of 2D matrix, $distMatrix$: distance matrix, $attractiveForceMatrix$: attractive force matrix

Output: None

```
1  numBeads ← length (Beads);
2  foreach i ∈ range (numBeads) do
3      if i mod NumWorkers != threadId then
4          continue;
5      #Transform from linear index to
        (row, col) index
6      rowIdx ← i // numCols;
7      colIdx ← i mod numCols;
8      #Add right hand side neighbor
9      neighborIdx ← [ ];
10     colIndex ← colIdx + 1;
11     if colIndex < numCols then
12         neighborIdx ← append ((rowIdx, colIndex));
13     else
14         neighborIdx ← append (None);
15     #Add lower side neighbor
16     rowIndex ← rowIdx + 1;
17     if rowIndex < numRows then
18         neighborIdx ← append ((rowIndex, colIdx));
19     else
20         neighborIdx ← append (None);
21     #Get vectors for bead_i and bead_j
22     indexI ← i;  bead_i ← beads[indexI, :];
23     cnt ← 0;
24     foreach t ∈ neighborIdx do
25         if t is None then
26             cnt ← cnt + 1; continue;
27         #Get linear index of bead_j
28         indexJ ← t[0] * numCols + t[1];
29         bead_j ← beads[indexJ, :];
30         v ← bead_j − bead_i;
31         dist ← distMatrix[indexJ, indexI]
32         magnitude ← H * dist^(1+β);
33         attractiveForceMatrix[indexI, cnt, :] ←
            v/dist * magnitude;
34         cnt ← cnt + 1
```

for calculating attractive forces is $O(N*4/2) = O(N)$ where $N = NumCols * NumRows - 1$.

Algorithm 3 can be directly applied here. The only difference is that the output $attractiveForceMatrix$ is N by 2 by 3 rather than N by 3. In this matrix, each row_i stores the attractive forces from its right (in first column) and lower (in second column) neighbors. The forces from its left and upper neighbors can be inferred in terms of symmetry of the force. Next we focus on Algorithm 9 which is the composed task for worker thread. Compared to Algorithm 4, to get $bead_i$'s neighbors, we need to first transform the linear index to its original (row, column) indices pair (line 5 to 7). Because of symmetry property, we only need indices of right hand side and lower side neighbors for force calculation, and consider special case when $bead_i$ sits on edge row/column (lines 9 to 20). In the *for* loop, we transform pair index to linear index to access $bead_j$ (lines 27 to 29), retrieve distance from previously calculated distance matrix (line 31) and perform the force calculation (lines 32 to 33).

D. Parallel Algorithm for Total Force Calculation

Time Complexity Analysis. For 2D surface, each bead receives repulsive forces from all beads other than its four most adjacent neighbors, which only enforce attractive forces. Therefore, the time complexity analysis of total force calculation is $O(N * (N - 1 - 4 + 4) = O(N^2)$.

We can reuse Algorithm 5 as the parallelization framework and hence only need focus on the worker thread task, as shown in Algorithm 10. For $bead_i$'s total force, first we sum up attractive forces from its four neighbors (lines 5 to 13). Note that forces from left and upper neighbors can be inferred with opposite direction because of symmetry (lines 7 to 13). Next we add all repulsive forces (lines 14 to 18). Finally we subtract the forces from its four neighbors to get the total force.

E. Parallel Algorithm for New Position Calculation

Thanks to linear transformation, the time complexity analysis of 1D knot scenario applies here, with $O(N)$ complexity. In addition, we can also reuse Algorithm 7 and Algorithm 8 directly for 2D surface scenario.

IV. Sequential Access Pattern of Collision Avoidance

In Sec. II and Sec. III, we discussed how to parallelize the forces calculation and position adjustment based on the key observation that the calculation for one bead is independent of that of another. However, the property does not hold for collision avoidance procedure.

In 1D knot scenario, each iteration of the outer loop works on one segment formed by two adjacent beads in the line, and the distance between the segment in question and each of non-neighbor segments is calculated through an inner loop. If the distance falls within the specified threshold, the segment and its neighbor segment need to be pulled apart so the adjusted distance reaches the threshold. As we can see, not only iterations in outer loop are dependent but also the ones

Algorithm 10: Total Force Task - 2D Surface

Input : $NumWorkers$: number of worker threads, $threadId$: worker thread Id, $numRows$: # rows of 2D matrix, $numCols$: # columns of 2D matrix, $repulsiveForceMatrix$: repulsive force matrix, $attractiveForceMatrix$: attractive force matrix

Output: None

```
1  numBeads ← length (repulsiveForceMatrix);
2  foreach i ∈ range (numBeads) do
3      if i mod NumWorkers != threadId then
4          continue;
5      #Sum up attractive forces from right and lower neighbors
6      v = attractiveForceMatrix[i, 0, :] − attractiveForceMatrix[i, 1, :];
7      #Sum up attractive forces from left and upper neighbors
8      rowIdx = i // numCols;
9      colIdx = i mod numCols;
10     if colIdx − 1 > 0 then
11         v ← v − attractiveForceMatrix[i − 1, 0, :];
12     if rowIdx − 1 > 0 then
13         v ← v − attractiveForceMatrix[i − numCols, 1, :];
14     #Sum up all repulsive forces
15     foreach j ∈ range (i) do
16         v ← v + repulsiveForceMatrix[i, j, :];
17     foreach j ∈ range (i + 1, numBeads) do
18         v ← v − repulsiveForceMatrix[j, i, :];
19     #Subtract repulsive forces from four adjacent neighbors
20     #Left neighbor
21     if colIdx − 1 > 0 then
22         v −= −repulsiveForceMatrix[i, i − 1, :];
23     #Upper neighbor
24     if rowIdx − 1 > 0 then
25         v −= repulsiveForceMatrix[i, i − numCols, :];
26     #Right neighbor
27     if colIdx + 1 < numCols then
28         v += repulsiveForceMatrix[i + 1, i, :];
29     #Lower neighbor
30     if rowIdx + 1 < numRows then
31         v += repulsiveForceMatrix[i + numCols, i, :];
32     totalForceMatrix[i, :] ← v;
```

in the inner loop, since the position of a segment may need to be adjusted repeated based on its distance to other segments.

In 2D surface scenario, the collision avoidance not only requires segment distance detection (*i.e.*, line to line distance) as in 1D knot case but also involves point-triangle distance detection (*i.e.*, the distance of a point to any virtual triangle that it does not belong to). In a similar way, the position of a segment, point, or triangle need to be adjusted if its distance to another party falls within the threshold.

In summary, during collision avoidance the state of one party (*i.e.*, segment, point or triangle) depends on its prior state and the state of involved party (when the distance falls below threshold), therefore collision avoidance exhibits sequential access pattern and cannot be parallelized.

V. Performance Tuning

In Sec. II and Sec. III we propose parallel algorithms for functional units that perform force calculation and position adjustment. Here we discuss two aspects that have impact on performance: threading model and level of parallelization.

A. Thread Pool Model

Traditionally, to speed up a parallel eligible job with multi-threading, a set of worker threads are spawned before the job begins and are shut down after the job completes. For a mathematical simulation where the number of iterations can be large (*e.g.*, more than $5,000$ iterations), this can be problematic if we need to do so for every iteration and every functional unit calls within an iteration, since the total cost incurred by the overhead of thread fork and reclaim is nontrivial when the overhead is proportional to the number of iterations.

To handle such issue, we make worker threads long living across the whole simulation process instead of ephemeral for just a particular functional unit call. We use a thread pool to maintain the threads and each call just checks out X number of threads from the pool based on its computational need and returns the threads to the pool once its job is done. By such means, we effectively eliminate the recurring need of thread fork and reclaim.

B. Level of Parallelization

As a rule of thumb, in determining the level of parallelization (*i.e.*, the # of worker threads), the overhead incurred by threads should not outweigh the performance gain from distributing the computation. In general, two important factors to consider are the scale of the input and the time complexity of the task. In our specific scenario, 2D mathematical surface definitely has much more number of beads than 1D knot. In addition, calculations for repulsive and total forces are $O(N^2)$ time complexity and therefore are more computational intensive than calculations for attractive force and position adjustment which are linear $O(N)$ time complexity. These should serve as good indicators for choosing an appropriate multi-threading level.

TABLE I
Runtime (in minute) comparison under different parallelization levels, with a 96-bead knot and 1,000 iterations. The number in parentheses is the normalized one.

Serial	2-1-2-1	4-1-4-1	4-2-4-2	6-1-6-1
1.71 (100.0%)	1.56 (91.2%)	1.89 (110.5%)	2.03 (118.7%)	2.15 (126.3%)

VI. Experiments

Experimental Setup. We run our experiments on a Linux server with following hardware specification: two 8-core Intel Xeon processors, 256 GB of RAM, and 500 GB of hard disk drive. The dataset for 1D knot simulation contains 96 beads and for 2D surface we use one matrix of size 50 by 50. Since collision avoidance has sequential access nature, in experiments we bypass this procedure and only focus on force calculation. In addition, for the purpose of obtaining stable results, the figures presented below are the average of 10 independent runs. For each setting, we use format *A-B-C-D* to indicate the set of parallelization levels we use, where A indicates the # of threads for repulsive force, B indicates the # of threads for attractive force, C indicates the # of threads for total force, and D indicates the # of threads for new position update.

Table I shows for a 1D knot of size 96 beads, the runtime comparison under different parallelization levels with 1,000 iterations. We can see that by using 2 threads for the calculation of repulsive force and total force, approximately 10% reduction in computational time can be achieved. However, further increasing the multi-threading level leads to performance degradation. We argue that this is because 1D knot does not have sufficient data and therefore for higher parallelization level overhead of managing threads outweighs distributing the computation. In Figure 4, we show the bar char of runtime figures used by Table I.

Table II shows for a 2D surface of size 50 by 50 matrix, the runtime comparison under different parallelization levels. Since the computational cost of a 2D surface is significantly larger than a 1D knot, here we only show the runtime for one iteration. Number in parentheses shows the normalized runtime in terms of percentage with serial execution's runtime as the normalizer. Because of 2D surface has significantly larger data to process compared with 1D knot, we can accordingly see that with multithreading we can achieve better speedup with at least 40% of runtime reduction. Furthermore, as discussed in Sec. V-B, for calculations of attractive force and position update which have linear or $O(N)$ time complexity, using more threads does not yield significant performance gain. In addition, Figure 5 shows the bar char of runtime figures used by Table II.

VII. Conclusion and Future Work

In this paper we proposed a suite of parallelized topological relaxation algorithms for 1D mathematical knot and 2D surface simulation. The overall idea is based on the

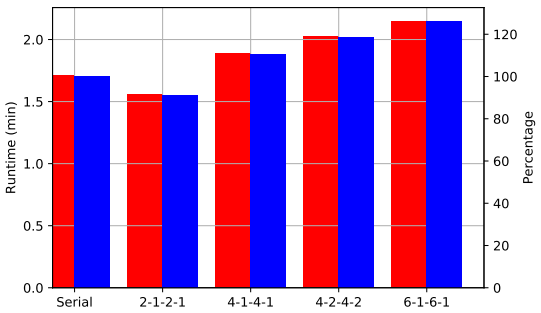

Fig. 4. Runtime comparison under different parallelization levels, with a 96-bead knot and 1,000 iterations. Red bar for raw time and blue bar for normalized time.

TABLE II
RUNTIME (IN MINUTE) COMPARISON UNDER DIFFERENT PARALLELIZATION LEVELS, WITH A 50 BY 50 MATRIX AND ONE ITERATION. THE NUMBER IN PARENTHESES IS THE NORMALIZED ONE.

Serial	2-1-2-1	4-1-4-1	4-2-4-2	6-1-6-1
1.15 (100.0%)	0.69 (60.0%)	0.53 (46.1%)	0.54 (47.0%)	0.53 (46.1%)

observation that force calculation can be decoupled into fine-grained functional units and that the calculation of one bead is independent of another.

In general, we can abstract the data structure used for beads (apply for both 1D and 2D scenarios) as linked data in a graph and device similar algorithms to update the state of each node in the graph in a parallel manner as long as there is no dependency constraint for state update amongst different nodes. This observation opens a door for the possibility of speeding up scientific simulation in other domains and will be our future direction of exploration.

REFERENCES

[1] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, "Algorithms for drawing graphs: An annotated bibliography," *Comput. Geom. Theory Appl.*, vol. 4, no. 5, pp. 235–282, Oct. 1994. [Online]. Available: http://dx.doi.org/10.1016/0925-7721(94)00014-X

[2] H. Zhang, S. Thakur, and A. J. Hanson, "Haptic exploration of mathematical knots," in *ISVC (1)*, 2007, pp. 745–756.

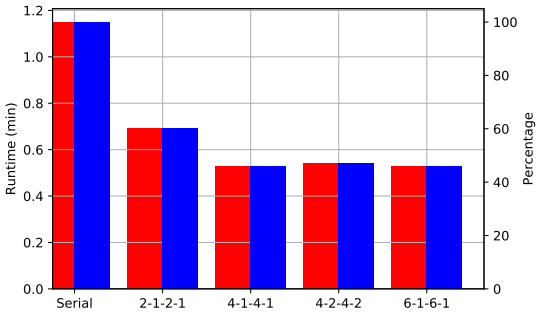

Fig. 5. Runtime comparison under different parallelization levels, with a 50 by 50 matrix and one iteration. Red bar for raw time and blue bar for normalized time.

[3] L. Jing, X. Huang, Y. Zhong, Y. Wu, and H. Zhang, "Python based 4d visualization environment," *International Journal of Advancements in Computing Technology*, vol. 4, no. 16, pp. 460–469, September 2012, http://www.aicit.org/dl/citation.html?id=IJACT-1290[Link].

[4] H. Zhang, J. Weng, and A. J. Hanson, "A pseudo-haptic knot diagram interface," in *Proc. SPIE*, vol. 7868, 2011, pp. 786 807–786 807–14, http://dx.doi.org/10.1117/12.872409[Link]. [Online]. Available: http://dx.doi.org/10.1117/12.872409

[5] H. Zhang and A. J. Hanson, "Shadow-driven 4d haptic visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1688–1695, Nov 2007, http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4376203[Link].

[6] J. Weng and H. Zhang, "Pseudohaptic interaction with knot diagrams," *Journal of Electronic Imaging*, vol. 21, no. 3, p. 033008, 2012, http://spie.org/Publications/Journal/10.1117/1.JEI.21.3.033008[Link].

[7] J. S. Carter, "Reidemeister/roseman-type moves to embedded foams in 4-dimensional space," 2012.

[8] R. G. Scharein, "Interactive topological drawing," Ph.D. dissertation, Department of Computer Science, The University of British Columbia, 1998.

[9] J. Weeks, "Snappea: a computer program for creating and studying hyperbolic 3-manifolds," 2001.

[10] J. Phillips, A. M. Ladd, and L. E. Kavraki, "Simulated knot tying," in *ICRA*. IEEE, 2002, pp. 841–846.

[11] J. Spillmann and M. Teschner, "An adaptive contact model for the robust simulation of knots," *Comput. Graph. Forum*, vol. 27, no. 2, pp. 497–506, 2008.

[12] H. Zhang and A. J. Hanson, "Physically interacting with four dimensions." in *ISVC (1)*, ser. Lecture Notes in Computer Science, G. Bebis, R. Boyle, B. Parvin, D. Koracin, P. Remagnino, A. V. Nefian, M. Gopi, V. Pascucci, J. Zara, J. Molineros, H. Theisel, and T. Malzbender, Eds., vol. 4291. Springer, 2006, pp. 232–242.

[13] H. Zhang and A. Hanson, "Shadow-driven 4d haptic visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1688–1695, 2007.

[14] H. Zhang, J. Weng, L. Jing, and Y. Zhong, "Knotpad: Visualizing and exploring knot theory with fluid reidemeister moves," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 18, no. 12, pp. 2051 –2060, dec. 2012.

[15] H. Zhang, J. Weng, and G. Ruan, "Visualizing 2-dimensional manifolds with curve handles in 4d," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2575–2584, Dec 2014, http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6876027[Link].

[16] G. Ruan, H. Zhang, and B. Plale, "Parallelizing dbscan algorithm with distributed iterative graph processing," in *Proceedings of the 2014 Workshop on Big Data Analytics: Challenges and Opportunities, in conjunction with ACM/IEEE SuperComputing 2014*, ser. BDAC'14, 2014.

[17] G. Ruan, H. Zhang, and B. Plale, "Parallel and quantitative sequential pattern mining for large-scale interval-based temporal data," in *2014 IEEE International Conference on Big Data (Big Data)*, Oct 2014, pp. 32–39.

[18] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation*, ser. OSDI'04, 2004.

[19] G. Ruan and H. Zhang, "Visual analysis of large dental imaging data in caries research," in *2014 IEEE Symposium on Large Data Analysis and Visualization*, ser. LDAV'14, 2014, pp. 77–84.