OCTOPUS: Efficient Query Execution on Dynamic Mesh Datasets

Farhan Tauheed[†‡], Thomas Heinis[†], Felix Schürmann[‡], Henry Markram[‡], Anastasia Ailamaki[†]

[†]*Data-Intensive Applications and Systems Lab,* [‡]*Brain Mind Institute*
École Polytechnique Fédérale de Lausanne, Switzerland

***Abstract*—Scientists in many disciplines use spatial mesh models to study physical phenomena. Simulating natural phenomena by changing meshes over time helps to understand and predict future behavior of the phenomena. The higher the precision of the mesh models, the more insight do the scientists gain and they thus continuously increase the detail of the meshes and build them as detailed as their instruments and the simulation hardware allow. In the process, the data volume also increases, slowing down the execution of spatial range queries needed to monitor the simulation considerably. Indexing speeds up range query execution, but the overhead to maintain the indexes is considerable because almost the entire mesh changes unpredictably at every simulation step. Using a simple linear scan, on the other hand, requires accessing the entire mesh and the performance deteriorates as the size of the dataset grows.**

In this paper we propose OCTOPUS, a strategy for executing range queries on mesh datasets that change unpredictably during simulations. In OCTOPUS we use the key insight that the mesh surface along with the mesh connectivity is sufficient to retrieve accurate query results efficiently. With this novel query execution strategy, OCTOPUS minimizes index maintenance cost and reduces query execution time considerably. Our experiments show that OCTOPUS achieves a speedup between 7.2 and 9.2× compared to the state of the art and that it scales better with increasing mesh dataset size and detail.

I. Introduction

No longer do scientists solely depend on the study of the phenomena in their laboratory or in nature. They nowadays improve their understanding by building mesh representations of natural phenomena and by simulating them. Numerical techniques like the finite element method (FEM) [26] with three dimensional mesh representations at their core are used in many disciplines. Examples include earthquake simulations, material deformation, fluid dynamics, growth processes in neuroscience and many more. The size of the meshes scientists use today ranges between several gigabytes and terabytes but will certainly grow. Scientists perpetually improve the precision of their simulations by building increasingly detailed (fine-grained) and thus bigger mesh models, as precise as their instruments allow and as big as the available memory permits.

A typical mesh simulation is carried out by changing the memory resident mesh datasets at discrete time steps to mimic the behavior of the natural phenomenon. While the meshes are changed in memory, the simulations need to be monitored and analyzed at runtime so that the scientists can steer the simulation interactively, stop, reset & repeat it, fix parameters etc. or to visualize it on the fly [5]. Efficient access to spatial mesh data during simulations becomes key as monitoring tools frequently need to retrieve parts of the mesh data for analysis.

Monitoring mesh simulations thus translates into the challenge of efficiently executing spatial range queries on a memory resident mesh dataset that is updated by changing the position of the mesh vertices in-place at every time step of the simulation. The updates of the vertices' positions are unpredictable (due to the unpredictable nature of the simulation) and massive, affecting the entire dataset. To monitor and analyze the simulation, spatial range queries are executed on the mesh between the discrete time steps simulated.

Executing the few queries needed to monitor the mesh simulation after every time step, followed by massive unpredictable updates to the spatial mesh dataset is a new extreme in the context of the query/update tradeoff studied in past work. Due to the unpredictable movement we cannot reduce the cost of maintaining an index by assuming that the objects move in a predictable trajectory, which is how it is used many applications indexing continuously moving objects [18], [19]. The cost of rebuilding lightweight spatial indexes [8] at every time step or maintaining indexes designed specifically to reduce update cost [13], [24] cannot be amortized over the few queries executed at every time step. In this extreme scenario a simple linear scan testing all mesh vertices for intersection with the query yields the best performance, but will only scale linear with mesh dataset size. Key to a new approach for executing range queries on mesh simulations therefore is that it scales well for increasing mesh dataset size and detail.

In this paper we develop OCTOPUS, an execution strategy for range queries on meshes that undergo massive updates. Because OCTOPUS solely relies on the mesh connectivity it is oblivious to changes to the position of mesh vertices and can efficiently execute range queries even in the face of changes to the entire mesh. As opposed to traditional indexes, OCTOPUS minimizes costly maintenance of data structures and, unlike the linear scan, it only needs to retrieve mesh data in the order of the size of the query result and thus scales sublinear with the size of the mesh dataset. With OCTOPUS we make the following contributions:

- fundamental to OCTOPUS' efficiency in face of massive changes to mesh positions is that it only needs to index the mesh surface, a small subset of the entire mesh, thereby minimizing index maintenance overhead. A small part of the query result is retrieved from the mesh surface index while the major share is retrieved by traversing the unindexed

mesh dataset in-memory.

- we build OCTOPUS on the key insight that the surface of the mesh provides sufficient information to start a mesh traversal to retrieve the complete query result, independent of the mesh geometry that may change during simulation.
- for applications where the mesh geometry remains convex during simulation we design OCTOPUS to use an outdated (stale) spatial index that can be used without sacrificing accuracy of query results.

With an extensive experimental analysis using neuroscience, earthquake and non scientific animation datasets, we show how OCTOPUS outperforms existing approaches while remaining competitive in terms of memory footprint. Most importantly, we demonstrate that OCTOPUS scales with increasing mesh size and detail.

The remainder of the paper is structured as follows. In Section II we discuss related work and in Section III we motivate our work. In Section IV we present OCTOPUS and discuss optimizations. We compare OCTOPUS to related approaches as well as analyze its performance in subsequent sections and finally draw conclusions in Section IX.

II. Related Work

Several approaches have been developed to analyze and visualize scientific simulations [1], [22] at runtime but they unfortunately provide only approximate results and are therefore inadequate for our problem.

Performing a complete linear scan over the entire dataset is arguably the most basic approach to compute the accurate result of a range query. While the linear scan has no memory overhead, query execution time will not scale as it directly depends on the dataset size. Indexes can reduce the time for query execution but suffer from high maintenance cost. Various optimizations have been proposed to maintain indexes in case of updates on one-dimensional data [9] but more fitting for our challenge, however, are indexes developed for spatial data. In the following we distinguish between spatial and moving object indexes and include approaches developed for disk as they can be used in memory as well.

A. Spatial Indexes

Several spatial indexes [7], [17] have been particularly devised for meshes. These indexes are very efficient in executing queries, but unfortunately they are very costly to update, e.g., DLS [17] needs to recompute the Hilbert-value for each moving object. Additionally, they only work on particular mesh types, e.g., DLS only works with static meshes with convex geometry, making them unsuitable for our application.

Spatial indexes used for generic datasets like the lazy update R-Tree (LUR-Tree) [13] or the LU-Grid [25] are optimized for efficiently supporting few updates. These two approaches reduce the update cost by avoiding expensive index maintenance, if the change in location of the updated object is very low. The LUR-Tree, on the other hand, avoids costly R-Tree insertions if the object remains inside the minimum bounding rectangle of the leaf node. Instead of indexing the moving objects, QU-Trade [24] indexes a grace window within which the objects are expected to move. The bigger the grace window is, the fewer updates need to be made but also the more irrelevant objects are retrieved by a query. By growing and shrinking the grace window this technique provides a good, tunable compromise between update and query intensive workloads.

If, however, a massive number of updates needs to be executed it is often cheaper to rebuild the index from scratch entirely [8] (at every simulation step in our scenario). This strategy can be used with several memory-based spatial indexes like the Octree [10], the Kd-Tree [4] or memory optimized R-Trees [11].

Approaches using buffered updates [6] work very well for disk based datasets. In-memory, these approaches result in performing bulk-updates. For our problem bulk-updates are equivalent to rebuilding the entire index because almost the entire dataset changes. Similarly, the memo-based update approach used in RUM-Tree [20] works by inserting the new positions into an R-Tree index and by invalidating (but not deleting) the past state of the object. In our scenario this approach requires repetitive insertions of all objects in the R-Tree at each timestep, which clearly is slower than bulkloading a new index.

B. Moving Object Indexes

A class of indexes closely related to our problem are spatio-temporal indexes used to index moving objects [15]. Spatio-temporal indexes can be classified based on whether they can answer range queries on the past, present or future location of the moving objects. In the context of monitoring applications we are only interested in monitoring the present state and thus only consider indexes that support queries on the present state.

Some moving object indexes exploit the predictability of the objects' movements to reduce index maintenance cost. The adaptive two-level hashing approach [12] classifies objects according to their speed of movement. Slow moving objects are indexed with a fine-grained grid whereas it uses a coarse-grained grid for fast objects. The index only needs to be updated once the object moves out of the grid cell. Queries retrieve all grid cells intersecting with the query and filter the objects that intersect with the grid cell but not the query.

To reduce the overhead of frequent updates, other approaches [18], [19] exploit that the movement of the objects is predictable and that it can be approximated with a trajectory. The movement is approximated using curve extrapolation and the index only has to be updated once the trajectory of the moving objects changes. In the case of mesh simulations, however, the movements are not predictable and the trajectories are not even static for short periods of time.

III. Motivation

The development of OCTOPUS is driven by the needs of scientists who face significant performance bottlenecks when querying mesh datasets that undergo massive changes in every time step of a simulation. In the following we first discuss scientific simulation applications, how the associated mesh datasets are queried (to monitor progress of the simulations) and we finally formalize the data management challenge.

A. Monitoring Simulation Applications

Simulations in physical science predominantly use three dimensional mesh representations [26]. The meshes used consist of millions of spatial polyhedral objects, each with a volume. The smaller the polyhedra are, the more fine-grained the model becomes and therefore also the more detailed and accurate the simulation is. An important trend in the simulation sciences is to increase the precision of the simulation [3] and hence the size of the mesh datasets used keeps growing.

Mesh based simulations can be categorized by the polyhedral primitives used. Different primitives provide different degrees of freedom, i.e., the more vertices and edges a polyhedra has, the more precise can deformation be simulated. The exact primitive used depends on the phenomena simulated. Example primitives like 3D tetrahedral mesh or 3D hexahedral mesh are illustrated in Figure 1(a) and (b).

The physical representation of mesh dataset differs with the implementation, but typically an adjacency list representation is used. The adjacency list stores for each vertex the position as well as pointers to neighboring (connected with an edge in the mesh) vertices in the list. The neighbor pointers can be used to access connected mesh vertices and hence no costly sequential scan or additional data structure is required. Additionally, a list of polyhedra and polyhedral faces is kept to provide a mapping for each polyhedra to its corresponding vertices. The overall geometry of the mesh may also differ in different simulation applications, for example, an earthquake simulation uses a mesh that remains convex during simulation, while a neuroscience simulation uses a non-convex geometry mesh as shown in Figure 1(c) and (d).

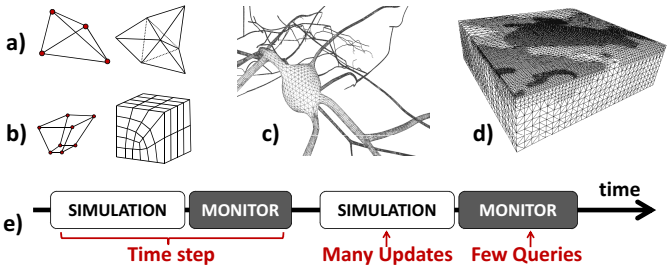


Fig. 1: Polyhedral mesh structure, (a) tetrahedral mesh, (b) hexahedral mesh, (c) neuron mesh, (d) earthquake mesh, (e) simulation timeline.

During simulation, mesh datasets are processed in-memory on a single or distributed hardware infrastructure for high performance. Figure 1(e) shows how a simulation is executed in discrete simulation steps. In each time step the simulation software takes as input a memory resident mesh, calculates the changes and updates the mesh in-place (overwriting the mesh in memory). The updates typically are minute changes to the positions of all vertices in the mesh dataset.

After the simulation has updated all vertex positions in one simulation step, different parts of the mesh are accessed by executing three dimensional range queries on the latest state in memory. The queries are issued by the monitoring tools to retrieve parts of the mesh inside the query region for statistical validation, for visualization or to steer the simulation and they are thus different in every time step. Monitoring tools are designed to work with any simulation, the simulation software is thus treated as a black-box and the simulation as well as monitoring step shown in Figure 1(e) cannot be merged.

B. Neuroscience Example

An example of a simulation based on three dimensional mesh is used by the neuroscientists we collaborate with in the Blue Brain project [14]. During brain simulation, the process of neural plasticity constantly rewires the neurons, i.e., it adds/removes synapses connecting different neurons over time (synapses are the structures that allow neurons to exchange signals with each other). By simulating the changes of synapses in discrete time steps, neuroscientists improve their understanding of communication between neurons.

Different monitoring applications are used that execute spatial range queries at each time step of the simulation. The queries are not fixed for each time step and the location and volume of the queries depends on the particular monitoring use case. The following three applications are frequently used:

- Structural Validation: range queries are executed to retrieve mesh data to perform statistical analysis, e.g., computing the neuron density, the number of branches in a given area, etc. The statistics are computed to ensure neuron mesh models remain bio-realistic during simulation.
- Mesh Quality: in the process of simulating neural plasticity mesh models are deformed. Deformation potentially leads to artifacts like the intersection of meshes representing different neuron branches. Range queries are executed to analyze model intersection particularly in the dense neural regions where the probability of finding artifacts is high.
- Visualization: Simulations can be visually monitored to show the progress over time. To visualize the subsets of the mesh, the view frustum needs to be retrieved using range queries. The quality of the visualization defines the number and size of the range queries required.

Today the neural mesh model consists of 1.3 billion tetrahedra or 33GB. Executing range queries for monitoring is already slow today and will become even slower as the neuroscientists will increase the detail of the meshes. Monitoring tools that scale with the increasing mesh detail are therefore pivotal.

C. Data Management Challenge

Indexing moving objects has been extensively studied in the past. In the context of the challenge of monitoring mesh simulations, however, the usefulness of state-of-the-art approaches is limited. Monitoring mesh simulations is a new extreme access pattern where massive updates of almost the entire dataset are followed by comparatively few queries.

Using state-of-the-art indexes to support monitoring mesh simulations is unlikely to be efficient: positions of all mesh vertices change during each simulation time step and the cost of updating an index or rebuilding it from scratch cannot be amortized over the few queries monitoring applications need to execute. Updating or rebuilding the entire index requires more operations (inspecting the current position of object,

building the index structure and then executing the queries) than performing a simple linear scan to retrieve the query results at every time step.

The complexity of the linear scan, however, depends on the dataset size. As the precision of the simulations increases, the level of detail and size of the meshes will increase at the same rate. Any approach like the linear scan that depends on the dataset size will not scale well with future datasets.

IV. The OCTOPUS Approach

OCTOPUS is a strategy for executing range queries on dynamic meshes. Even with unpredictable changes affecting the entire mesh dataset OCTOPUS outperforms the linear scan approach by (1) avoiding to maintain data structures in case of changes to the vertex positions in the mesh and by (2) accessing only the subset of the dataset needed to compute the query result accurately.

In the following, we first give a brief overview of the approach and then explain in more detail how OCTOPUS executes range queries. We then present a variant of OCTOPUS specifically designed for meshes with convex geometry. Finally, we develop an analytical model to predict the runtime performance of OCTOPUS and to identify the conditions where OCTOPUS yields better results than the linear scan.

A. Algorithm Overview

OCTOPUS uses the connectivity inherent in the mesh to avoid accessing the entire mesh when retrieving query results. Using the current state of the mesh directly from memory has the key advantage that, given one vertex inside the query region, OCTOPUS can use the mesh connectivity to retrieve vertices in the query range without having to consider the change of the vertex location in the last update.

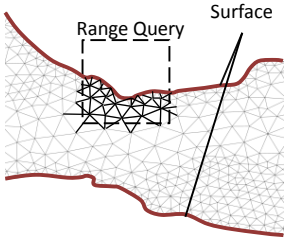


Fig. 2: OCTOPUS: Starting from the surface, the edges of the polyhedra will be visited.

Solely depending on the connectivity of the mesh, however, bears the risk that the result is not complete because parts of the mesh in the query region may not be connected. We address this problem by using the mesh surface, thereby relying on the key insight that every vertex inside a query region Q is reachable from a surface vertex inside Q.

OCTOPUS uses a mesh surface index to compute query results accurately. The index used is not a spatial index but a geometrical index designed to provide quick access to a subset of mesh vertices belonging to the surface. Key to the mesh surface index is that it rarely needs maintenance because the mesh surface only changes in the rare event where the connectivity of the mesh changes. With the surface index and the mesh itself OCTOPUS executes range queries in the following three phases:

1) **Crawling:** OCTOPUS traverses the edges of the mesh polyhedra, thereby exploiting the connectivity of the mesh. Vertices are retrieved recursively and the traversal continues until all vertices inside the query are retrieved.
2) **Surface Probe:** To provide starting vertices for the crawling phase, OCTOPUS scans all surface vertices using the surface index and identifies those inside the query region.
3) **Directed Walk:** If no surface vertex exists inside the query region, OCTOPUS traverses the edges of the mesh always picking the edge that leads to a vertex closer to the query region until a vertex inside the query region is found.

Figure 2 illustrates the algorithm in two dimensions: OCTOPUS first chooses a vertex inside the range query (dashed line) by probing all surface vertices (boundary in 2D; bold, solid line) and then recursively traverses all edges (thin black lines) inside the query range to retrieve all vertices. In the following we discuss the three phases of the OCTOPUS query execution, crawling, surface probe and directed walk, in more detail.

B. Crawling

An important characteristic meshes share independently of the particular polyhedral primitives used (e.g., tetrahedra, hexahedra) is that they can be understood as a graph, where the vertices of the polyhedra are the vertices of the graph and the edges of the polyhedra connect the vertices in the graph.

Given the graph representation of meshes, OCTOPUS crawls along the edges of the polyhedra using a breadth-first-search to retrieve the vertices inside the query region. OCTOPUS keeps track of edges visited and stops following an edge if its end vertex is outside the query region. The entire crawl process stops once all vertices inside the query region are visited. This stop criteria ensures that all vertices in the query are retrieved and that the number of vertices visited solely depends on the query selectivity.

The geometry of the mesh influences the accuracy of the crawling phase. The query result computed by the crawling phase is accurate (or complete) if the mesh has *internal reachability*, i.e., if (a) there exists a path between any two vertices inside the query region and (b) that path only contains vertices that are inside the query region. If internal reachability is not given, not all vertices of the result may be retrieved.

Meshes used in simulations can generally be divided into two classes: convex and non-convex meshes [26]. A mesh is convex if a straight line connecting any pair of its vertices remains inside the mesh. A convex mesh will remain convex during a simulation if the changes it undergoes only affect the positions of its vertices. Convex meshes are frequently used for fluid dynamics or earthquake simulations [3].

The intersection of a rectangular range query with a convex mesh is always a convex subset of the mesh that satisfies complete internal reachability. OCTOPUS' crawl along the vertices of a convex mesh to execute a rectangular range query is consequently always accurate. A comprehensive proof of the accuracy of OCTOPUS' query results on convex meshes is given in the technical report [23].

If, on the other hand, a mesh contains holes or the straight line between any two vertices crosses the mesh surface, i.e., is outside the mesh, then the mesh is non-convex. As we discuss in the next subsection, OCTOPUS can still execute range

queries accurately on non-convex meshes or in the scenario where a mesh transforms from a convex to a non-convex geometry (and vice versa).

C. Surface Probe

Not all meshes used in simulations are convex and during simulation the mesh dataset may undergo changes that make it non-convex (concave or disjoint). In the following we discuss how OCTOPUS addresses the problem of non-convex meshes by starting crawling from multiple starting vertices on the mesh surface to guarantee accuracy.

Unlike convex meshes, the intersection between a query and non-convex mesh can result in not one, but several disjoint sub-meshes (e.g., in Figure 3 the mesh is split into two disjoint sub-meshes). In this case complete internal reachability is no longer given and crawling from any arbitrary vertex inside the query may only retrieve a part of the results. Crawling outside the query ensures that the complete result is retrieved if there exists a path that connects the disjoint sub-meshes. Even if a path exists outside the query region it is difficult to define a stop criteria and this may end up visiting the entire mesh.

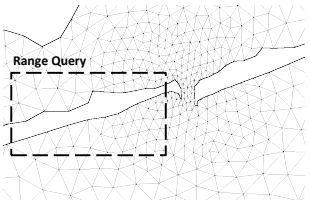


Fig. 3: Non-convex mesh where a query divides mesh into two unreachable parts.

We address the problem of non-convex meshes by treating the disjoint sub-meshes as a collection of meshes that independently satisfy the criteria of complete internal reachability. All OCTOPUS needs to compute the complete query result is one or several vertices in each sub-mesh from where to start the crawl. What separates the disjoint meshes from each other is the mesh surface and empty space. Consequently (and as Figure 3 shows), each disjoint sub-mesh obtained by the intersection of the query and a non-convex mesh contains at least one surface vertex inside the query range. By starting the crawl from all surface vertices inside the range query, OCTOPUS computes the accurate result of a query. A proof of correctness is given in the technical report [23].

Non-convex meshes are the reason why we use the surface vertices inside the query range as starting points of the crawl. To execute a query, OCTOPUS retrieves all vertices from the mesh surface index, probes them to find those inside the query range and finally starts to crawl from each of them. As opposed to the linear scan, OCTOPUS does not need to iterate over the entire dataset, but only needs to probe the mesh surface (as well as crawl along the edges in the query). Consequently, compared to approaches where the query execution time depends on the dataset size, OCTOPUS will scale better for future datasets because the subset of surface vertices shrinks for future datasets (the mesh surface increases quadratic while the number of non-surface vertices grows cubic).

D. Directed Walk

So far we have assumed that there is always at least one surface vertex inside the query region to start crawling. No surface vertex, however, is in the query if (a) the query does not intersect with the mesh at all (empty query) or (b) if the query is completely enclosed inside the mesh.

In either case the surface probe will find no surface vertex and we instead use the surface probe to find the surface vertex v closest to the query q by computing the minimum euclidean $distance(v, q)$. From v we start a *directed walk*, similar to a depth first search that always recursively picks the next neighboring vertex that is closest to the query region. This process continues recursively until a vertex inside the query region is found. The first vertex inside the query region is used as the starting vertex for the crawling phase. Algorithm 1 illustrates the query execution algorithm with pseudocode.

If during the directed walk no vertex inside the query region or closer to it than the previous vertex can be found, then the

Algorithm 1 OCTOPUS Query Execution

Input: q: range query, S: surface index
Output: resultSet = 0
Data: sv: start vertices = 0, minVertex = 0, minDistance = $\infty, oldMinDistance = \infty$

```
// Surface Probe
foreach v ∈ S do
    if v enclosed inside q then
        put v into sv
    end
    if sv = ∅ then
        if distance(v, q) < minDistance then
            minVertex ← v
            minDistance ← distance(v, q)
        end
    end
end
//Directed Walk
if sv = ∅ then
    while true do
        if minVertex enclosed inside q then
            put minVertex into sv
            break
        end
        foreach v ∈ neighbor(minVertex) do
            if distance(v, q) < minDistance then
                minVertex ← v
                oldMinDistance ← minDistance
                minDistance ← distance(v, q)
            end
        end
        if minDistance = oldMinDistance then
            break
        end
    end
end
//Crawling
foreach v ∈ sv do
    resultSet ← BreadthFirstSearch(v, q)
end
return resultSet
```

query does not intersect with the mesh and the result is empty.

E. Indexing the Surface

Keeping track of vertices on the surface is critical for ensuring accurate query execution. OCTOPUS uses an index (surface index) solely during the surface probe phase of the query execution. The surface index is built once before the simulation starts and is only maintained in the (rare) case where the connectivity of the mesh changes. If only the positions of the vertices change, the surface index does not need any maintenance.

1) Indexing Building: OCTOPUS identifies the vertices on the surface by accessing the polyhedra face list and constructing a global face list. Multiple polyhedral faces make up any polyhedral, e.g., four triangles make up a tetrahedral. The global face list contains all faces for each polyhedron in the dataset and keeps the pointers to the vertices for each face. A face may be shared by at most two adjacent polyhedra (see Figure 1(a)) and so duplicate faces exist in the list. A face F belongs to the mesh surface if it occurs once in the list, i.e., there exists no adjacent polyhedra that shares face F.

The surface index is implemented using a hash table where the vertex identifier serves as the hash-key and the hash-value represents a pointer to the surface vertex in memory. During the surface probe, all surface vertices are accessed via the pointers in the hash table in no particular order.

2) Index Maintenance: The mesh can undergo two different transformations during simulation: (1) mesh deformation, where only the position of the vertices of the mesh change and (2) mesh restructuring, where the connectivity in the mesh and thus the surface can change.

The surface index does not require any maintenance in the case of mesh deformation because the connectivity does not change: any vertex on the surface remains on the mesh surface although the location of the vertex is changed. Restructuring the mesh during simulation, on the other hand, can change the surface vertices as polyhedra may be split, thus increasing the number of vertices on the surface, or merged, hence reducing the vertices on the surface. In this case the surface index is updated with insert or delete operations on the hash table used in the index. Although, OCTOPUS supports mesh being restructured during simulation, the transformation is rarely implemented [26] in practice.

F. Convex Meshes

Computational fluid dynamics, material science and environmental science frequently use simulations of solid, liquid and gaseous phenomenon restricted inside a convex container. Improving the efficiency of range query execution on convex meshes is therefore particularly important. As discussed previously, convex meshes satisfy internal reachability, i.e., any vertex can be reached from any other vertex by crawling along the mesh edges. In case of convex meshes OCTOPUS hence does not need to build a surface index because the surface probe is not required. Instead, a directed walk can start from any vertex to reach the query region. The cost of the directed walk, however, can become substantial if the directed walk starts far away from the query region.

To find a starting vertex close to the query region for the directed walk OCTOPUS-CON thus uses a grid based spatial index for the mesh. The grid index is built once and never updated. Although the index is quickly outdated it still can help to find a starting point close to the query needed to reduce the cost of the directed walk. Using an outdated index to find a starting point close to the range query is fundamentally different than using a spatial index for the entire mesh: in the former case we can tolerate an outdated index, in the latter case query execution may not compute correct results.

OCTOPUS-CON uses a simple three dimensional uniform grid as spatial index. Before the simulation, the index is built by mapping each vertex of the mesh to the grid cell enclosing the vertex. To find the closest vertex OCTOPUS-CON finds the cell that encloses the center of the query region and then uses any of the mesh vertices assigned to this cell to start the directed walk. If no vertex exists the neighboring cells are recursively checked until a vertex is found. The grid index provides a cost effective means to reduce directed walking cost. After completing the directed walk the crawling phase follows to retrieve the accurate query result as described in Algorithm 1.

G. Analytical Model

We develop an analytical model to predict the performance benefits of OCTOPUS and confirm with experiments its accuracy (predictions within 2% error).

OCTOPUS' total execution cost entails the cost of probing the surface vertices, the directed walk and crawling. For the sake of simplicity we exclude the time for the directed walk from the model (experimentally we show that the time spent on directed walk is insignificant on average). The remaining costs are then defined as follows:

Surface Probe Cost: If V is the total number of vertices in the dataset and S is the surface-to-volume ratio (the number of surface vertices divided by the total number of vertices), the number of surface vertices is $S \times V$. Equation 1 describes the cost of the surface probe that accesses each surface vertex once, where C_S is defined as the constant cost of accessing a single vertex sequentially and comparing it to the range query:

$$Cost(SurfaceProbe) = C_S \times (S \times V) \quad (1)$$

Crawling Cost: Crawling needs to follow all edges inside the query region. With the query selectivity $Selectivity\%$, V the total number of vertices in the dataset and M the mesh degree (average number of neighbors per vertex) we can determine the number of edges inside the query region. Combining these parameters with the cost C_R of accessing one vertex in the adjacency list, Equation 2 defines the total cost as follows:

$$Cost(Crawling) = C_R \times M \times (Selectivity\% \times V) \quad (2)$$

By combining Equation 1 and Equation 2 we obtain a cost model for OCTOPUS:

$$Cost(OCTOPUS) = C_S \times V \left\{ S + \frac{M \times Selectivity\%}{C_S/C_R} \right\} \quad (3)$$

With Equation 3 we can infer how dataset characteristics affect the performance of OCTOPUS. An increase in mesh degree M and selectivity $Selectivity\%$ increases the cost of the crawling (graph traversal) as Equation 2 shows. If the mesh degree M grows, more edges need to be followed, making crawling slower. Similarly, if the selectivity grows, more edges and vertices in the query range need to be visited, also making crawling slower. Estimating the selectivity of spatial range queries has been studied thoroughly in past work. For our analytical model we use the histogram based estimation technique proposed in [2].

To compare the cost of OCTOPUS to our baseline we define the cost of the linear scan as:

$$Cost(LinearScan) = C_S \times V \quad (4)$$

The relative performance of OCTOPUS versus the linear scan, the speedup, can then be calculated by combining Equations 3 and 4 into Equation 5:

$$Speedup = \left\{ S + \frac{M \times Selectivity\%}{C_S/C_R} \right\}^{-1} \quad (5)$$

Clearly, an increasing mesh degree M and an increasing surface-to-volume ratio have an adverse impact on the speedup by slowing down crawling and the surface probe respectively. From Equation 5 we can further infer that the speedup is inversely proportional to the selectivity of the query (assuming the same dataset characteristics). Clearly the linear scan will outperform OCTOPUS for very high query selectivity and the upper limit of selectivity can be calculated, i.e., when $speedup > 1$, by transforming Equation 5 into Equation 6:

$$Selectivity\% < \frac{(1 - S) \times C_S/C_R}{M} \quad (6)$$

Equations 5 and 6 thus help us to decide when to use OCTOPUS given that we know workload characteristics (M and S) and also the runtime constants on the particular hardware used (C_S/C_R).

H. Optimizations

In the previous subsections we focused on strategies to make query execution accurate in face of arbitrary meshes. In the following we discuss optimizations to speed up the query execution with OCTOPUS in general and in cases where we can make assumptions about the simulation application.

1) Graph Data Organization: The graph traversal in the crawling phase is a costly operation because it requires accessing neighboring vertices randomly in main memory. By rearranging the vertices based on spatial proximity we can reduce the number of random reads required on average and thereby improve the L1 and L2 data cache hit rate. We use the Hilbert space filling curve to sort the vertices and organize spatially close vertices, close together in memory. This type of optimization can of course only be used if the simulation application allows to reorder the vertex and edge information in memory.

Dataset Size [GB]	# of Tetrahedrals [Billions]	# of Vertices [Millions]	Mesh Degree [Avg # of edges per vertex]	Surface : Volume [Ratio]
3.2	0.13	20.5	14.5	0.07
4.3	0.17	27.4	14.6	0.06
6.5	0.26	41.1	14.52	0.05
12	0.52	82.7	14.4	0.04
33	1.32	208.1	14.51	0.03

Fig. 4: Neuroscience Dataset Characterization.

2) Surface Approximation: For OCTOPUS we do not make any assumptions about the velocity with which the mesh vertices move between two time steps and therefore probing the entire set of surface vertices is necessary to guarantee the accuracy of the result. If a use case allows to sacrifice accuracy we can further improve performance by taking a sample of equidistant vertices on the surface rather than considering the entire surface set, thereby reducing the time required for the surface probe.

This optimization works well because groups of neighboring mesh elements move similarly throughout the simulation. We show with experiments that little or no accuracy is sacrificed for substantial performance benefits. This optimization is particularly useful in visualization applications where the accuracy of the query retrieval process can be compromised.

V. Experimental Evaluation

In this section we first describe the experimental setup and demonstrate the benefits of OCTOPUS by benchmarking it with microbenchmarks based on neuroscience use cases. We also study the performance of OCTOPUS with a sensitivity analysis and compare OCTOPUS to other approaches.

A. Setup and Methodology

The experiments are run on a Linux Ubuntu 2.6 machine equipped with 2x Intel Xeon Processors with 6 cores running at 2.8GHz, with 64kb L1, 256KB L2 and 12MB L3 cache and 48GB RAM at 1333MHz. The storage consists of 2 SAS disks of 300GB capacity each.

For the measurements we use a neuron simulation that works on a 3D tetrahedral mesh representing two neuron cells. The simulation deforms the mesh model at discrete intervals of time to dynamically adjust the distances between the neuron connections (spine lengths). The details of the datasets used are listed in Figure 4. The largest dataset we use takes 33GB in memory of which 79% define the mesh structure. The mesh is represented using an array of vertices and for each vertex a pointer list representing the edges. The remaining 21% are used for keeping identifiers and attributes of nodes used in the simulation.

We primarily compare OCTOPUS without any optimizations with the linear scan. We further include in the comparison spatio-temporal indexes that allow query execution on the

current state of moving objects and that do not make any assumptions about the movement of objects, i.e., the in-memory implementation of the LUR-Tree [13] and QU-Trade [24] are used. Both approaches base their implementation on the same in-memory R-Tree implementation with a fanout of 110.

Because each vertex in our mesh dataset changes location for every time step we tune QU-Trade for update intensive workloads, increasing the window sufficiently so that fewer than 1% of the location updates trigger the costly R-Tree maintenance process. We also include a lightweight throw-away spatial index [8] Octree and rebuild the index for every time step. The Octree implementation uses a bucket strategy, where a node is split into eight children if it contains more than 10,000 vertices. The configuration parameters, fanout in case of the R-Tree and bucket size for the Octree, have been determined with a parameter sweep, choosing the parameters providing the best performance. All approaches are single-threaded C++ implementations for a fair comparison.

Range queries are executed at each time step after simulation completes updating the mesh. Rebuilding or updating an index during simulation is not possible because the mesh dataset is not in a consistent state. We measure the total query response time, i.e., the time it takes to execute all range queries for all time steps, including the time it takes to rebuild or update the index. Preprocessing steps like building the initial R-Tree for the LUR-Tree as well as the QU-Trade algorithm and surface index in OCTOPUS are done once the mesh is loaded into memory and the time spent is shown separately and not included in the total query response time.

B. Benchmark Evaluation

We compare the performance and memory overhead of OCTOPUS with the other approaches using four microbenchmarks tabulated in Figure 5 based on three neuroscience use cases described in Section III-B.

Figure 6(a) shows performance comparison for each benchmark, using the most detailed neuroscience mesh dataset (33GB) and simulating it for 60 time steps. As shown in the results, the linear scan outperforms spatio-temporal indexing approaches and lightweight throw away Octree index because of the high cost of rebuilding/maintaining. Although 99.5% of the query response time of the Octree is spent for rebuilding the index, it still outperforms spatio-temporal indexes which dedicate 80% (LUR-Tree) and 42% (QU-Trade) to maintain their data structures. LUR-Tree and QU-Trade are slower than the Octree because their query execution is based on the R-Tree which suffers from overlap (due to the high density of the mesh dataset). By using grace windows, QU-Trade spends less time restructuring the R-Tree and thus performs better than the LUR-Tree.

OCTOPUS outperforms all other approaches including the linear scan as shown in Figure 6(a). The speedup relative to the linear scan depends on the selectivity of the queries, i.e., OCTOPUS achieves a maximum speedup of 9.2× for the structural validation use case and a minimum speedup of 7.3× for the visualization (high quality) use case.

Micro-benchmarks	Queries per Time step [#]	Range Volume [μm^3]	Query Selectivity [%]
A) Structural Validation	13 to 17	$2x10^{-5}$	0.11 to 0.16
B) Mesh Quality	7 to 9	$2x10^{-5}$	0.02 to 0.14
C) Visualization (Low Quality)	22	$6x10^{-5}$	0.18
D) Visualization (High Quality)	22	$5x10^{-6}$	0.12

Fig. 5: Neuroscience Benchmarks.

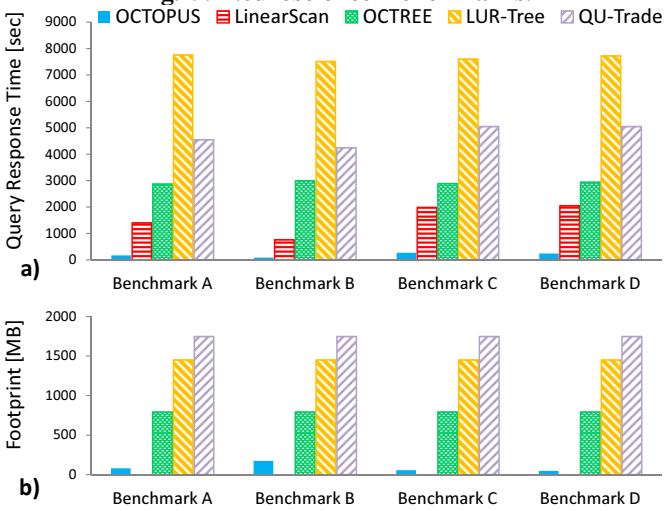


Fig. 6: Performance evaluation (a) and memory overhead evaluation (b) of benchmarks.

The linear scan requires no additional data structures and has thus the least memory overhead. The Octree requires space to keep the tree structure but still requires less space than spatio-temporal indexes LUR-Tree & QU-Trade that are based on an in-memory R-Tree and thus need to store an R-Tree along with a hash index for quick lookups. OCTOPUS uses less memory than all other approaches except the linear scan as shown in Figure 6(b). Its memory footprint mainly depends on query selectivity as we further explain in Section VI-A.

C. Sensitivity Analysis

In this set of experiments we test OCTOPUS and vary particular parameters of datasets and query workload (mesh detail, query selectivity etc.) to analyze its behavior. We also compare the performance of OCTOPUS to the linear scan in order to understand how the trend for the speedup changes.

For the following experiments we use the neuroscience mesh dataset and fix the parameters for each experiment unless mentioned otherwise, i.e., we use the mesh dataset containing 260 million tetrahedra, simulated for 60 time steps. For each time step we execute 15 range queries of selectivity 0.1% located uniform randomly in the mesh.

1) Mesh Detail: Scientists progressively increase the level of detail of the mesh datasets to improve simulation precision. To test how OCTOPUS performs with increasing level of detail, we use meshes of five different levels of detail as defined in Figure 4. Increasing the level of detail and keeping the queries constant, however, increases the number of results

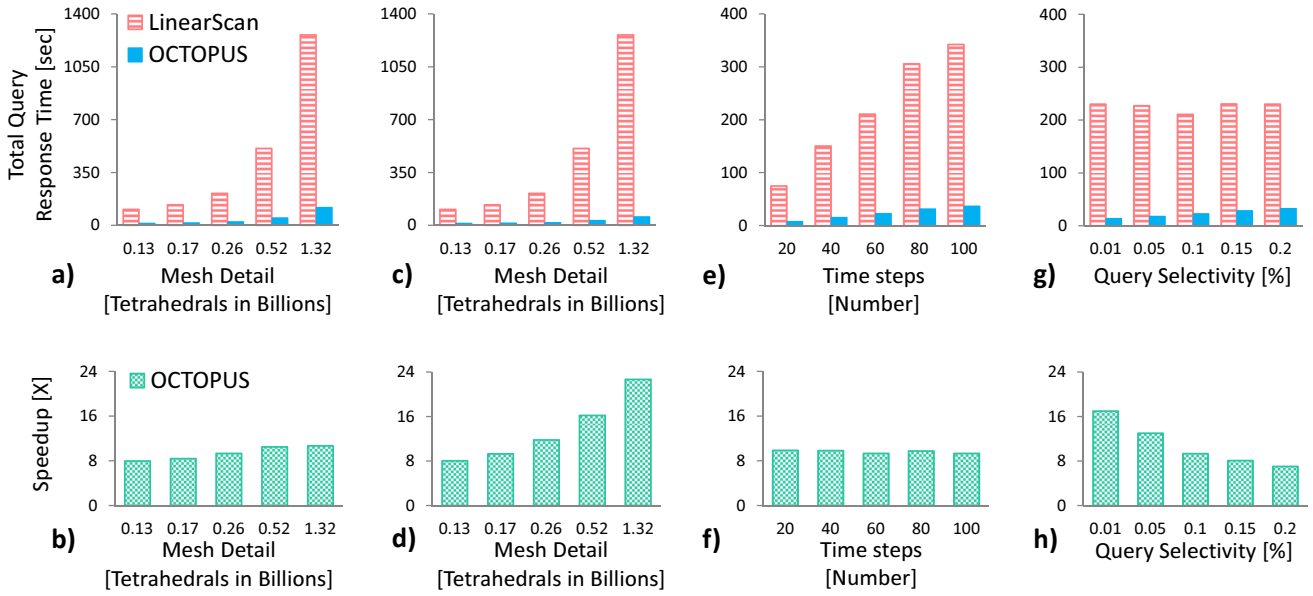


Fig. 7: Sensitivity analysis.

per query while the selectivity of the query remains the same. We thus perform two types of experiments: one where we keep the size of the range queries the same (and hence the number of results increases) and one where we reduce the query size (and thus the number of results remains the same).

First, we execute fixed range queries while increasing the mesh detail. Figure 7(a) shows how the query response time for the linear scan increases proportionally with the size of the dataset. The query response time of OCTOPUS, however, does not increase in proportion to the size of the dataset because as the mesh detail increases, the surface-to-volume ratio of the mesh dataset decreases and hence the time for the surface probe decreases. The relative speedup therefore gradually increases for OCTOPUS from 8 to 10× as shown in Figure 7(b).

Second, we keep the number of results per query fixed by reducing the volume of each range query when executed on more detailed meshes. The linear scan query response time remains the same because its performance solely depends on the dataset size as shown in Figure 7(c). The query response time of OCTOPUS on the other hand is decoupled from the dataset size and the relative speedup of OCTOPUS thus increases considerably from 8 to 23× as Figure 7(d) shows. The increasing speedup has two reasons: (a) the surface-to-volume ratio decreases with increasingly detailed meshes and hence probing the surface meshes becomes comparatively faster and (b) the selectivity of the query decreases if the number of results is kept constant, thereby reducing the share of time needed for crawling.

2) Time Steps: Simulations typically span from tens to thousands of discrete time steps. In this experiment we study the impact of simulating the same deformation of the mesh in an increasing number of time steps (thus decreasing the magnitude of the changes in the entire mesh per time step) on the query execution performance. Figure 7(e) shows how the query response time of both, the linear scan and OCTOPUS, increases linearly when the number of time steps is increased from 20 to 100. Because we execute more time steps with the same number of queries per time step, the total time for each datapoint increases. The relative speedup of OCTOPUS, however, remains constant at 9.5× as shown in Figure 7(f) because similar to the linear scan, the performance of OCTOPUS does not depend on the magnitude of change of the vertices' positions or the number of time steps used in simulations.

3) Query Selectivity: The linear scan does not depend on the selectivity of the range queries as the entire dataset needs to be scanned anyway. Increasing the selectivity, however, means that the share of time OCTOPUS spends during crawling increases. OCTOPUS' graph traversal is costly and when we increase the query selectivity from 0.01% to 0.2% a bigger share of the total time is spent on traversal and the speedup drops from 17 to 7× as Figure 7(h) shows.

D. Convex Mesh Simulations

We test OCTOPUS-CON discussed in Section IV-F on two datasets from the earthquake simulation project using the Archimedes simulator [3] which uses the mesh of the greater Los Angeles basin shown in Figure 1(d). Both meshes have different resolutions (see Figure 8) and are simulated for 60 time steps. 15 uniform random queries are executed per time step each with an average selectivity of 0.1%.

Figure 9(a) indeed shows how the optimized version OCTOPUS-CON outperforms OCTOPUS by eliminating the surface probe and by reducing the time for the directed walk. Because crawling time depends on the query selectivity it is the same for both approaches as shown in Figure 9(b).

The relative speedup compared to the linear scan is 5.7× for OCTOPUS on SF2 and increases to 6.7× on SF1 because SF1 has a smaller surface-to-volume ratio resulting in a faster surface probe. Because OCTOPUS-CON skips the surface probe, it becomes insensitive to the surface-to-volume ratio and its speedup is 15.5× for both datasets.

Earthquake Mesh Dataset	Size [MB]	# of Tetrahedrals [Millions]	# of Vertices [Millions]	Mesh Degree [Avg # of edges per vertex]	Surface : Volume [Ratio]
SF2	64	2.07	0.38	13.3	0.16
SF1	371	13.98	2.46	13.5	0.09

Fig. 8: Earthquake simulation, convex mesh datasets.

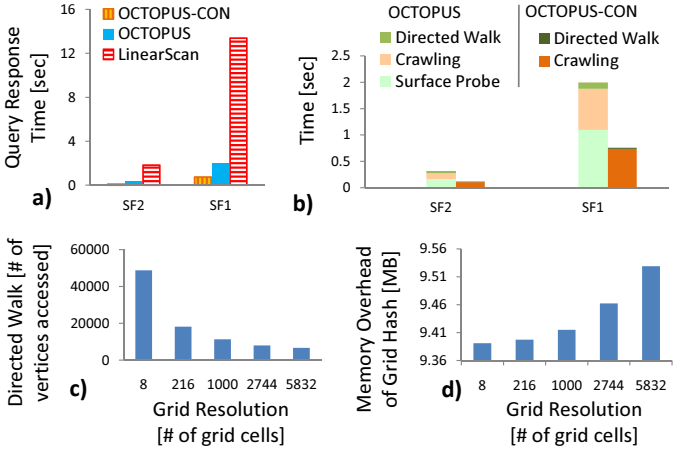


Fig. 9: Convex Datasets.

The 3D grid index used in OCTOPUS-CON reduces the time needed for the directed walk. If we use a coarse resolution grid, the start vertex used for the directed walk may be further away from the query region compared to using a fine resolution grid. This is shown with the experiment in Figure 9(c) where the number of vertices visited during the directed walk decreases as we make the grid resolution finer-grained. The memory required to store the grid index, however, increases with the grid resolution as shown in Figure 9(d).

Choosing the grid resolution is thus a trade-off between space and time and for the experiments shown in Figure 9(a) and (b) we use a 1000 cell grid. Choosing a good grid resolution is difficult in practice but at the same not crucial as it (a) only affects the time required for directed walk which is a small fraction of the total time and (b) even a very coarse resolution grid of 8 cells can reduce the time for directed walk by a factor of 8 compared to using no grid.

VI. OCTOPUS Analysis

In this section we analyze OCTOPUS' performance for a neuroscience simulation. We start by quantifying the cost in terms of time and space and then compare the cost model developed in Section IV-G with the actual performance.

A. Overhead Analysis

In the first experiment we break down the time for query execution and measure the execution time of each of the phases of OCTOPUS. To perform this experiment we execute the same queries on datasets with increasing size. We use 60 time steps of simulation for this experiment.

Figure 10(a) shows that OCTOPUS' query execution is dominated by the surface probe and crawling, while the directed walk barely contributes to the execution time. This is because the directed walk is only used when the query does not enclose any surface vertex - a rare case. The time for performing the surface probe increases with dataset size but not proportionally as larger datasets have proportionally fewer surface vertices. The time for crawling, on the other hand, increases proportionally with the dataset size because by executing the queries of the same size, more vertices and edges of the mesh will be in the result and thus more time is spent in the graph traversal.

Computing the surface vertices index used in OCTOPUS is a one time process. For the most detailed neuroscience dataset of 33 GB size it only takes 62 seconds. The surface index does not need to be updated because the transformations during simulation primarily affect the position of vertices and not the mesh connectivity, leaving the surface the same. Restructuring the mesh during simulation is rarely implemented and did not occur when simulating any of the datasets used for the experiments. OCTOPUS, however, still supports surface changes through insert and delete operations on the surface index as discussed in Section IV-E2.

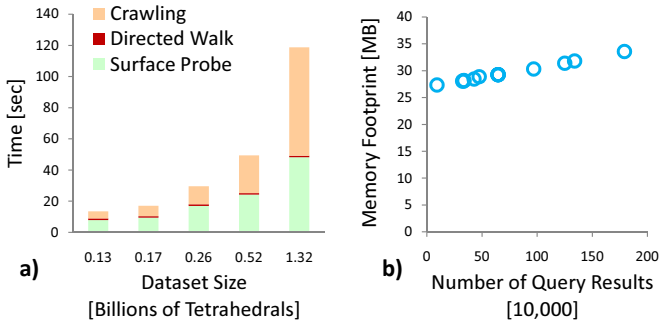


Fig. 10: Performance breakdown (a), footprint (b).

The memory footprint of OCTOPUS includes the surface index (hash table of pointers to surface vertices) and the data structures used during crawling (visited vertices and queues used in the graph traversal). Figure 10(b) shows the direct correlation between the number of query results and memory footprint. For example, for executing 15 queries (leading to 480,000 result vertices) on a 33GB dataset OCTOPUS only uses 1.9MB memory for data structures related to the graph traversal in addition to the 27MB needed for surface index.

B. Analytical Model

We validate the analytical model described in Equation 3 with experiments on five different datasets (parameters S and M are listed in Figure 4 for each dataset). We use 60 time steps for each experiment with 15 randomly placed queries of fixed selectivity of 0.01%, 0.1% and 0.2% per time step. Runtime constants C_S and C_R are determined empirically for the machines used (see hardware configuration) by averaging a long run of a linear scan and graph traversal over the smallest dataset. For our hardware C_R is 2.7×10^{-8} and C_S is 6.6×10^{-9}, C_R is thus approximately 4 times slower than C_S.

As Figure 11 shows, the analytical model provides estimates with little error, showing that OCTOPUS behaves predictably. By using Equation 5 we can also compute the relative speedup of OCTOPUS over the linear scan. For example, using queries

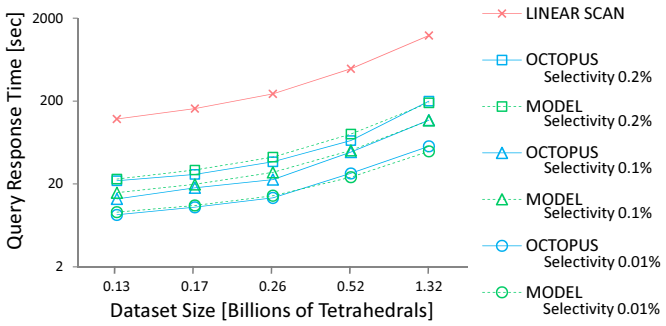


Fig. 11: Validation of the analytical model.

of 0.01% selectivity on a dataset containing 1.32 billion tetrahedra the expected speedup is 11.1, matching the measurements shown in Figure 7(b).

Figure 7(h) shows how the speedup decreases with the increase in selectivity. With increasing selectivity the benefit of OCTOPUS diminishes and we can use Equation 6 to determine at what point a linear scan is faster. For a dataset containing 1.32 billion tetrahedra OCTOPUS performs better if the query selectivity is less than 1.61%. In practice, however, queries with very high selectivity, i.e., more than 1% are rarely executed simply because processing the massive number of results takes too much time. For example, in our visualization based monitoring tool, a query with a selectivity of 0.01% already contains nearly 20'000 results and therefore takes substantial time for rendering.

VII. Impact of Optimizations

In the following we measure the performance improvement and overhead of the optimizations proposed in Section IV-H.

A. Surface Approximation

Probing the entire surface is necessary to guarantee accurate results. Only probing a subset of the surface vertices (uniform random chosen subset), however, gives surprisingly accurate results. The results are accurate because the entire result set can be reached from only a few surface vertices. Figure 12(a) shows that the algorithm can retrieve more than 90% of the result set although 99.9% of the surface vertices are ignored (approximation of 0.1%). For approximations higher than 0.1% the results are accurate as Figure 12(a) shows.

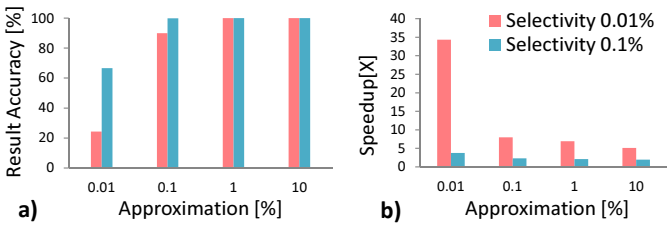


Fig. 12: Effect of Surface Approximation.

The bigger the query (the selectivity), the more likely it is that enough surface vertices are inside the query to retrieve accurate results, as confirmed by the results for the approximation of 0.01% and 0.1%. Figure 12(b) shows the speedup achieved with the approximation when compared to OCTOPUS without this optimization. The speedup is primarily achieved because less time is spent in the surface probe. In case of a very coarse approximation (0.001%), the accuracy drops considerably because the complete result is no longer reachable. Consequently also crawling requires less time resulting in a bigger speedup (at the expense of accuracy).

B. Graph Data Organization

Sorting the vertices of the mesh according to the Hilbert-order as described in Section IV-H1 reduces time for the crawling because storing neighboring vertices close together in memory reduces the number of L1 and L2 cache misses. Sorting, however, has primarily an impact on the crawling whereas the surface probe does not benefit as Figure 13(a) shows. In this experiment we use a dataset containing 1.3 billion tetrahedral objects and execute 900 queries of increasing selectivity. The bigger the result, the more time is spent traversing the graph and hence the bigger is the impact of the optimization. Figure 13(b) makes the impact more explicit by showing the speedup comparing the performance of OCTOPUS with and without Hilbert-sorted data.

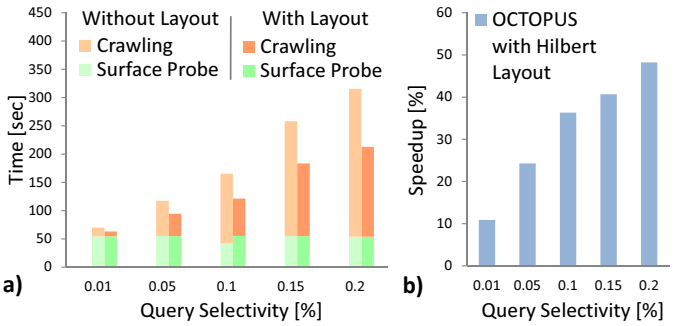


Fig. 13: Effect of hilbert based data layout.

VIII. Applicability

While OCTOPUS is very useful for simulations in scientific disciplines where simulations are based on 3D mesh datasets, e.g., meteorology, structural mechanics, aeronautics and so on, it can also be useful for other applications like rendering part of 3D volumetric models in games and movies.

A. Applicability on Other Datasets

To demonstrate the effectiveness of OCTOPUS on datasets other than scientific simulations, we use three different deforming mesh animation sequences [21] used in 3D volumetric visualization (Figure 14). For each sequence we execute 15 randomly chosen range queries per time step with an average selectivity of 0.1%. Not all of these animation datasets feature the same number of sequences (time steps) and instead of using the total query response time we thus use the average query response time per time step.

In all of the animation sequences OCTOPUS outperforms the linear scan. As expected, the query response time for the linear scan is proportional to the size of the dataset as shown in Figure 15(a). OCTOPUS' performance, on the other hand, depends on the surface-to-volume ratio of the mesh and the best performance is therefore achieved for the facial expression

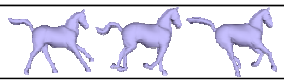
Mesh Deformation		Time steps [#]	Dataset Size [GB]	# of Vertices [Millions]	Surface: Volume [Ratio]
Horse Gallop		48	3.1	20.0	0.023
Facial Expression		9	13	83.6	0.010
Camel Compress		53	6.1	39.8	0.019

Fig. 14: Deforming Mesh Datasets.

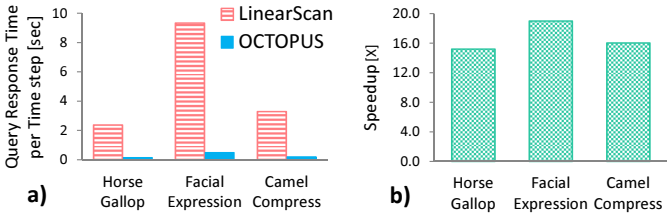


Fig. 15: Query Response Time and Speedups for Deforming mesh datasets.

simulation with a ratio of 0.01 as Figure 15(b) illustrates. Because the datasets used in these experiments have a lower (better) surface-to-volume ratio than the neuroscience meshes, the speedup obtained is even bigger.

B. Limitations

Indexing approaches may be able to amortize the maintenance cost, if the number queries are increased massively (in the order of the dataset size), thereby potentially outperforming OCTOPUS and the linear scan. Furthermore, based on the analytical model, three main factors may limit the performance:

Surface-to-volume Ratio: the worst case is when the mesh consists of only surface vertices i.e., the surface-to-volume ratio ($S = 1$). OCTOPUS query execution thus scans all vertices and degrades to a linear scan. The trend in the simulation sciences, however, is to use ever more detailed volumetric meshes to model the three dimensional structure of the phenomena being simulated, thereby reducing S.

Query Selectivity: as we show with Equation 6 in case of very big queries the linear scan outperforms OCTOPUS. This is rarely the case in practice but Equation 6 can be used to decide what approach to use.

Mesh Degree: The higher the mesh degree, the more edges have to be followed in crawling. In practice the mesh degree is nearly constant for a particular mesh topology, e.g., in case of tetrahedral meshes $M = 14$ [16].

IX. Conclusions

Traditional indexing approaches and the linear scan do not provide good query performance on mesh datasets that undergo massive and unpredictable changes in simulations and will not scale well to bigger and more detail meshes.

We propose OCTOPUS, a query execution strategy that uses the surface of the mesh along with mesh connectivity to retrieve accurate results. With OCTOPUS we achieve a speedup of the query response time between 7.2 and 9.2× in a neuroscience simulation and between 15 and 19× for other simulations compared to the linear scan. More importantly, OCTOPUS is particularly useful for mesh datasets that undergo massive and unpredictable changes during simulations. Finally, as our experiments show, OCTOPUS scales well with increasing mesh detail and is thus ready for substantially more detailed meshes of the future.

X. Acknowledgments

This work is supported by the Hasler Foundation (Smart World Project No. 11031), the Human Brain Project and the US Department of the Navy (Grant N62909-12-1-7010).

References

[1] G. Abdulla et al. Approximate ad-hoc query engine for simulation data. In *Joint Conference on Digital Libraries*, 2001.
[2] S. Acharya, V. Poosala, and S. Ramaswamy. Selectivity estimation in spatial databases. In *SIGMOD*, 1999.
[3] V. Akelik et al. High resolution forward and inverse earthquake modeling on terascale computers. In *Supercomputing*, 2003.
[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
[5] J. Biddiscombe, J. Soumagne, G. Oger, D. Guibert, and J.-G. Piccinali. Parallel computational steering and analysis for hpc applications. In *Eurographics Conference on Parallel Graphics and Visualization*, 2011.
[6] L. Biveinis, S. Saltenis, and C. S. Jensen. Main-memory operation buffering for efficient r-tree update. In *VLDB*, 2007.
[7] L. De Floriani, R. Fellegara, and P. Magillo. Spatial indexing on tetrahedral meshes. In *GIS*, 2010.
[8] J. Dittrich, L. Blunschi, and M. A. V. Salles. Indexing moving objects using short-lived throwaway indexes. In *SSTD*, 2009.
[9] G. Graefe. B-tree indexes for high update rates. *SIGMOD Record*, 35(1):39–44, 2006.
[10] C. L. Jackins and S. L. Tanimoto. Oct-trees and their use in representing three-dimensional objects. *CGIP*, 14(3), 1980.
[11] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *SIGMOD*, 2001.
[12] D. Kwon, S. Lee, W. Choi, and S. Lee. An adaptive hashing technique for indexing moving objects. *Data Knowl. Eng*, 56(3):287–303, 2006.
[13] D. Kwon, S. Lee, and S. Lee. Indexing the current positions of moving objects using the lazy update r-tree. In *Mobile Data Management*, 2002.
[14] H. Markram. The blue brain project. *Nature Reviews Neuroscience*, 7(2):153–160, 2006.
[15] L.-V. Nguyen-Dinh, W. G. Aref, and M. F. Mokbel. Spatio-temporal access methods: Part 2. *IEEE Data Eng. Bull*, 33(2):46–55, 2010.
[16] D. R. OHallaron. Properties of a family of parallel finite element simulations. Technical report, Carnegie Mellon University, 1996.
[17] S. Papadomanolakis, A. Ailamaki, J. C. López, T. Tu, D. R. O'Hallaron, and G. Heber. Efficient query processing on unstructured tetrahedral meshes. In *SIGMOD*, 2006.
[18] J. M. Patel, Y. Chen, and V. P. Chakka. Stripes: An efficient index for predicted trajectories. In *SIGMOD*, 2004.
[19] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, 2000.
[20] Y. N. Silva, X. Xiong, and W. G. Aref. The rum-tree: supporting frequent updates in r-trees using memos. *VLDB Journal*, 18(3):719–738, 2009.
[21] R. W. Sumner and J. Popović. Deformation transfer for triangle meshes. *ACM Transactions on Graphics*, 23(3):399–405, 2004.
[22] T. Szalay et al. Gpu-based interactive visualization of billion point cosmological simulations. *Microsoft eScience Conference*, 2008.
[23] F. Tauheed, T. Heinis, and A. Ailamaki. Efficient query execution on dynamic mesh datasets. Technical Report 190792, EPFL, 2013.
[24] K. Tzoumas, M. L. Yiu, and C. S. Jensen. Workload-aware indexing of continuously moving objects. In *VLDB*, 2009.
[25] X. Xiong, M. F. Mokbel, and W. G. Aref. Lugrid: Update-tolerant grid-based indexing for moving objects. In *MDM*, page 13, 2006.
[26] O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu. *The Finite Element Method: its Basis and Fundamentals*, volume 1. Butterworth-Heinemann, 2005.