Parallel Simulation of Electrophoretic Deposition for Industrial Automotive Applications

Kevin Verma*† Luis Ayuso* Robert Wille†
*ESS Engineering Software Steyr GmbH, Austria
†Institute for Integrated Circuits, Johannes Kepler University Linz, Austria
Email: {kevin.verma, luis.ayuso}@essteyr.com robert.wille@jku.at

***Abstract*—*Electrophoretic Deposition* (EPD) coating is one of the key applications in automotive manufacturing. In the recent years, tools based on *Computational Fluid Dynamics* (CFD) have been utilized to simulate corresponding coating processes. However, the complex data used in this application frequently brings standard CFD applications to its limits. For that purpose, a CFD-based tool named *ALSIM* has been proposed, which employs a unique volumetric decomposition method that addresses these problems. However, certain characteristics of this methodology yield drawbacks for the typical process used in this application – resulting in large execution times. In this work, we present a parallel scheme for this application which addresses these shortcomings. To this end, two layers of parallelism are introduced. Both are implemented by employing OpenMP, allowing for the execution on shared memory parallel architectures. Experimental evaluations confirm the scalability and efficiency of the proposed methods. The simulation time of a typical use case in the automotive industry could be reduced from almost 6 days to 13 hours when employing 16 processing cores.**

Index Terms*—*Electrophoretic Deposition*; *Computational Fluid Dynamics*; *Volumetric Decomposition*; *Parallel Simulation

I. Introduction

The application of coating is one of the key processes in automotive manufacturing. Those coatings are often applied by *Electrophoretic Deposition* (EPD, [1], [2]), in which car assemblies or entire car bodies (also known as *Body in White* or BIW for short) are moved through a tank of liquid. Fig. 1 sketches this process. The object is dipped into the tank by a certain kinematic, while the exact kinematic varies between different manufacturers.

In EPD, it desired that the entire surface of the object is covered by the coating to essentially prevent corrosion. However, the emergence of air bubbles while dipping into the liquid may prevent a complete coverage of the surface. Moreover, entrapped liquid during dipping out may lead to corrosion in the consecutive manufacturing process. For many years, it was therefore practically suited to perform EPD on prototypes. Based on these prototypes, manufacturers were able to asses the problematic areas of the car body and modify it accordingly, e.g. by adding an additional hole to allow entrapped liquid to drain off.

However, prototypes can only be built at a very late development stage – implying that every modification made in this stage requires to roll back to early design stages. This typically causes immense costs and may lead to huge delays in the manufacturing process. Therefore, there is a demand for an efficient and accurate simulation tool, which not only drops the need for a costly prototype, but also allows to detect problematic areas in an early development stage, e.g. during design phase.

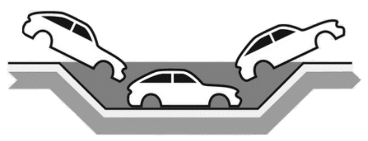


Fig. 1: A simplified Electrophoretic Deposition tank.

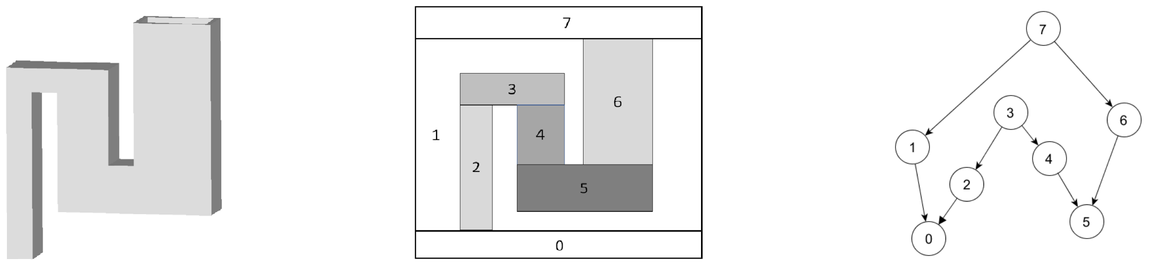
In the recent years, tools based on *Computational Fluid Dynamics* (CFD, [3], [4], [5]) have been utilized to simulate corresponding coating processes. CFD is a well established methodology to simulate scenarios that involve fluid flows. However, the complex data used in this application frequently brings standard CFD methods to its limits – causing extremely large simulation times even on dedicated computer clusters.

To address this problem, the *ALSIM* architecture (from the German "Auslaufsimulation", i.e. drainage simulation) has been proposed [6], [7]. ALSIM is a CFD-based tool, which uses a geometric kernel that employs a unique volumetric decomposition method. This volumetric decomposition allows to use triangular surface meshes, as against CFD, where typically volume meshes are required. In this method, the object gets decomposed into so-called *flow volumes* based on *critical vertices* of the input mesh. As a result, a so-called *reeb graph* accurately represents the topology of the object (this is reviewed in more detail in Section II).

However, this volumetric decomposition has to be applied frequently when the object is rotated. Empirical evaluations yield that this method is by far the most expensive part of the whole simulation in terms of execution time. In an industrial application, this decomposition typically needs to be applied 72 times[1] resulting in large execution times.

In this work, we propose a parallel scheme for this industrial electrophoretic deposition simulation tool, which addresses these shortcomings. The proposed scheme introduces

[1]The 72 applications results from the fact that an object is usually rotated the complete 360°, whereby 5° steps are considered sufficient.



(a) Input Object (b) Flow Volumes (c) Resulting Graph

Fig. 2: Geometric decomposition of a simple object.

two layers of parallelism, which are implemented by employing OpenMP – allowing for the execution on shared memory multi-core architectures. The outer layer enables the parallel construction of the volumetric decomposition, while the inner layer enables parallelism inside the decomposition methodology. Experimental evaluations confirm the efficiency and scalability of the proposed scheme, e.g. by showing that the simulation time of a typical use case in the automotive industry could be reduced from almost 6 days to 13 hours when employing 16 processing cores.

The remainder of this paper is structured as follows: The following section provides an overview of the state-of-the-art and the open challenges. Afterwards, Section III discusses the proposed parallel simulation scheme. Finally, Section IV summarizes the results obtained by experimental evaluations, before the paper is concluded in Section V.

II. State-of-the-art and Open Challenges

In this section, we review the state-of-the-art for simulation of *Electrophoretic Deposition* (EPD, [1], [2]) which is based on methods for *Computation Fluid Dynamics* (CFD, [3], [4], [5]). We particularly review the main characteristic of previously proposed methods which, originally, prevented an efficient simulation and motivates the consideration of a so-called volumetric decomposition scheme. While this scheme addresses a major obstacle for an efficient EPD simulation, it causes other drawbacks which are discussed afterwards. Resolving this drawback and, by this, eventually enabling an efficient EPD simulation is then considered in the remainder of this work.

A. CFD-based Simulation for Electrophoretic Deposition

In electrophoretic deposition processes, objects (e.g. BIWs) get dipped through a tank of paint. In order to simulate such processes, a three-dimensional representation of the object which can serve as input data is necessary. In manufacturing processes, such objects are usually designed using common CAD-tools and, then, exported as meshes which can be used as input for various simulation tools. Fig. 2a provides an example of an object representation used in EPD.

Using such a representation, standard tools based on *Computation Fluid Dynamics* (CFD) can be utilized (and, in fact, have been used for many years) in order to simulate corresponding coating processes. CFD allows the simulation of fluid flows by the numerical solution of the governing Navier-Stokes equations, which have been known for over 150 years.

However, CFD is usually applied to simulate a large number of small volumes like meshes composed of tetrahedra or hexahedra [8], [9]. Simulating large objects such as entire car bodies frequently brings CFD to its limits and, hence, typically requires significantly large computation times (even on dedicated HPC clusters). Besides that, CFD is very sensitive to the choice of boundary conditions. A small difference in boundary conditions may lead to a huge deviation in results.

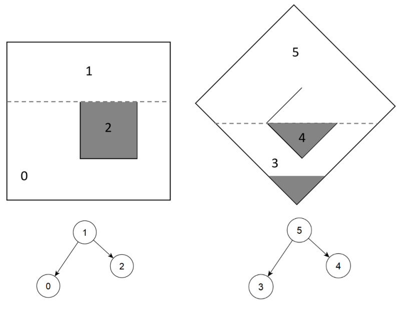
These drawbacks motivate alternative representation of the considered objects which is more suited to the simulation of EPD. For that purpose, the ALSIM architecture has been proposed. This architecture is based on a decomposed volumetric representation whose key ideas are reviewed next.

B. Volumetric Decomposition

One of the main ideas of ALSIM is to use fewer and larger volume units compared to standard CFD methods in order to reduce the computational complexity. For that purpose, the input model is typically a triangular surface mesh, as against CFD where volumetric meshes (consisting of e.g. tetrahedras) are widely used. However, since the main task is to show the fluid distribution inside the volumes of the object, a volumetric representation of the object is inevitable. Therefore, as an alternative volumetric representation, a geometrical decomposition into so-called *flow volumes* has been introduced by Strodthoff et. al. [10]. Here, flow volumes are defined as connected parts of a given triangulated solid, with the boundary consisting of triangles of the triangulated solid and parts of horizontal planes on top and bottom. To generate these flow volumes, the object is scanned for local minimums, maximums, and saddle points (also referred to as *critical vertices*) while sweeping from bottom to top. Each of these identified points ends the former flow volume and starts a new one.

Example 1. *Consider the object representation from Fig. 2a. This is geometrically decomposed by vertical cuts into flow volumes as illustrated in Fig. 2b. Each number denotes one identified flow volume.*

Based on this volumetric decomposition, a graph is constructed which represents the topology of the object by flow



(a) Initial State (b) After Rotation

Fig. 3: Influence of rotation to volumes

volumes with their respective relations. The resulting graphs can be seen as so-called *reeb graphs* [11], which are originally a concept of Morse theory [12], where they are used to gather topological information. This graph representation describes the topology of the object, which is important for the purpose of EPD simulation where it is key to know possible flow paths of liquids.

Example 2. *Fig. 2c shows the resulting graph, while the node numbers correspond to the identified flow volumes shown in Fig. 2b. By means of this graph, it is known that e.g. fluid of volume 3 can flow into volume 2 and volume 4, while fluid of volume 4 can only flow to volume 5.*

This approach provides advantages compared to standard CFD methods, however, it also yields a major disadvantage which is discussed next.

C. Drawback

The volumetric decomposition technique allows for a volumetric representation with significantly reduced complexity. However, the topology of the object is changed as soon as the object is rotated – leading to a different flow volume decompositions. This is a crucial problem, since, as reviewed in Section I, electrophoretic deposition is a dynamic process in which the object is constantly rotated. Due to this inherent dynamic behavior, the volumetric decomposition needs to be applied frequently for various rotation angles. An example illustrates the problem:

Example 3. *Fig. 3a shows an object with a filled cup in the interior before rotating. The graph underneath shows the resulting simple reeb graph. Node 0 represents the volume of the object up to the fill level of the liquid. Node 1 represents the volume above the fill level, while node 2 represents the interior of the cup containing the liquid.*

Fig. 3b shows the object (and the resulting reeb graph) after a 45° rotation. The topology is now completely different, since node 3 represents the volume up to the new filling level of the liquid. The volume of node 5 is significantly larger than the volume of node 1, while node 4 is smaller compared to node 2. Hence, if the objects are rotated, correspondingly adjusted representations have to be created.

TABLE I: Distribution of execution times.

Method	Execution Time Distribution(%)
Setup	5
Create Reeb Graph	80
Hydro-static Solving	5
Hydro-dynamic Solving	10

In the industrial application, a typical process consists of a full 360° rotation of the object through the tank. Since empirical evaluations have shown that time steps of maximum 5° rotation give the most accurate results, a 360° rotation yields 72 discrete time steps. For each of these time steps a new reeb graph needs to be created – obviously resulting in large execution times (especially for complex input data) which poses one of the main problems in EPD simulation thus far. In this work, we are addressing this problem by parallelizing and optimizing this process.

III. Parallel Simulation of Electrophoretic Deposition

As already discussed, the process of EPD simulation inherently is a sequential process. The simulation consists of a set of discrete time steps T, while every discrete time step $0 \leq t < T$ inherently depends on the results of its predecessor. Therefore, the simulation of time step $t-1$ needs to be completed before the simulation of time step t can be started. This dependency prohibits parallelization in the outermost layer, i.e. independently simulating each time step t in parallel is not possible.

However, the simulation of each time step itself offers potential. In fact, four basic tasks have to be conducted for each time step t: (1) primary setup work, (2) creation of the reeb graph, and the actual simulation composed of (3) a hydro-static solving process (rotation) as well as (4) a hydro-dynamic solving process (translation). Table I summarizes the effort needed for average assemblies for each of these steps with respect to their required execution time.

This distribution of efforts is surprising, considering that EPD is essentially performing fluid simulation, where typically the solving part is rendered as the bottleneck (see e.g. [13], [14]). However, these results clearly suggest that the target of any kind of optimization should be the reeb graph construction. The actual simulation (hydro-static/hydro-dynamic solving) consumes only 15% of the total execution time, which shows that the whole process is clearly dominated by the volumetric decomposition. Note that for larges objects (e.g. BIW), the time spent on setup tasks will be less and shifted towards reeb graph construction.

In order to speedup the process of reeb graph construction, two basic layers of parallelization are introduced:

- Independent parallel computation of each graph: Instead of computing the graph sequentially for every time step t and then directly performing the simulation, n graphs are computed in parallel, while n is the amount of cores available on the system. Time step t is then only simulated when the corresponding graph construction is

completed and time step $t-1$ has already been simulated (hereinafter referred to as *outer parallel layer*).
- Parallelization of the reeb graph construction itself: Some of the employed methods for constructing the graph are applicable for data parallelism and are therefore target of a nested parallelism approach (hereinafter referred to as *inner parallel layer*).

The description of the implementation of corresponding techniques is covered by the following subsections.

A. Outer Parallel Layer

For the sake of parallelizing the reeb graph construction, the basic flow of the architecture needs to be re-developed. To this end, we first review the reeb graph construction process as applied thus far and sketched in Algorithm 1. The flow consists of mainly three steps while iterating through all time steps T. The first step is to rotate the input mesh according to the kinematic of the real process (see Line 3). Based on this rotated mesh, a new reeb graph is created (Line 4). Once this reeb graph is constructed, the hydro-static and hydro-dynamic equation systems are solved (Line 5). Afterwards the results of this time step t are available and can be exported for further analysis (Line 6).

Algorithm 1 Original simulation flow

```
1: $M \leftarrow$ input Mesh
2: **for each** time step $t \in T$ **do**
3:     $M_r \leftarrow rotateMesh(M)$
4:     $G_i \leftarrow createGraph(M_r)$
5:     $G_i \leftarrow solveEquationSystem(G_i)$
6:     exportResults($G_i$)
7: **end for**
```

In order to parallelize the reeb graph construction, the base algorithm is re-developed as follows: Instead of iterating through the time steps $t \in T$ and solving the equation system for time step t after creating the graph, n graphs are created in parallel while keeping the equation systems in sequential order. This is sketched in Algorithm 2.

Algorithm 2 Proposed (parallel) simulation flow

```
1: $V \leftarrow$ empty list
2: **for each** $t \in T$ **do**
3:     $S \leftarrow getStepData(t)$
4:     push $S$ onto $V$
5: **end for**
6: **#pragma omp parallel num_threads($n$)**
7: **#pragma omp parallel for ordered schedule(dynamic,
   1)**
8: **for each** $v_i \in V$ **do**
9:     $G_{v_i} \leftarrow createGraph(v_i)$
10:    **#pragma omp ordered**
11:    solveEquationSystem($G_{v_i}$)
12: **end for**
```

Here, the method starts with reading in the step data, which contains the positions and rotation angles of the object (see Line 3). Afterwards, n threads are entrusted with the construction of the reeb graph and the simulation (hydro-static/hydro-dynamic solving) of each time step t (see Line 5-8). As soon as the first graph is completed, the same thread starts with computing the first simulation step (see Line 9-10). Once the second graph has been constructed, the corresponding thread computes the second simulation step and so on. If a thread has completed its simulation step, it will continue with creating more graphs if there are any steps still not simulated. Considering the fact that the graph creation consumes much more computation time than the actual simulation of a step, a thread rarely needs to wait for the previous simulation step to be completed. In-fact, this happens only when the graph construction for time step t consumes significantly less time than the simulation of time step $t-1$.

In general, this method exhibits a rather irregular workload distribution among its iterations. This is because the amount of time needed for the graph construction may differ heavily between the single time steps (depending on the respective rotation angles). Hence, if there are e.g. 8 graphs to be constructed and each thread gets assigned 2, it might happen that the last graph creation terminates earlier than the first one and, thus, the last thread is just busy waiting. Therefore, to receive optimal performance, a dynamic scheduling paradigm is employed (as shown in Line 6 of Algorithm 2). The actual solving of the equation system needs to be kept in the same order as it was executed in serial, since the result of time step t inherently depends on time step $t-1$. This behavior is achieved, by employing the ordered clause. This still allows for a high degree of concurrency, since the graph construction has substantial run time.

Fig. 4 illustrates the complete parallel work flow. Each thread gets assigned one reeb graph and, as soon as the corresponding graph construction is completed, time step t is simulated. Once the simulation of time step t terminated, the reeb graph for another time step is constructed if required. Otherwise the thread terminates.

B. Inner Parallel Layer

The second layer focuses on the parallelization of the reeb graph construction itself. To this end, two basic steps have to be considered:

1) Identifying the critical vertices of the mesh (i.e. its local maxima, minima, and saddle points) and,
2) Using these local information, constructing the volume decomposition by a sweep plane algorithm [15].

Additionally, another step needs to be considered, which was omitted thus far to keep the descriptions simple, namely:

3) Integrating so-called *bottlenecks* into the reeb graph.

The following descriptions provide details to all three steps.

1) Identifying Critical Vertices: The first step to construct the reeb graph is to identify the critical vertices, such as local minima, maxima, and saddle points. The basic algorithm

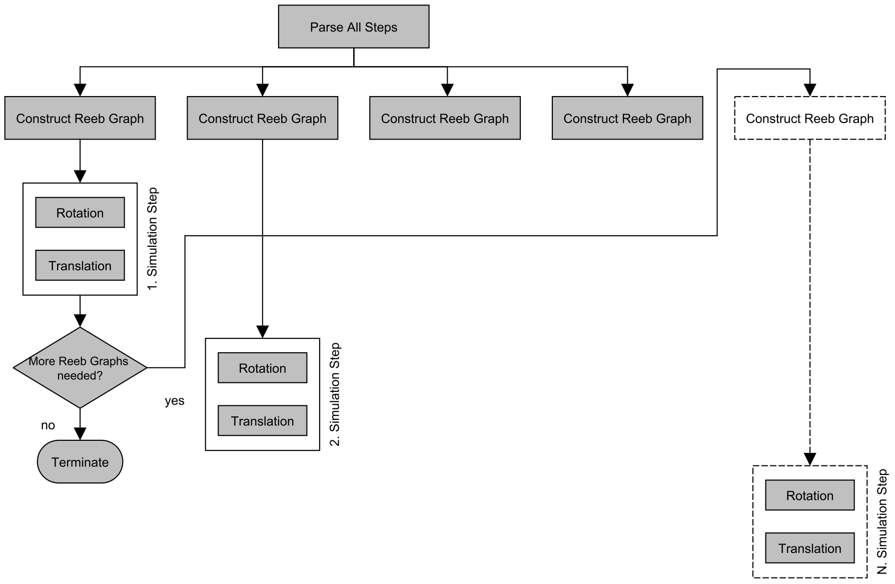


Fig. 4: Illustration of the parallel flow.

iterates through the mesh and creates a new extremum for every identified extrema point. An extrema point can be identified by considering their adjoining points. For example, if adjoining points have lower z-coordinate values, the given point is identified as a local maxima. Algorithm 3 sketches the baseline of the algorithm.

Algorithm 3 Baseline of collecting extrema values

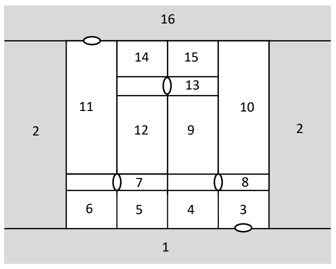
```
1: $EL \leftarrow$ empty list
2: $M \leftarrow mesh$
3: for each vertex $v \in M$ do
4:     if $v$ is critical then
5:         $e \leftarrow$ new extremum induced by $v$
6:         determine region sets and store them in $e$
7:         push $e$ onto $EL$
8:     end if
9: end for
```

Since the mesh is essentially a set of vertices, this algorithm is parallelized by portioning the mesh container. Each thread iterates through its respective portion and generates a new extremum for every critical point.

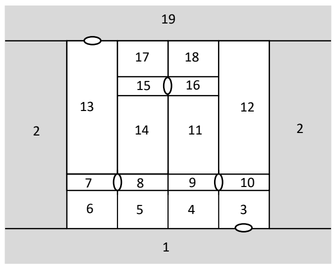
2) Constructing the Volume Decomposition: As stated above, the volume decomposition is conducted using a plane sweep algorithm. In related work, there have already been several attempts of parallelizing plane sweep algorithms. These attempts traditionally employ methods to statically or dynamically segment the input. The operations are then performed in parallel over these segments. However, some of these proposed solutions introduce the need for heavy synchronization, or can only be applied on input arranged as orthogonal line segments [16] [17]. In [18], the plane sweep methodology is avoided to achieve a solution without the need for synchronization. This approach however does not scale well for a smaller number of cores. In [19], the input is dynamically segmented to spatial operations, while performing the operation on multiple portions of the input in parallel without the need for synchronization. The presented methodology splits the input into strips, while the split lines are generated at roughly equal intervals and are parallel to the sweep plane. Each strip is then used as an input to a sweep plane algorithm.

However, these methods for parallelization are not applicable to the problem considered in this work. Here, the sweep plane algorithm constructs the volume decomposition by scanning for the identified critical vertices. Each of such critical vertex triggers an event, which starts a new flow volume and causes the previous flow volume to end. When separating the input into strips, the information about the previous critical vertex and, hence, the start vertex of a new flow volume is lost. In other words, a parallelization would cause significant data dependencies requiring substantial synchronization efforts and, hence, would basically consume all possible benefits gained by a parallel execution. Consequently, we have chosen to keep the sweep plane method sequential and do not suggest any parallelizations for this step.

3) Integrating Bottlenecks: The final step of the graph construction is the integration of so-called bottlenecks. These are needed to consider liquid flows with respect to time. Without the consideration of time, liquids would touch every reachable surface of the object immediately. But of course, liquids needs time to spread, especially when it goes through narrow channels. For that purpose, it is essential to detect all narrow channels of an object in order to limit the liquid flow speed. Such narrow channels are referred to as *bottlenecks*.

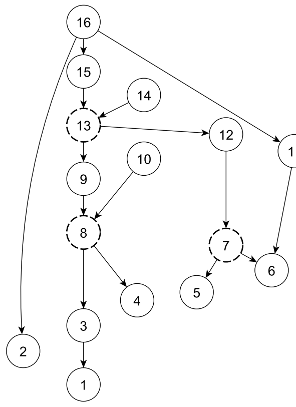


(a) Initial decomposition.

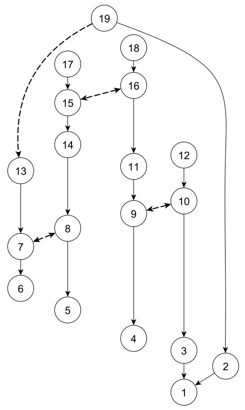


(b) Decomposition after bottleneck integration.

Fig. 5: Integration of bottlenecks into the volumetric decomposition.



(a) Initial reeb graph.



(b) Reeb graph after bottleneck integration.

Fig. 6: Influence of bottleneck integration to the reeb graph.

For the purpose of detecting bottlenecks, which are defined as the shortest loop round a narrow channel, an algorithm can be employed which creates a distance field (using the Dijkstra algorithm) in order to find the shortest path between two nodes in a graph. Starting from one vertex of the mesh, its neighbors are added to the distance field, along with their neighbors, until the given circumference is reached. A bottleneck is found whenever the neighborhood of a newly inserted vertex fulfills a certain criterion.

As discussed in Section II-B, the flow volumes and their relations are represented by the reeb graph. Bottlenecks are found and represented as lists of vertices. For the simulation, their position with respect to the flow volumes is relevant. Therefore, the bottlenecks need to get integrated into the reeb graph.

There exist basically two types of bottlenecks, horizontal and vertical bottlenecks, as illustrated in Fig. 5a (bottlenecks are depicted as small elipses). Vertical bottlenecks can only appear inside of a volume, therefore a split of the volume is indispensable. Horizontal bottlenecks can either appear between two volumes or inside a volume. In case they appear inside a volume, the volume needs to be split, otherwise the respective node connection just needs to be marked as a bottleneck connection. This bottleneck integration significantly influences the flow volumes, as seen in Fig. 5b. The influence to the reeb graph is shown in Fig. 6. Fig. 6a shows the reeb graph before bottleneck integration. The nodes with dashed lines correspond to volumes which need to be split. Fig. 6b shows the reeb graph after splitting. The dashed lines show the newly introduced bottleneck connections, through which the liquid flow is limited.

This whole process of bottleneck integration is especially important, considering that empirical evaluations have shown that for a BIW, typically more than 3000 of such bottlenecks are detected. This causes the amount of flow volumes to increase heavily and takes up to 50% of the whole reeb graph generation run time.

For the purpose of parallelizing this process, the bottleneck integration is split into two basic methods: The volume splitting and the volume connecting. Algorithm 4 sketches the basic algorithm which can be executed in parallel. A map M of volumes V to be split by their associated bottlenecks B is stored (see Line 1-3) . While iterating through the map, the volume v_i is split by its associated bottleneck b (Line 5). The resulting volumes are stored in the list L_{v_s}, which is then integrated into the volume list L_V (Line 6). Since L_V is a global container, which stores all volumes sorted by ascending z-levels, the write to this container needs to be sequential and, hence, locked.

After the volumes are split, they need to be connected to

Algorithm 4 Split volumes by bottlenecks

```
1: $L_V \leftarrow$ list of all volumes $V$
2: $B \leftarrow$ list of all detected bottlenecks
3: $M \leftarrow$ map of all volumes $V$ with associated bottlenecks $B$
4: **for each** element $e \in M$ **do**
5:     $L_{v_s} \leftarrow$ split $v$ of $e$ by associated bottleneck $b$
6:     integrate $v_s$ into $L_V$
7: **end for**
```

their respective upper and lower volumes. This is sketched in Algorithm 5. While iterating through all splitted volumes L_{V_s} (see Line 1-2), a given splitted volume v_s is first connected to the upper volumes and then to its lower volumes (Line 3-4). Since it might happen that the upper or lower volumes of v_s were also splitted, the actual setting of the connection needs to be locked in order to avoid race conditions.

Algorithm 5 Connect splitted volumes

```
1: $L_{V_s} \leftarrow$ list of all splitted volumes $V_s$
2: **for each** splitted volume $v_s \in L_{V_s}$ **do**
3:     connect $v_s$ to upper volumes
4:     connect $v_s$ to lower volumes
5: **end for**
```

Both algorithms can be executed in parallel and, by this, significantly speed up the process. Evaluations summarized in the next section, confirm this improvement.

IV. Experimental Evaluations

In order to evaluate the performance of the proposed parallel scheme, a range of experiments was conducted whose results are summarized in this section. The following subsection shows the speedup obtained for the reeb graph construction alone. Afterwards, the speedup obtained for the entire simulation is presented.

A. Speedup for the Reeb Graph Construction

To evaluate the scalability of the methods in terms of input size and number of execution cores, data sets composed of different numbers of triangles were executed employing a range of up to 16 cores. Table II shows the data sets considered for the experiments. The number of triangles refers to the size of the input triangular surface mesh as introduced in Section I. The considered data sets are typical data sets used in the automotive industry: *Spare wheel case*, *liftgate*, and *cabin* are car assemblies (i.e. car parts), while BIW (*Body In White*) refers to an entire car body.

TABLE II: Considered data sets.

	Spare Wheel Case	Liftgate	Cabin	BIW
# triangles	60k	200k	850k	3M

All experiments have been executed on a two socket Intel Xeon E5-2660 v3. The source code was compiled with GCC 5.4.0 with optimization level -O3 and executed on Ubuntu 16.04. Additionally, a *fill-socket-first* policy was adopted. We are comparing both, the *outer parallel layer* and the additional *inner parallel layer* (as introduced in Section III-A and Section III-B, respectively) with respect to speedup.

Fig. 7a shows the obtained speedup without including the *inner parallel layer*. The values show that for all considered test cases, this basic parallel approach results in a significant speedup. However, for smaller data sets the efficiency drops exceedingly when increasing the number of cores (e.g. for the *spare wheel case* 16 cores yield the same speedup as 8 cores). This is caused by the fact that, for smaller data sets, the time spent on reeb graph construction is proportionally smaller compared to bigger data sets and shifts more towards the actual simulation. Hence, some threads are just busy waiting after completing their respective reeb graph construction until the simulation of the preceding step is completed. For the entire car body (BIW), a speedup of 12.3 was achieved when employing 16 cores.

Fig. 7b shows the speedup obtained when additionally including the *inner parallel layer*. The values show that in almost all the considered test cases, this approach yields an additional speedup compared to using only the basic parallel scheme. Only for the smallest data set, the *spare wheel case*, this approach does not excel, particularly when employing higher number of cores. For the largest data set, the BIW, a speedup of 14.6 was achieved when using 16 cores.

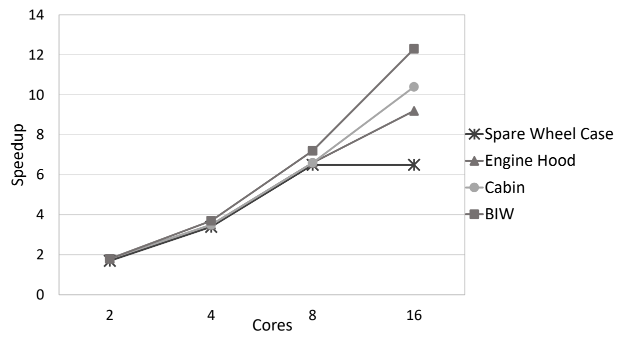
The presented results show that the introduced methods yield significant speedups for all ranges of input sizes when employing a smaller number of cores. For bigger data sets, also a higher number of cores scales well. However, a higher number of cores also introduces an overhead for smaller data sets resulting in less efficiency for the considered smaller test cases.

B. Speedup for the Entire Simulation

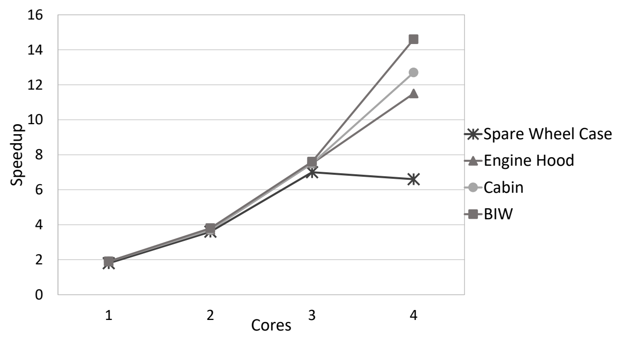
To show the improvements gained for a typical industrial automotive use case, we are also showing the speedup in absolute times of the entire simulation (i.e. all steps listed in Table I) gained for a BIW. The BIW is composed of 2.5 million triangles, the simulation is using 72 discrete time steps of 5° rotation each. Fig. 8 shows the timings in hours received for the BIW, including both, the *inner parallel layer* and the *outer parallel layer*. While a sequential simulation consumes almost 6 days, the parallel simulation employing 16 cores can be conduced within 13 hours.

V. Conclusion and Future Work

In this work, we presented a parallel scheme for a electrophoretic deposition simulation tool for shared memory architectures. Starting from a sequential architecture, potential



(a) Without including the *inner parallel layer*.



(b) With including *inner parallel layer*.

Fig. 7: Speedup of the parallel reeb graph construction scheme.

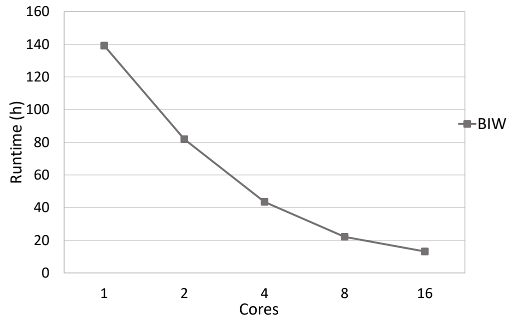


Fig. 8: Speedup of the entire simulation in absolute execution time.

parallelism was unveiled in the process of reeb graph construction. In order to speedup this construction, two parallelization layers where introduced which were implemented in C++ employing OpenMP.

The presented methods yield significant speedups for the reeb graph construction for both, smaller as well as larger data sets. For an entire car body, a speedup of 14.6 was achieved for the reeb graph construction when employing 16 cores. The execution time of the entire simulation could be reduced from almost 6 days to 13 hours for a typical automotive industry use case where a BIW is considered.

Future work includes a parallel implementation for distributed memory architectures, as well as improved scheduling methodologies to achieve further speedups.

Acknowledgment

This work has been supported by the Austrian Research Promotion Agency (FFG) within the project "Industrienahe Dissertationen 2016" under grant no. 860194.

References

[1] L. Besra and M. Liu, "A review on fundamentals and applications of electrophoretic deposition (epd)," *Progress in Materials Science*, vol. 52, no. 1, pp. 1 – 61, 2007.

[2] F. N. Jones, M. E. Nichols, and S. Peter Pappas, "Electrodeposition coatings," in *Organic Coatings: Science and Technology*, 08 2017, pp. 374–384.

[3] H. K. Versteeg and W. Malalasekera, *An introduction to computational fluid dynamics: the finite volume method*. Pearson Education, 2007.

[4] C. Chu, "Computational fluid dynamics," in *Numerical Methods for Partial Differential Equations*, 1979, pp. 149 – 175.

[5] G. Strang and G. J. Fix, *An analysis of the finite element method*. Wellesley-Cambridge Press, 1988.

[6] M. Schifko, S. Xinghua, and K. Kazumasa, "Enhanced dip paint simulation at the very first milestone of car development," *JSAE Annual Congress*, vol. 99, pp. 5–9, 2013.

[7] M. Schifko, H. Steiner, H. Mohri, and C. Bauinger, "Enhanced e-coating - thickness plus gas bubbles, drainage and buoyancy force," *SAE World Congress and Exhibition*, pp. 1–9, 2016.

[8] T. J. Baker, "Mesh adaptation strategies for problems in fluid dynamics," *Finite Elements in Analysis and Design*, vol. 25, no. 3, pp. 243 – 273, 1997.

[9] J. F. Thompson, "Grid generation techniques in computational fluid dynamics," *American Institute of Aeronautics and Astronautics*, vol. 22, no. 11, pp. 1505 – 1523, 1984.

[10] B. Strodthoff, M. Schifko, and B. Juettler, "Horizontal decomposition of triangulated solids for the simulation of dip-coating processes," *Computer-Aided Design*, vol. 43, pp. 1891–1901, 2011.

[11] H. Doraiswamy and V. Natarajanb, "Efficient algorithms for computing Reeb graphs," *Computational Geometry*, vol. 42, pp. 606–616, 2009.

[12] J. Milnor, "Morse theory." *Princeton University Press*, vol. 51, 1963.

[13] K. Verma, K. Szewc, and R. Wille, "Advanced load balancing for SPH simulations on multi-GPU architectures," in *IEEE High Performance Extreme Computing Conference*, 2017, pp. 1–7.

[14] P. Zaspel and M. Griebel, "Massively parallel fluid simulations on Amazon's HPC Cloud," in *First International Symposium on Network Cloud Computing and Applications*, 2011, pp. 73–78.

[15] J. Nievergelt and F. P. Preparata, "Plane-sweep algorithms for intersecting geometric figures," *Communications of the ACM*, vol. 25, 1982.

[16] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter, "External-memory computational geometry," in *IEEE 34th Annual Foundations of Computer Science*, 1993, pp. 714–723.

[17] H. Kriegel, T. Brinkhoff, and R. Schneider, *The combination of spatial access methods and computational geometry in geographic database systems*. Springer Berlin Heidelberg, 1991, pp. 5–21.

[18] M. T. Goodrich, "Intersecting line segments in parallel with an output-sensitive number of processors," in *First Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '89. ACM, 1989, pp. 127–137.

[19] M. McKenney and T. McGuire, "A parallel plane sweep algorithm for multi-core systems," in *International Conference on Advances in Geographic Information Systems*. ACM, 2009, pp. 392–395.