Simulating Industrial Electrophoretic Deposition on Distributed Memory Architectures

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

***Abstract*—The application of coatings by employing *Electrophoretic Deposition* (EPD) is one of the key processes in automotive manufacturing. Here, car assemblies or entire car bodies are dipped into a tank of liquid aimed for preventing the object from future corrosion. However, this process is highly non-trivial. In fact, it has to be ensured that no air bubbles emerge during the dipping which may lead to an incomplete coverage of the coating. Moreover, entrapped liquids that remained after dipping out may lead to corrosion in the consecutive manufacturing process. To detect such problems in an early development stage, simulation methods based on *Computational Fluid Dynamics* (CFD) are utilized. Additionally, employing a dedicated volumetric decomposition method, this has led to a tool chain ALSIM which allows to simulate the process of EPD with significantly reduced complexity as compared to standard CFD tools. However, despite these benefits, the method still suffers from large execution times. In this work, we are proposing a parallel scheme which allows for an execution on distributed parallel memory architectures. To that end, dedicated workload distribution and memory optimization methods are presented, which eventually allow for an efficient simulation of EPD coatings. Experimental evaluations based on industrial use cases confirm the obtained benefits: While a serial simulation required more than 8 days, the parallel method proposed in this work allows to complete the simulation with 32 processes in less than 15 hours.**

I. Introduction

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

The coatings applied by EPD are used to prevent the object from corrosion. However, this is a highly non-trivial task. In fact, to completely prevent corrosion, it is required that the whole surface of the object is completely covered by the coating – which is frequently prevented by air bubbles that may emerge when dipping the object into the tank. Moreover, entrapped liquids that remained after dipping out may lead to corrosion in the consecutive manufacturing process. To detect and avoid these issues, it has been practically suited for many years to perform EPD on prototypes. Based on these prototypes, the manufacturers were able to assess the problematic areas of the object and to modify them accordingly, e.g. by adding an extra hole to drain entrapped liquids.

However, prototypes inherently can only be built at a very late stage of development. This leads to the fact that every change made at this stage requires a rollback to early development stages. By that, not only immense costs are caused, but also manufacturing processes are frequently delayed. Thus, there is a high demand for an efficient and accurate simulation tool that not only drops the need for an expensive prototype but also allows to detect problem areas at an early stage of development.

In recent years, tools based on *Computational Fluid Dynamics* (CFD, [3], [4], [5]) have been used to simulate corresponding coating processes. CFD is a well established method for the simulation of fluid flows. However, the complex data used in EPD often pushes standard CFD methods to their limits – and, even on dedicated computer clusters, cause extremely large simulation times.

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 (this volumetric decomposition is reviewed in more detail in Section II). This 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 different volumetric decompositions. This is a crucial problem, since 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. Moreover, it has been shown that this frequent volumetric decomposition is the most time consuming part of the whole simulation [8], in contrary to other fluid simulation methods, where typically the solving part is rendered as the bottleneck (see e.g. [9], [10], [11]).

To overcome this drawback, in [8] a parallel framework on a threading level has been presented. This framework employs OpenMP to introduce two layers of parallelism and, by that, allows to compute the volumetric decomposition of each discrete time step in parallel. Following this flow, the computational time is reduced significantly. However, this approach still suffers from two major aspects: (1) high memory

Fig. 1: A simplified Electrophoretic Deposition tank used in automotive industry to apply coatings.

consumption as a result of concurrently storing multiple reeb graphs in the memory, and (2) limited parallel computing cores due to hardware limitations of shared memory architectures. To make this parallel approach practically applicable, it is therefore key to enable the execution on distributed memory architectures.

In this work, we are addressing these shortcomings by extending the parallel framework for distributed parallel architectures based on the MPI framework [12], [13]. In order to accomplish that, we consider the following major aspects in this regard: (1) The initial workload distribution of the discrete input rotation degrees, (2) the inherent serial dependency of the concrete time step simulation, (3) the high memory requirement of the employed graph structure, and (4) the irregular workload distributions.

The correspondingly obtained method is described in the remainder of this paper as follows: First, Section II and Section III briefly review the basics on CFD and the volumetric decomposition as well as how to utilize that for EPD simulations, respectively. Afterwards, we describe in Section IV how parallelization of those simulations can be improved by addressing the four aspects listed above – eventually leading to an efficient simulation of EPD on distributed memory architectures. Finally, Section V summarizes the results obtained from experimental evaluations, before the paper is concluded in Section VI.

II. Background

In this section, we review the background 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 developed 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 [14], [15]. 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.

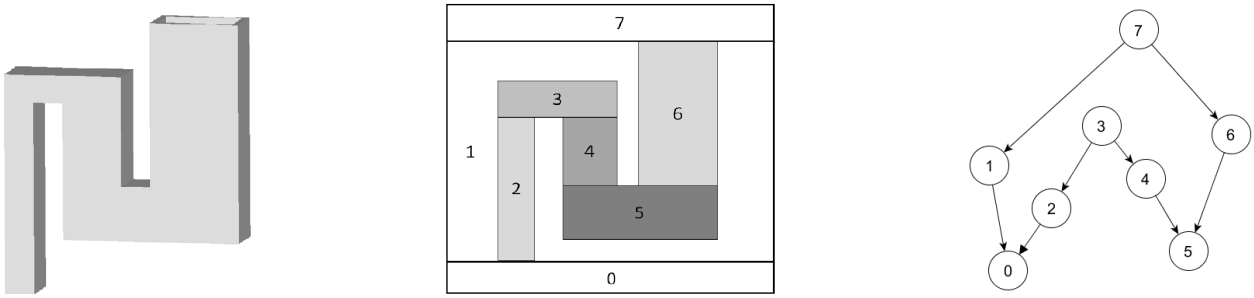
These drawbacks motivate alternative representation of the considered objects which is more suited to the simulation of EPD. For that purpose, the ALSIM architecture has been proposed. This architecture is based on a decomposed volumetric representation whose key ideas are reviewed next.

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. [16]. 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 volumes with their respective relations. The resulting graphs can be seen as so-called *reeb graphs* [17], [18], which are originally a concept of Morse theory [19], 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.



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

Fig. 2: Geometric decomposition of a simple object.

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.*

Based on this reeb graph representation, the actual simulation can be conducted. The simulation consists of two processes: (1) a hydro-static solving process (rotation), and (2) a hydro-dynamic solving process (translation). In this context, the reeb graph is not only used to decompose the input object into flow volumes, but also to store the simulation result. After the hydro-static and hydro-dynamic solving, for each reeb graph node the liquid filling level is stored, which fundamentally represents the simulation result.

This approach provides advantages compared to standard CFD methods, however, it also yields a major disadvantage related to rotation. Every time the rotation angle of the object is changed, the volumetric decomposition is changed and hence the reeb graph needs to be re-computed. This essentially leads to the fact that during every EPD simulation the reeb graph frequently needs to be re-computed which results in increased execution time and memory consumption.

In this work, we are overcoming this drawback by allowing the execution of the EPD simulation on distributed memory architectures. Thereby, the workload of each discrete time step is distributed to the individual processes and then executed in parallel, while keeping the inherent serial dependency of each time step to its predecessor. The basic methodology for this is described next.

III. Simulation of Electrophoretic Deposition

The basic serial simulation flow employed thus far is sketched in Algorithm 1. This 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 actual simulation (hydro-static and hydro-dynamic solving) is conducted (Line 5). Afterwards the results of this time step t are available and can be exported for further analysis (Line 6).

Algorithm 1 Basic simulation flow

1: $M \leftarrow$ input Mesh
2: **for each** time step $t \in T$ **do**
3: $\quad M_r \leftarrow rotateMesh(M)$
4: $\quad G_t \leftarrow createGraph(M_r)$
5: $\quad G_t \leftarrow simulate(G_{t-1}, G_t)$
6: $\quad$ exportResults(G_t)
7: **end for**

In order to allow the execution of the EPD simulation on distributed parallel architectures, this basic simulation flow needs to re-developed.

In the distributed setup, each process is performing the computations of the time steps $t \in T$ independently. For that purpose, each process initially requires two inputs: (1) the original input mesh and (2) the input time step $t \in T$ to be computed.

Therefore, at first, the input time steps are subdivided and distributed to the independent processes. This data contains the rotation angles of the input object of each discrete time step. The rotation will be performed on the original input mesh, which is also distributed to the individual processes.

However, as already discussed, the simulation of time step $t-1$ needs to be completed before the simulation of time step t can be started. Therefore, each process not only requires the original input data, but also the result of the preceding simulation step. Based on this preceding result, the current process can compute its corresponding simulation step.

Overall, for each process this yields the basic workflow as sketched in Algorithm 2. Here, each process is assigned a partition of the input time steps $t \in T$, which contains the positions and rotation angles of the object (see Line 1). Based on this data, each process independently rotates the input mesh and constructs the corresponding reeb graph (Line 4-5). In order to simulate this time step, the process needs to receive the preceding simulation result, based on which the current step can be simulated (Line 6-7). The results are stored inside the reeb graph and dispatched to the subsequent process, as well as exported to disc for further analysis (Line 8-9). This basic workflow is also illustrated in Fig. 3.

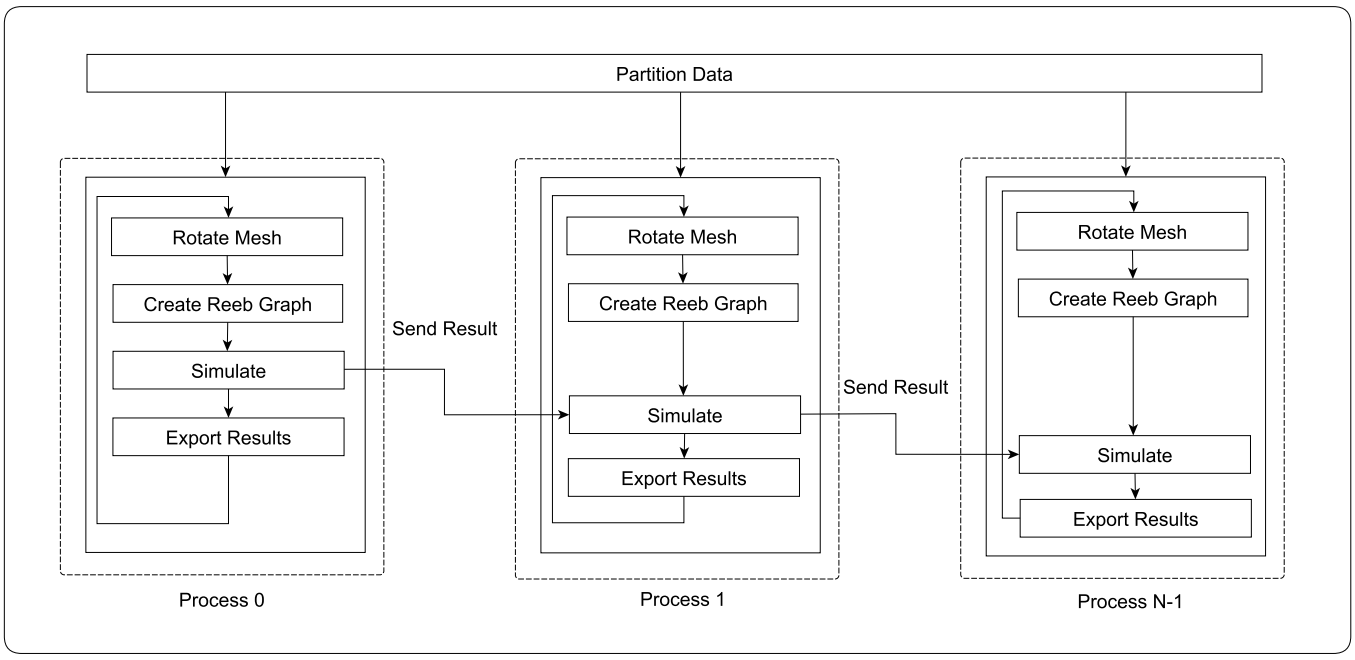


Fig. 3: Illustration of the parallel flow.

Algorithm 2 Proposed distributed simulation flow

```
V ← assigned time steps
M ← input Mesh
for each v ∈ V do
    M_r ← rotateMesh(M)
    G_v ← createGraph(v)
    recv G_{v-1}
    G_v ← simulate(G_{v-1}, G_v)
    send G_v
    exportResults(G_v)
end for
```

However, to receive optimal performance, it is key to consider the important aspects of the implementation, as introduced in Section I. More precisely, at first, the initial workload distribution of the discrete input rotation degrees needs to be considered. Moreover, the inherent serial dependency of the concrete time step simulation and the high memory requirement of the employed graph structure further increase the complexity. Finally, the irregular workload distributions among the concrete time steps needs to be taken into account to receive optimal performance.

The details for this are covered next.

IV. Implementation of the Distributed Algorithm

In order to efficiently implement the proposed workflow, the four aspects as described above need to be considered. To this end, the following section covers the implementation of the workload distribution, as well as a dedicated memory optimization strategy. Furthermore, load balancing issues and limitations of the presented scheme are discussed.

A. Workload Distribution

The general setup of the distributed algorithm is based on a master-slave architecture. Instead of iterating through the time steps $t \in T$ and solving the equation system for time step t after creating the graph (as sketched in Algorithm 1), the first process (N_0) starts by partitioning the input data and distributing it to the remaining processes. Due to the inherent dependency of time step t on $t-1$, an optimal partition should allow a contiguous computation of the time steps $t \in T$. Therefore, time steps t_0 to t_{p-1} should be the first steps to be computed, where p is the total number of processes. Hence, the data for each process N is partitioned as

$$
\begin{aligned}
N_0 &= \{t_0, t_{0+p}, t_{0+2p}, ..., t_{0+(T-p)}\}, \\
N_1 &= \{t_1, t_{1+p}, t_{1+2p}, ..., t_{1+(T-p)}\}, \\
N_{p-1} &= \{t_{p-1}, t_{p-1+p}, t_{p-1+2p}, ..., t_{p-1+(T-p)}\}
\end{aligned}
$$

where p is the total number of processes and T the total number of time steps.

However, in some certain kinematics it might occur that identical rotation degrees are used within one simulation. For example: (1) when the object dips in, an identical rotation degree might occur during the dip out phase, or (2), when a specific rotation degree needs special consideration and is therefore splitted into small time slices to receive more accurate results. In such cases, the reeb graph from the duplicated rotation degree could be re-used and does not need to be constructed again. For that purpose, the identical input rotation angles are assigned to the same process, such that the reeb graph for the duplicated rotation degree does not need to be created again, but can be re-used instead.

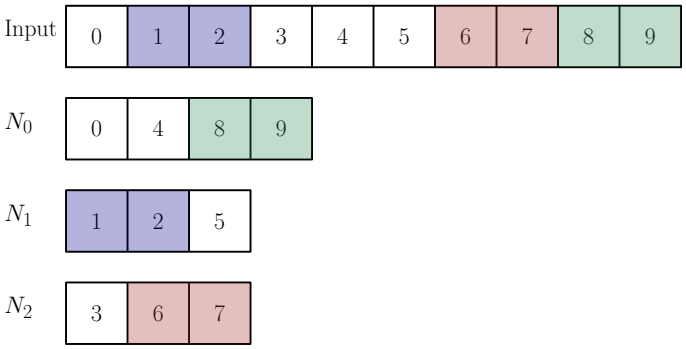


Fig. 4: Partition of the input data to three processes. Colored fields mark identical values.

Example 3. *Fig. 4 shows input data containing rotation degrees for 10 time steps which should exemplarily be partitioned to three processes. Each of the colored fields marks one tuple with identical rotation degrees. During the iteration through the input, the individual rotation degrees are assigned consecutively to the processes. The duplicated rotation degrees are assigned to the same process to allow re-usage of the constructed reeb graph.*

After this initial data partitioning, the first process sends: (1) the original input mesh, and (2) the data partition to the corresponding processes. Asynchronously to this memory transfer, the first process starts the creation of the reeb graph of time step t_0 and the subsequent simulation. Once the remaining processes have received their corresponding input data, each process is computing its rotated mesh, based on which the reeb graph is constructed. As soon as the simulation of time step t_0 is completed by the first process N_0, its result is transfered to the subsequent process N_1. Based on the simulation result of t_0, process N_1 can conduct its corresponding simulation of time step t_1 and once completed, dispatch the result to process N_2. This flow is repeated until all time steps $t \in T$ are simulated.

By employing this communication flow, the actual simulation of the time steps $t \in T$ are kept in the same order as they were executed in serial.

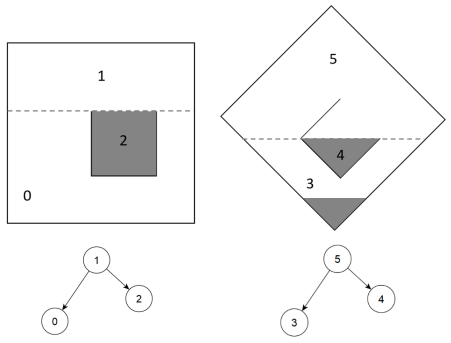
However, as introduced in Section I, not only the inherent serial dependencies, but also the high memory consumption of the reeb graph renders the dispatching of the simulation result to the consecutive processes a non-trivial task. To receive optimal performance, further optimization on the memory consumption is required, as discussed next.

B. Memory Optimization

The high memory requirement of the reeb graph method yields a major drawback in a distributed setup, where the reeb graph needs to be dispatched between the processes. Although the reeb graph method allows for efficient volumetric representation with reduced complexity as compared to volumetric mesh approaches, it suffers from high memory requirements as summarized in Table I. Here, the memory requirements for

TABLE I: Reeb graph memory requirements.

Data set	# Triangles	Memory Requirement
Spare Wheel Case	60k	150MB
Liftgate	200k	450MB
Cabin	850k	1.3GB
BIW	3M	6GB



(a) Initial State (b) After Rotation

Fig. 5: Influence of rotation to the reeb graph.

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. Considering that in a parallel approach using n processes, n reeb graphs will be allocated at the same time, this clearly suggests that the reeb graph construction needs to be distributed among a dedicated cluster. In this distributed setup however, the reeb graphs frequently need to be dispatched between the processes, since the simulation result is incorporated into the graph.

Fig. 5 shows a simple object with the corresponding reeb graph in two discrete time steps with different rotation angles. The simulation result is incorporated into the reeb graph in a way that each reeb graph nodes stores its corresponding liquid filling level. Hence, in Fig. 5a reeb graph node 2 is fully filled, while 0 and 1 are completely empty. In Fig. 5b, reeb graph node 3 and 4 are partially filled and 5 is completely empty. Based on this reeb graph, the simulation result is represented and can also be projected back onto the original input mesh. Therefore, the reeb graph is also employed to transfer the result from one simulation step to another. In Fig. 5 it is also shown that the reeb graph is heavily influenced by the corresponding rotation degree. The volumetric decomposition of Fig. 5b significantly differs from the decomposition before rotation as shown in Fig. 5a. Due to that, the transfer of simulation results from one reeb graph to another is a non-trivial task. In order to transfer the result, the connection from one reeb graph node to the corresponding reeb graph node after rotation needs to be computed. However, as shown in Table I, the reeb graph data structure is extremely heavy in terms of memory requirement. Since dispatching such heavy data structure among a distributed architecture is highly inefficient, a simplified reeb graph has been developed which stores the following data:

1) In order to compute the relation between two rotated reeb graphs, the vertices of the original input mesh are employed. Every reeb graph node represents one volume of the underlying mesh. To compute the relation after rotation, the IDs of the vertices of the underlying mesh are stored.
2) For every vertex of the underlying mesh, every reeb graph node additionally stores a boolean which represents if the corresponding vertex is in touch with liquid or not.
3) For each reeb graph node, the corresponding liquid filling level is stored.

This simplified reeb graph representation has significantly less memory requirement than the standard reeb graph, while the preceding simulation result can still be accurately represented.

While this reduced memory requirement essentially simplifies the communication among the distributed memory architecture, the methods still exhibits a rather irregular workload distribution among their iterations caused by the inherent serial dependency. The details on this are discussed next.

C. Load Balancing

The whole simulation can essentially be abstracted by three major tasks: (1) the reeb graph construction, (2) the actual simulation (hydro-static and hydro-dynamic equation systems), and (3) various setup and I/O tasks. Out of these three major tasks only the reeb graph construction offers potential parallelism, while the other tasks are inherently serial. The ratio between the reeb graph construction and the other tasks, hence the ratio between parallel and serial tasks is roughly 80% to 20%. Fig. 6 illustrates an abstract task view of the whole workflow. The serial part has been summarized by the *Simulation* task. The illustration shows how this inherent serial dependency of time step t to $t-1$ strongly limits the parallel scalability of the whole process. The higher task numbers yield a large idle time between the reeb graph construction and the actual simulation. This strongly suggest that this scheme cannot scale well on a higher number of processing cores. Furthermore, it is also evident that further optimization to the reeb graph construction itself will not result in much higher speedup. Regardless of the magnitude of improvement, the limiting factor is the inherent serial work, as described by Amdahl's Law.

Since this inherent serial dependency cannot be overcome, the only way to reduce this idle time would be to reduce the time of the serial work, hence the actual simulation time. However, in this process a hydro-static and hydro-dynamic equation system is solved which itself offers very limited potential to parallelism. Therefore, the presented scheme of parallelizing the reeb graph construction on a distributed memory architecture is close-to-optimal considering the inherent limiting factors of the application. Evaluations summarized in the next section, confirm these enhancements.

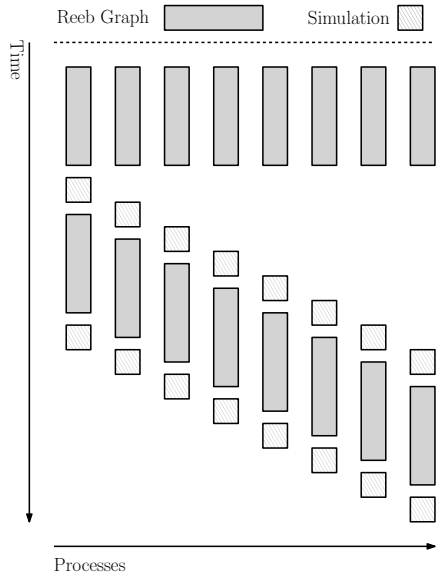


Fig. 6: Abstract processes view of the whole parallel scheme.

TABLE II: Considered data sets.

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

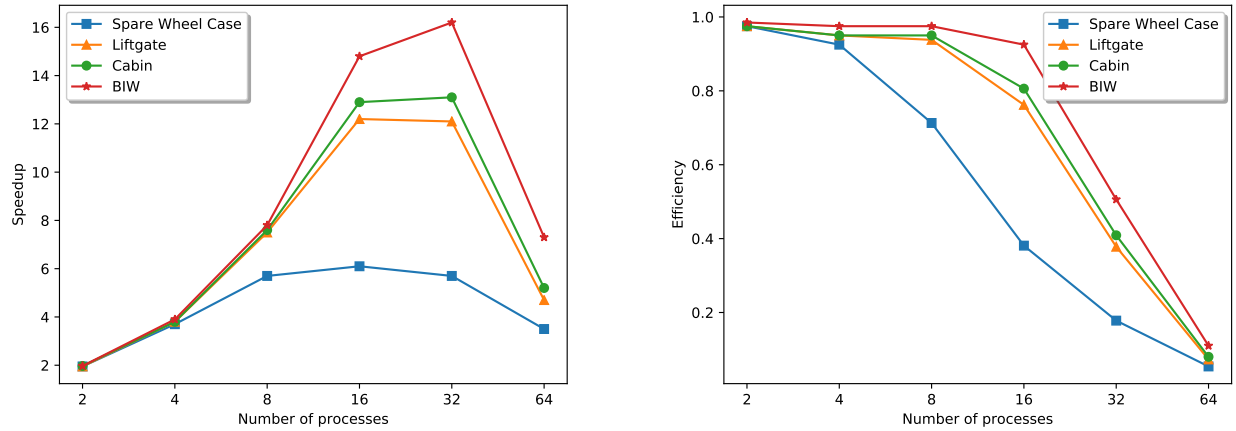
V. Experimental Evaluations

In order to evaluate the performance of the proposed methods, a range of experiments have been conducted whose results are summarized in this section. More precisely, we present the obtained speedup for the reeb graph construction alone (i.e. excluding the actual simulation) as well as for the entire simulation process. Before results on that are presented, however, the utilized test environment and the considered data sets are described first.

A. Test Environment and Considered Data Set

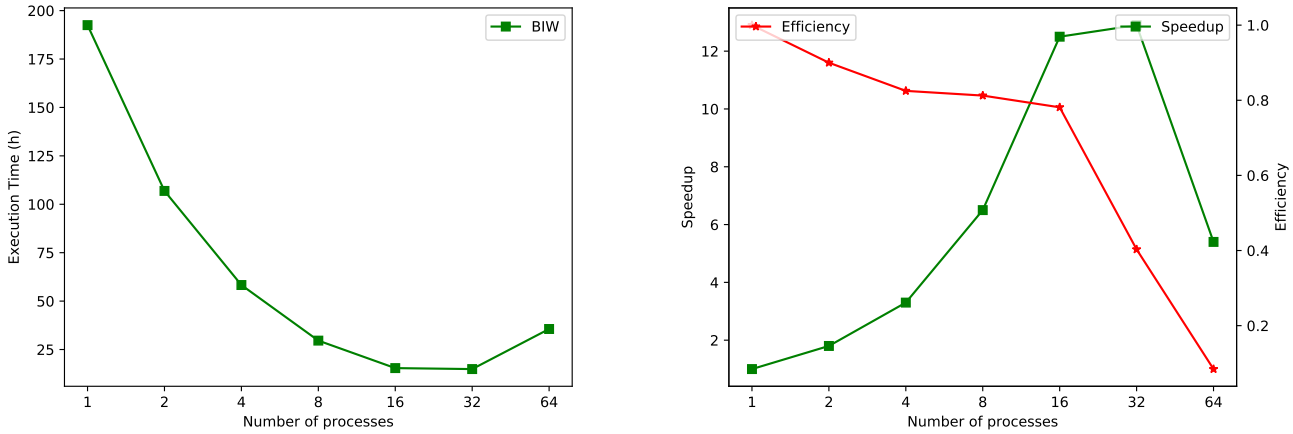
The experiments have been conducted on a cluster containing 8 individual nodes. Each of the nodes contains two Intel Xeon E5-2620 v4 2.10 GHz with 8 physical cores each. The source code was compiled with GCC 5.4.0 with optimization level -O3 and executed on CentOs 7.

To evaluate the scalability of the methods with respect to the input size and number of processes, data sets of different sizes, i.e. composed of different numbers of triangles forming the surface mesh (c.f. Section II), have been considered. All of them constitute typical data sets used in the automotive industry such as a *Spare wheel case*, a *Liftgate*, and a *Cabin* (which all represent parts of a car) as well as an BIW (i.e. a *Body In White*) which represents an entire car body (and, hence, is the largest data set). Table II provides the number of triangles for each of those data sets. Each data set is considered for 72 discrete time steps which yields a simulation of a complete rotation of 360° assuming that simulations steps after each 5° are considered sufficient For each of the data sets, the results of the proposed distributed method are compared with the results obtained by the serial method, i.e. using a single processing core.



(a) Speedup. (b) Efficiency.

Fig. 7: Speedup and efficiency of the reeb graph construction.



(a) Absolute execution time. (b) Speedup and Efficiency.

Fig. 8: Absolute execution time, speedup and efficiency of the whole simulation.

B. Speedup in the Reeb Graph Construction

Table III shows the speedup and efficiency obtained for the four considered data sets using the proposed distributed method compared to the serial method. The results are also visualized in Fig. 7. The obtained values show that the best speedup is achieved when using 32 processes. Then, e.g. for the Cabin, a speedup of 13.1 and, for the BIW, a speedup of 16.2 is obtained. Only for the smallest data set, the spare wheel case, 16 processes yield the best speedup of 6.1. Generally, for the spare wheel case the obtained speedup is significantly lower as compared to larget data sets. This results from the fact that for smaller data sets, the execution time for the reeb graph construction is proportionally smaller as compared to the time spent on synchronization efforts. This generically renders the obtained speedup for smaller data sets underperforming as compared to larger data sets. For the larger data sets it is shown that an efficiency of over 0.9 is obtained when employing up to 8 processes. For the largest data set, the BIW, even for 16 processes an efficiency of 0.93 is achieved.

However, the obtained values also strongly suggest that, for more than 32 processes, the speedup drops drastically. This originates from the fact that in total 72 discrete times steps are considered in the experiments and, hence, 72 reeb graphs need to be computed. When employing 64 processes, 64 reeb graphs will be computed in parallel. After those are completed, only 8 reeb graphs are missing, yielding workload for 8 processes, whereas the remaining processes remain idle. This essentially results in dramatic drop of efficiency, e.g. 0.11 for the BIW when employing 64 processes.

TABLE III: Results obtained for the reeb graph construction.

(a) Speedup

Processes	Spare Wheel Case	Liftgate	Cabin	BIW
2	1.9	1.9	1.9	1.9
4	3.7	3.8	3.8	3.9
8	5.7	7.5	7.6	7.8
16	6.1	12.2	12.9	14.8
32	5.7	12.1	13.1	16.2
64	3.5	4.7	5.2	7.3

(b) Efficiency

Processes	Spare Wheel Case	Liftgate	Cabin	BIW
2	0.98	0.98	0.99	0.99
4	0.93	0.95	0.95	0.98
8	0.71	0.94	0.95	0.98
16	0.38	0.76	0.81	0.93
32	0.18	0.38	0.41	0.51
64	0.05	0.07	0.08	0.11

TABLE IV: Results obtained for the entire simulation.

Processes	Absolute (h)	Speedup	Efficiency
1	192.5	1	1
2	106.9	1.8	0.9
4	58.3	3.3	0.82
8	29.6	6.5	0.81
16	15.4	12.5	0.78
32	14.9	12.9	0.40
64	35.6	5.4	0.08

C. Speedup in the Entire Simulation

To show the improvements gained for a typical industrial automotive use case, we are also presenting the speedup in absolute times of the entire simulation gained for a BIW. The BIW is composed of 3 million triangles, the simulation is considering 72 discrete time steps of 5° rotation each.

Table IV shows the obtained values for absolute execution time hours as well as speedup and efficiency compared to the serial method. The results are also visualized in Fig. 8. The results confirm that the proposed method yields significant improvements. While the serial simulation took 192.5 hours, the simulation method proposed in this work utilizing 32 processes terminated in 14.9 hours, resulting in a speedup of 12.9. When 64 processes are used, the presented approach does not excel due to the inherent serial dependencies of the application. In a simulation using 72 time steps, the 64 processes are frequently just busy waiting after completing their respective reeb graph construction until the simulation of the preceding step is completed. Therefore the efficiency for 64 processes is merely 0.08, as presented in Table IV.

However, for up to 32 processes, significant speedups with satisfying efficiency is obtained. In fact, the time needed to simulate the industrial example considered here can be reduced from over 8 days to just less than 15 hours.

work. The proposed methods result in significant speedup for

VI. Conclusion and Future Work

In this work we presented a distributed parallel scheme for an industrial electrophoretic deposition simulation. Dedicated workload distribution and memory optimization methods were presented an implemented in C++ employing the MPI framework. The proposed methods result in significant speedup for the entire simulation. For an industrial use case simulation using a full car body (Body In White), a serial simulation consumed over 190 hours, whereas the parallel approach could be completed in less than 15 hours using 32 processes. Future work includes the extension of the simulation with new physical phenomena, as well as geometrical optimizations for the reeb graph construction.

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] K. Verma, L. Ayuso, and R. Wille, "Parallel Simulation of Electrophoretic Deposition for Industrial Automotive Applications," in *International Conference on High Performance Computing & Simulation*, 2018, pp. 1–8.

[9] 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.

[10] K. Verma, C. Peng, K. Szewc, and R. Wille, "A Multi-GPU PCISPH Implementation with Efficient Memory Transfers," in *IEEE High Performance Extreme Computing Conference*, 2018, pp. 1–7.

[11] 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.

[12] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA: MIT Press, 1999.

[13] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[14] 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.

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

[16] 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.

[17] T. L. Kunii and Y. Shinagawa, "Constructing a Reeb graph automatically from cross sections," *IEEE Computer Graphics and Applications*, vol. 11, pp. 44–51, 1991.

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

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