

Compression Methods for Simulation Models in SDM Systems

Justus Richter¹, Matthias Büchse², Wolfgang Graf¹, Marko Thiele²,
 Clemens Löbner², and Martin Liebscher²

¹ Institut für Statik und Dynamik der Tragwerke, TU Dresden
²SCALE GmbH

1 Introduction

Models in Simulation Data Management (SDM) systems have grown tremendously in recent years (see Fig. 1). At the same time, these models typically exhibit a great deal of redundancy. This is not being fully exploited by established compression techniques, such as ZIP. In view of the size of state-of-the-art SDM systems, data storage and transfer cause large costs, which makes more advanced compression approaches necessary.

Therefore, we consider two such advanced compression approaches: *Data deduplication* on the one hand exploits the mentioned redundancy, *mesh compression* on the other hand exploits the specific mesh structure.

The first approach, data deduplication, reduces space by removing repetitive data patterns. Every pattern is saved only once, and wherever it reappears, it is replaced by a link to its first occurrence. So far, this approach has largely been applied to backup scenarios, where data is assumed to be immutable and throughput is considerably more important than latency; or it has been applied to large-scale computing with multiple nodes. In the SDM domain, however, we need random access to the data, including deletion, and we usually deal with a single machine, even for the server. Therefore, existing solutions cannot be readily applied.

We implemented a deduplicated storage system and incorporated it into SCALE's SDM solution LoCo, which runs on both Linux and Windows. In the process we solved challenges such as choice of parameters, storage, deletion, data integrity, concurrency, deduplicated transfer, and encryption. We measured

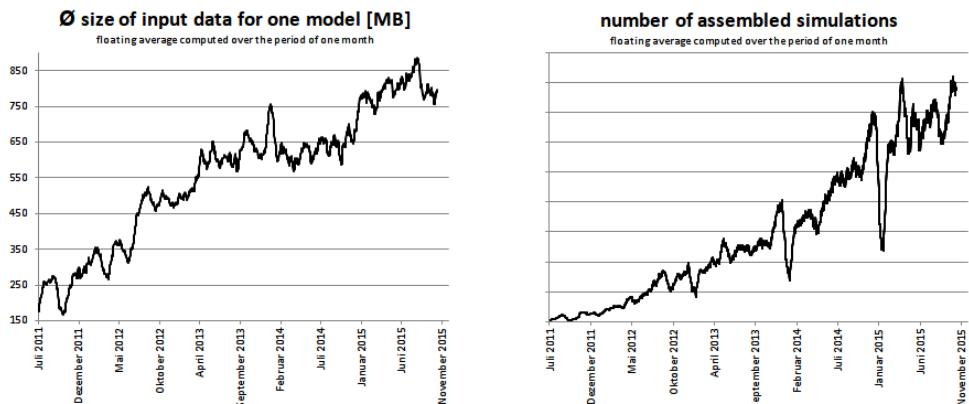


Fig. 1: Model size and assembled simulations (absolute numbers withheld) over time.

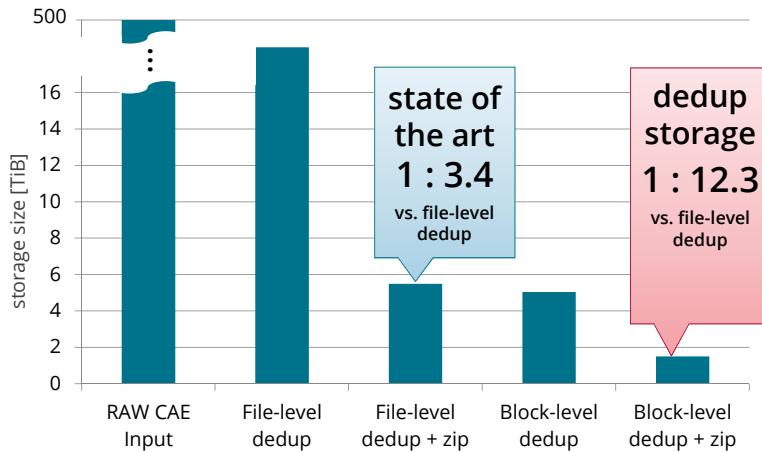


Fig. 2: Compression ratio improvements achieved using data deduplication.

runtime performance and deduplication gain (i.e., space saved compared to non-deduplicated storage) on several datasets. In summary, the runtime performance is completely adequate for an SDM client (around 50 MiB/s write and 150 MiB/s read) and promising for an SDM server. Figure 2 shows the compression ratio improvements of data deduplication in comparison to the state of the art.

The second approach, mesh compression, exploits the known organisation of mesh data containing vertices and connecting elements. We use a *degree encoding* algorithm, which traverses the mesh in a determined manner and stores the element type, and the degree of vertices or edges. Provided this information, it is possible to fully reconstruct the mesh.

In the general traversal process, each step starts by choosing a focus face among a set of incomplete faces, which brings the algorithm to a new solid element to be processed. Focus expansion acknowledges the implicit known faces of the solid element. In the end only relevant data of the solid element is stored in data sequences.

This approach is combined with geometry prediction, which improves the compression by transforming the vertices into suitable coordinates. To this end, the position of each vertex is predicted, based on the already processed part of the mesh and only the offset between original and predicted coordinates is stored.

Beside the presented algorithm for solid element meshes, our implementation contains also support

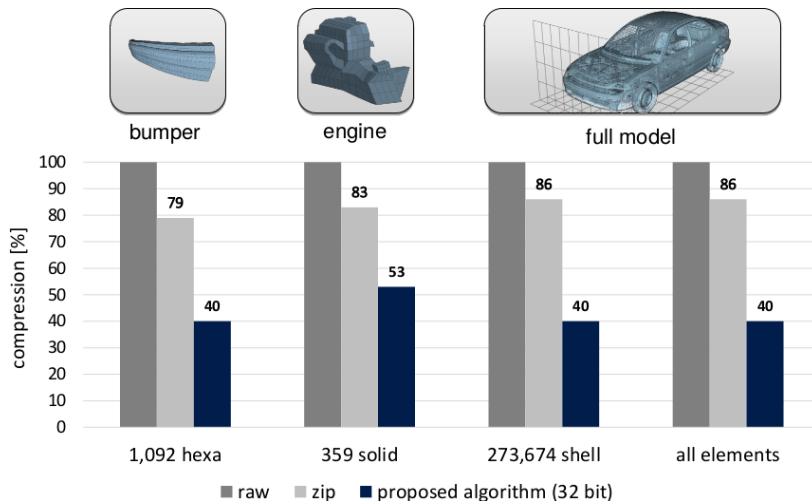


Fig. 3: Lossless compression for mesh data from a Chrysler Neon model (right) and different parts (left and center): raw data (dark gray), zipped data (light gray), and data obtained from the proposed algorithm (blue).

for shell and bar elements, so that all common element types are covered. In first applications quite promising compression ratios around 2.5 could be observed without introducing any loss of information, see Fig. 3. Although this is highly dependent on the regularity of the mesh and can be enhanced much more by introducing quantization.

The following two sections will detail our implementation of data deduplication and mesh compression in the context of SDM systems (Secs. 2 and 3, respectively). The final section will provide a summary and an outlook, in particular, into combining the approaches (Sec. 4).

2 Data deduplication

2.1 Previous work

Paulo and Pereira [11] define “*deduplication*” as a technique for automatically eliminating coarse-grained and unrelated duplicate data.” Deduplication is not new: variants have been described as early as 1996 [16]. By now, many implementations exist, with various application scenarios in mind – cf. Ref. [11] for a comprehensive overview. However, none of these implementations match our requirements.

For instance, most implementations address backup and archival scenarios, where data is assumed to be immutable, and throughput is considerably more important than latency [11, Table 1]. Many implementations come packaged as a product, such as a network-attached storage or a file system. Other systems are targeted at large-scale installations with multiple nodes, or their licence is unsuitable for commercial purposes. In addition, no system covers deduplication of both storage and transfer.

The bottom line is that no off-the-shelf deduplication solution can be integrated into a single-node SDM server (or client) application. However, as we shall see in the following, the techniques and systems described in the literature provide a copious supply of valuable ideas and concepts.

Roughly speaking, the general approach of data deduplication consists of two major steps:

Chunking Partition an input file into *chunks*.

Indexing Look up in an *index* which chunks are already in store.

In a third, but less demanding step, add the new chunks to the store. In many cases, one ought to also store a recipe for reconstructing the file from the chunks.

The deduplication *gain* is the space saved by deduplication divided by the space required by the raw data. This quantity depends on two factors: (a) the number of duplicates found and (b) the storage overhead incurred by the index data structure. For instance, we may find more duplicates by using smaller chunks, but then we need a bigger index (which may also pose problems with respect to latency or throughput).

2.1.1 Chunking

The most simplest of chunking merely partitions the file into fixed-size chunks. The problem with this method is that inserting or deleting bytes in the file will lead to completely new chunks.

A widely used alternative is *content-defined chunking*. Here we need five parameters: the minimum chunk size, the maximum chunk size, the checksum method (such as the one from *rsync* [16] or *Rabin-Karp* [9]), the window size w and the divisor d (both nonnegative integers). We proceed as follows.

We move a “sliding window” of size w over the input data, and at each position we compute the checksum of the window. For the above-mentioned checksum methods, this computation can be done in constant time: we only have to consider the byte that leaves the window and the byte that enters the window. The current right-hand window boundary marks a chunk boundary if (a) the minimum-size requirement is satisfied and the checksum is divisible by d , (b) moving the window further would violate the maximum-size requirement, or (c) we reached the end of the file.

The divisor can be construed as a *target chunk size*, if we assume that the window checksums are more or less evenly distributed.

2.1.2 Indexing

As noted in Ref. [11, Sec. 2.4], systems generally summarize the chunk content via a cryptographic hash (such as SHA-1), and the hash values are used to query or update the index. In its classical form, the *index* then is a mapping that associates, for each chunk in store, its hash value with its storage address. This is also called a *full index*. In contrast, a *partial index* may keep only a subset of the chunks; this saves index space and lookup time, but if a chunk is not present in the index, it may be added to the store a second time, i.e., the deduplication is partial.

A more elaborate approach to partial deduplication is the *sparse index* [10]. Here chunks are grouped together into *segments*, and each segment is represented by a sample of its chunk hashes. The sparse index then is a mapping that associates each sampled hash value with storage information (*manifest*) about the segment that contains the chunk data with that hash value. (A chunk can be referenced in several segments, but the chunk data is stored in exactly one segment.)

For an incoming segment, one samples its chunk hashes in order to find and select matching segments in the sparse index. Before the segment is stored, every chunk that occurs in any of the selected segments is replaced by a reference. We note that this manner of deduplication is partial, for the new segment can still contain a chunk that is present in some segment that did not get selected.

Ghosh [3] proposes a similar technique, but instead of sampling the hashes, he uses a similarity sketch.

2.1.3 Challenges

Parameter values The choice of minimum chunk size, maximum chunk size, checksum method, window size, and divisor is not obvious, and it may depend on the characteristics of the data.

Storage and recipe Chunks should be stored in a way that facilitates reconstructing files; e.g., we might store adjacent chunks in a file in adjacent locations, thereby reducing seeks. Depending on the storage layout, a dedicated recipe may be necessary for reconstructing a file.

Deletion Before a chunk can be deleted, we have to make sure that it is not referenced any more. Common techniques to achieve this are *reference counting* as well as *mark and sweep* [5, Sec. 3.3].

Data integrity The data structure is more complex than a simple collection of files, and a damaged chunk can affect a number of files.

Concurrency Files are no longer independent: the index acts as a synchronization barrier. The index must be protected from concurrent access (in particular, if multiple processes are involved).

Deduplicated transfer We want to transfer only the chunks that the target side lacks, with a reasonable amount of roundtrips (cf. rsync). The deduplicated representations on client and server have to be compatible (support the same abstraction).

Encryption Storer et al. [13] show how to achieve end-to-end encryption for multiple parties via *convergent encryption*. Their setting assumes a full index. It is unclear whether this approach is compatible with, e.g., a sparse index.

parameter	values considered	value at optimal point
checksum method	rsync, Rabin-Karp	Rabin-Karp
minimum chunk size (bytes)	$2^7, 2^8, \dots, 2^{13}$	2^{12}
maximum chunk size (bytes)	2^{20}	2^{20}
window size (bytes)	$2^6, 2^7, \dots, 2^9$	2^6
divisor (target chunk size; bytes)	$2^{13}, 2^{14}, \dots, 2^{23}$	2^{14}

Table 1: Restricted parameter space, optimized values.

2.2 Approach and implementation

Content-defined chunking (Sec. 2.1.1) is employed for its superior deduplication gain. In conjunction with our deletion requirement, this decision has consequences for the index design: with variable-size chunks, overwriting deleted chunks is not a viable option. For defragmentation, chunks must be moved, and so the storage address is variable as well. Therefore, it cannot be used as a long-lived chunk reference. Instead, we reference chunks via their hashes. This, in turn, forces us to use a full index.

When the index becomes prohibitively large, we permanently “freeze” (i.e., mark read only) the current store and open a new one. In that case, our deduplication is partial, because new files are not deduplicated against the frozen stores. This approach is trivial to implement and, provided that related files end up within the same store, leads to reasonable deduplication gain and runtime performance.

2.2.1 Parameter values

In order to explore the parameter space, we considered a dataset of 436 GiB of uncompressed real-world data (mostly simulation models). As objective function we used the function that maps every point in the parameter space to the deduplication gain that is obtained by using the corresponding parameter values to deduplicate our dataset. Our aim was to maximize the objective function.

However, this function is quite expensive to compute. Therefore, we first restricted the parameter space as shown in Table 1. Second, we used a metamodel approach: we computed the objective function on a 100-point random sample of the restricted parameter space and fitted a radial basis function to that sample. Then we maximized the latter function, obtaining the values shown in Table 1. The whole optimization problem was modelled and solved using LS-OPT [12].

2.2.2 Storage and recipe

Our abstraction for storing and retrieving chunks is the *chunk store*. This is a data structure that basically maps chunk hashes (or, more generally, keys) to the respective chunk data, plus a reference count.

In addition to individual chunks, it supports named chunk sequences of bounded size and with arbitrary ancillary data. Each chunk in such a sequence can be accessed individually via its key, but the sequence can also be read or written like a stream. The sequence is preserved on disk, so that seeks are reduced. The sequence size is bounded to facilitate defragmentation. Finally, the chunk store supports transactions for atomically adding or deleting multiple chunks.

When we chunk a file, we obtain a list of chunks. In order to reconstruct the file, we need the list of chunk hashes, or *recipe*. Our abstraction supports three strategies for storing the chunks and the recipe:

Index chunk We can encode any sequence of chunk hashes in binary form and store that representation as a chunk in the chunk store. This kind of chunk we call *index chunk*. We store each chunk individually, and we store the whole recipe as one index chunk.

Hierarchical index Since an index chunk is merely a special chunk, it can refer to other index chunks, which gives rise to a hierarchical index. We factor parts of the recipe out as “subordinate” index chunks; this approach can be likened to a deduplication of index chunks.

Chunk stream We store the incoming *new* chunks in named sequences, commencing a new sequence whenever the size boundary is hit. With each sequence we store (as ancillary data) the corresponding part of the recipe, thus accommodating references to *existing* chunks. We treat the list of sequence names like an index chunk. The recipe is the concatenation of the partial recipes.

In order to implement the chunk store, we created the following concepts and mechanisms:

Flatstore A file of bounded size that is basically a concatenation of chunks, plus checksums. With the exception of reference counts, flatstores can not be updated. Existing flatstores are opened read only; new flatstores can be read from and appended to. Concurrent writes are not supported.

Composite flatstore Combines multiple flatstores to overcome their limitations: it removes the size boundary, it can always be appended to, and it permits concurrent writes. Also, it encapsulates the bulk of our defragmentation procedure (see Sec. 2.2.3).

Journal Provides transactions for atomically manipulating multiple reference counts.

Dictabase The dictabase maps chunk keys to flatstore addresses; currently we mainly use *leveldb* [1].

Dictabase locker Allows selectively locking the dictabase for given keys.

Put bluntly, “chunk store = composite flatstore + journal + dictabase + dictabase locker”.

In order to store a chunk with some key, we lock the dictabase for that key and look up the address. If the lookup succeeds, we merely increase the reference count for the chunk. Otherwise, the data is written to the composite flatstore, and the resulting address is put into the dictabase under said key. Finally, we release the lock on the dictabase. The deletion of chunks is analogous, but the key is not removed from the dictabase; this happens only at defragmentation.

In order to retrieve a chunk, we query the dictabase to obtain the address, and then we read the chunk data from the composite flatstore. Note that there is no need to lock the dictabase for the key.

Technically, a new chunk is first stored at a reference count of zero, and then this count is increased by one. Consequently, the journal allows atomically adding or removing multiple chunks.

2.2.3 Deletion

We use the reference counting facility provided by the chunk store to keep track of the number of times that a chunk occurs in the stored recipes. That is, a chunk has a reference count of zero precisely when there is no recipe that refers to it. Such a chunk is essentially garbage. In order to reclaim the space occupied by garbage, we need to defragment the composite flatstore.

During defragmentation, the storage address of any chunk can change, and that has to be reflected in the dictabase. Moreover, if the defragmentation process is interrupted, we have to make sure that the composite flatstore and dictabase are left in a state that is consistent or easily made consistent.

For this reason, we do not defragment “in place”, but we copy chunks from one flatstore into a new one. Before we start, we flag the old flatstore as “under defragmentation”. The new flatstore is only incorporated into the composite flatstore when it is finished, and only then is the old flatstore deleted. Finally, we update the dictabase.

Any interruption is easily detected: either we find a not-yet-incorporated new flatstore – then we delete it and remove any defragmentation flags we find –, or we delete any flagged flatstore. Interruptions during the dictabase update are not critical: the address of every copied chunk will be carried over the next time the chunk store is opened (see Sec. 2.2.4). Remaining addresses of garbage chunks will be detected easily (upon lookup) because the whole address space will have vanished together with the old flatstore.

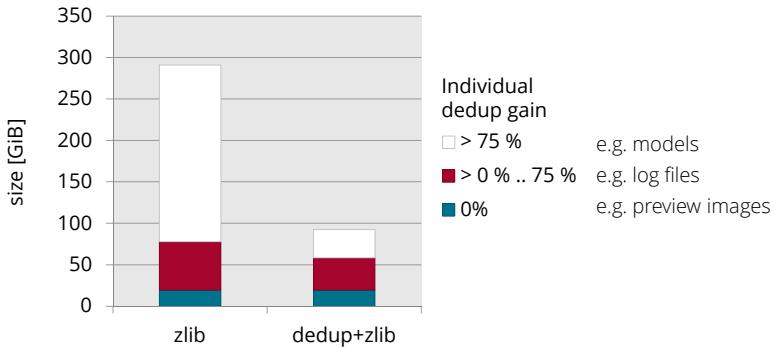


Fig. 4: File classes in deduplicated vault grouped by their individual deduplication gain.

2.2.4 Data integrity

There is a multitude of reasons why data integrity may be at risk: when the disk is full, files may be written partially. Or when the power goes out, write buffers in the operating system do not make it onto the disk. Or the application crashes in the midst of adding a file to the store, before the recipe is complete, leaving behind a number of garbage chunks or even inaccurate reference counts.

Naturally, we want to detect and handle any of these situations. To that end, flatstores include CRC-32 checksums for chunks and CRC-8 checksums for metadata. Any read operation is anticipated to be partial. The composite flatstore detects incomplete defragmentation operations (see Sec. 2.2.3).

Upon opening the chunk store, we perform maintenance: we try to read any chunks that were newly appended since the last “sync”, i.e., when the operating system buffers were forced on disk. If we find an inconsistency, we truncate the respective flatstore. Also, we keep information in the database about the address space it covered at the last sync, and we update the database should it miss any addresses. Finally, we replay the journal transactions since the last sync, rolling back any incomplete transaction.

Each step is idempotent by design; therefore, maintenance can safely be repeated after an interruption.

2.2.5 Encryption

We use the authenticated model described in Ref. [13, Sec. 4.2]. Our main concern has been to protect the *local* storage, and we only support one key pair: that of the (fictitious) local storage owner. For an end-to-end encrypted deduplicated transfer, we would need a globally valid key pair per user.

2.3 Experiments and results

We implemented our deduplicated storage in Python (with the chunking in Cython), and we incorporated it into the SDM client software LoCo [14], which runs on both Linux and Windows. Subjectively, no difference in the performance between the conventional and the deduplicated storage could be noticed.

Intuitively, it is clear that some files are more amenable to deduplication than others. More precisely, if we partition the set of files in a data storage into classes and deduplicate each class on its own, we obtain different deduplication gains. For our analysis, we selected the partition such that two files are in the same class precisely when they have a chunk in common. On a real-world dataset with 260 K files, 292 GiB (zlib compressed), and a total deduplication gain of 75 %, we found 208 K classes.

These classes can be divided into three groups (see Fig. 4): almost 200 K classes consist of only one file each; these make up 19.6 GiB. There are 8 K further classes whose individual gain is at most 75 %; these have 40 K files and 59.8 GiB in total. The remaining 323 classes consist of 20 K files and 213.0 GiB. The three groups have total deduplication gains of 0 %, 35 %, and 93 %, respectively, corresponding to deduplication rates of 1, 1.5, and 14, respectively.

In summary, the deduplication gain that can be achieved very much depends on the data at hand. For instance, the first group contained a lot of preview images and no simulation models whatsoever, while most simulation models were in the third group. Therefore, we surmise that a deduplication rate in the range of 3 to 8 is possible for mixed SDM data; for pure simulation models 12 and more is possible.

3 Mesh compression

In this context meshes are the discretized geometry of the numerically simulated structures, which is described by means of vertices (also called nodes) and elements connecting these vertices, such as polygons or polyhedra.

Thus a mesh description combines two types of information, namely coordinates and connectivity. This subdivision is reproduced by many file formats, e.g., input files for Finite Element Analysis. This is illustrated in Table 2 for an LS-DYNA Keyword file. Here, the coordinate section (second column) defines the spatial position of every vertex. Moreover, a unique vertex ID is explicitly assigned. The connectivity section (third column) groups the given vertices into elements, by defining the element type and referring to the associated vertex IDs.

As coordinates and connectivity represent rather different types of information, we consider their compression independently in the following two sections.

3.1 Compression of coordinates: geometry prediction and quantization

For the compression of vertex coordinates, we consider entropy compression such as ZIP. This leads to the problem that in typical SDM models, mesh coordinate data has a high entropy, which manifests in a strongly fluctuating distribution (black curve in Fig. 5). To solve this problem, we precondition the data in order to produce coordinates of a more predictable distribution (such as the blue curve in Fig. 5). In an optional step, we also quantize the coordinate data into an appropriate format of controllable bit size.

Geometry prediction We use a convenient representation of the vertex coordinates to reduce their entropy. For this purpose a specific coordinate system is chosen for each vertex, putting the origin where the vertex is predicted to be (red dot in Fig. 6), based on a prediction rule which uses already known information. That is, we represent the true vertex position by its offset vector (red arrow) with respect to its predicted position. If the prediction rule is chosen appropriately, the resulting offsets exhibit substantially lower entropy than the original coordinates (see the blue curve in Fig. 5 and resulting offsets in Fig. 9).

Several choices for this prediction rule are possible, based on the considered element types. For purely triangular meshes, Touma and Gotsman [15] introduced the so-called *parallelogram rule* to predict the third vertex of a triangle by expanding an already known adjacent triangle to a parallelogram. This rule gives good predictions if the triangle pair is fairly planar and convex. It is also applicable to predict the fourth vertex of a quadrangle based on the other three vertices. A simple example of the application

mesh	coordinates	connectivity
	<pre>*NODE 1, 0.00, 0.21, 0.00 2, 1.51, 0.09, 0.00 3, 1.39, 1.70, 0.00 4, 2.70, 2.02, 0.00 5, 3.12, 0.31, 0.00</pre>	<pre>*ELEMENT_SHELL 1, 1, 1, 2, 3 2, 1, 2, 3, 4, 5</pre>

Table 2: Decomposition of a mesh into coordinates and connectivity information, illustrated using the LS-DYNA Keywords

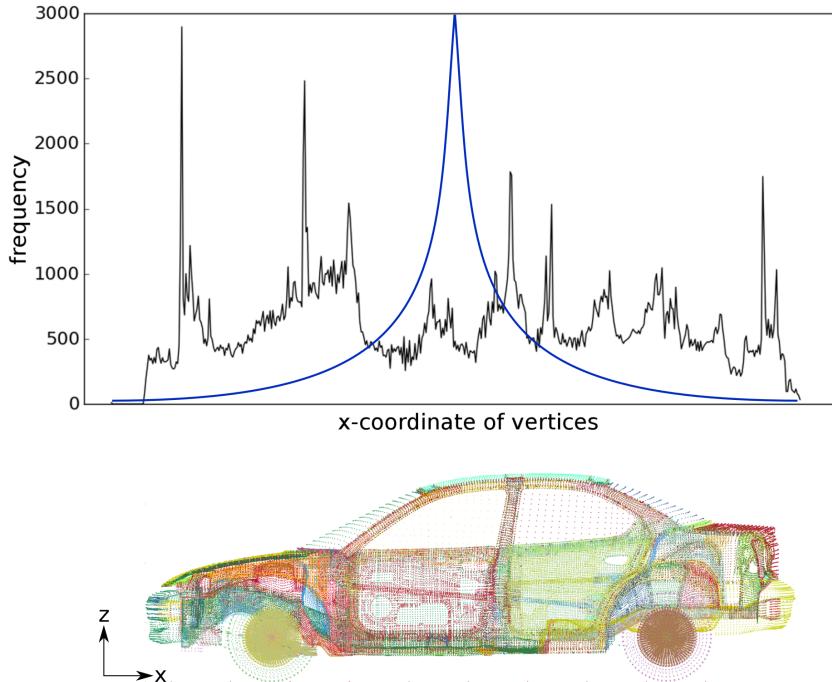


Fig. 5: Frequency graph for x-coordinates of vertices (black curve) and target distribution (blue curve) for a car model (Chrysler Neon)

of that rule is shown in Fig. 6. A generalization of the parallelogram rule to polygonal meshes is given by Isenburg et al. [8], which uses the Fourier representation of polygons. As hexahedra consist of quadrangles, Isenburg and Alliez [6] use the parallelogram rule to predict new vertices in such meshes.

For tetrahedral meshes, it was shown that they “are too irregular to predict vertex coordinates much better than with the proximity information of the connectivity alone.” [4]. The center of a known triangle is therefore used to predict the fourth vertex of the tetrahedron.

Quantization A further improvement of the compression ratio can be achieved by introducing a lossy compression scheme, where the resolution of the coordinates is coarsened by transforming them into integers of bit size n . The quantization q_n of a 3-D vertex point $\mathbf{x} = (x_1, x_2, x_3)$ is given by

$$q_{n,i} : [x_{i,\min}, x_{i,\max}] \rightarrow \{0, 1, \dots, 2^n - 1\}, \quad (i = 1, 2, 3), \quad (1)$$

$$x_i \mapsto \text{int}\left((2^n - 1) \cdot \frac{x_i - x_{i,\min}}{x_{i,\max} - x_{i,\min}} + 0.5\right), \quad (2)$$

where $x_{i,\min}$ and $x_{i,\max}$ are the extremal values of the i -coordinate over the whole dataset. Due to this quantization the algorithm becomes lossy. Equation (2) implies, that the maximal error vanishes expo-

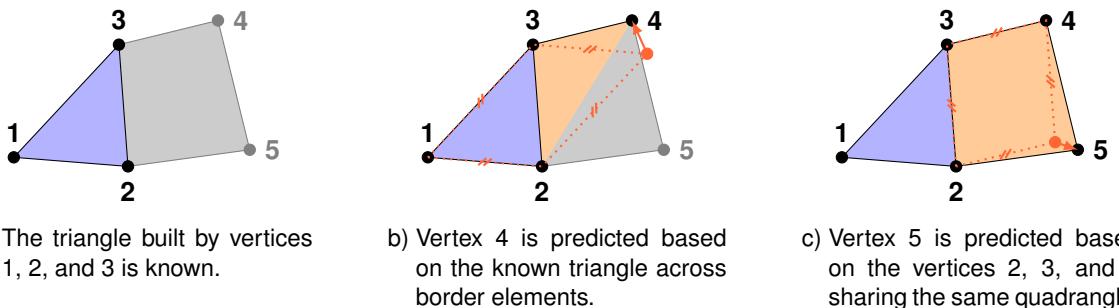


Fig. 6: Prediction for a very simple polygon mesh using the parallelogram rule

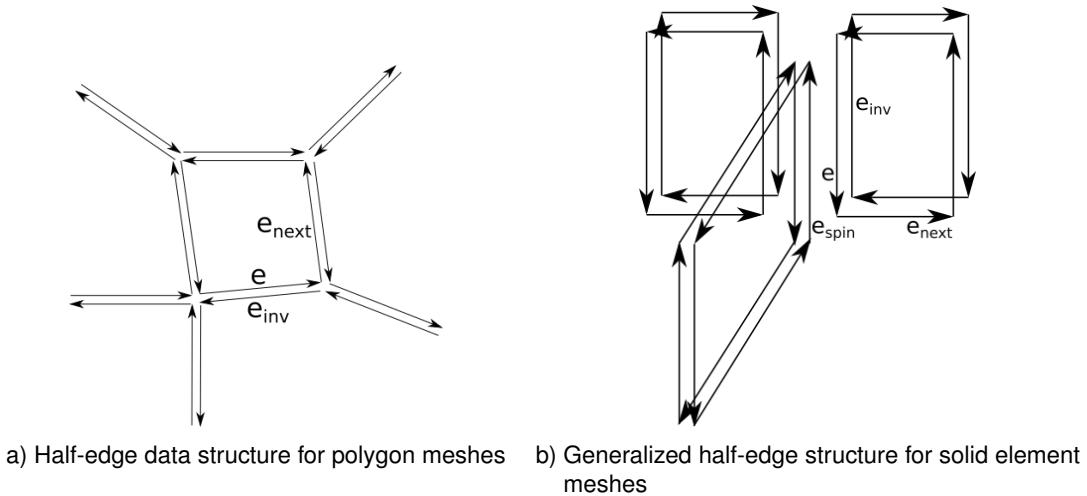


Fig. 7: Mesh data structures

nentially with increasing bit size n ,

$$|q_n(\mathbf{x}) - \mathbf{x}|^2 \leq \frac{1}{2 \cdot (2^n - 1)} \sum_{i=1}^3 (x_{i,\max} - x_{i,\min})^2 \leq \frac{\text{const.}}{2^n}. \quad (3)$$

3.2 Compression of connectivity: degree encoding

3.2.1 Preliminaries

Degree encoding is a traversal algorithm, i.e. its basic idea is to go through the mesh in a predetermined manner. While traversing the mesh, an encoding algorithm defines which local data is relevant to be collected at each step. Given only the sequences of this collected data, the corresponding decoding algorithm can resimulate the traversal, and thereby, reconstruct the original mesh.

Degree encoding determines where to continue the traversal on the basis of the known degree of edges or vertices. The term *degree* is used here in the following meanings.

Edge degree The number of faces adjacent to a given edge.

Vertex degree The number of edges adjacent to a given vertex.

Dependant on the mesh characteristics, different approaches are suitable. Considering a mesh consisting of polygon elements, all edge degrees are 2, so that degree encoding is based on the vertex degree [7]. In Ref. [6] an algorithm is developed to compress hexahedron element meshes using the edge degree.

In the following, (Secs. 3.2.2 and 3.2.3) we present a generalization of the hexaedic mesh compression algorithm [6] which covers also tetrahedra and wedges. In addition, we also implemented a polygon mesh compression algorithm adapted from Ref. [7]. The program is also able to handle bar elements. Because they are assumed to be quite rare, the compression technique is kept simple. There is no prediction; the only sequence the bar encoder writes, contains the quantized coordinates.

In addition we implemented graphical user interfaces, to visualize the state of processing (see Fig. 10).

3.2.2 Mesh data structure

To traverse the mesh efficiently without “searching” for neighbours each time we want to process the next element, the adjacency relations between edges need to be accessible directly. That is why the internal mesh representation is based on the *half-edge data structure*.

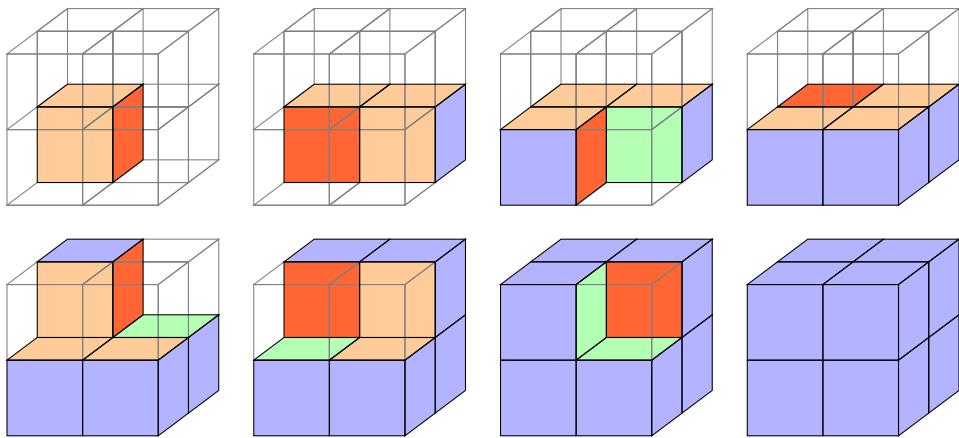


Fig. 8: Traversal steps for a simple cubic mesh, where unprocessed elements are displayed as gray wireframe. At each step we show the current focus face (red), incomplete faces (light red), and border faces (blue). Focus expansion is applied to green faces.

The common half-edge data structure is designed to represent a polygon mesh by vertices and directed half-edges (see Fig. 7a). Each half-edge e is related to its starting vertex, its following half-edge e_{next} and its inverse half-edge e_{inv} .

It is necessary to extend the half-edge data structure, which is done similar to Ref. [6], so that it is capable to represent solid element meshes. By splitting each face into two “half-faces”, half-edge is not correct any more, because an edge usually gets split up more than once. To represent that extension in the data structure the so-called spin-edge e_{spin} is introduced (see Fig. 7b). Furthermore a list is attributed to each vertex containing all edges starting in that vertex. This list is built while traversing the mesh, so that the order of these edges is equal in the process of encoding and decoding.

3.2.3 Degree encoding for solid element meshes

By traversing a mesh, there is an already processed region of the mesh and an unprocessed one. The barrier between these two regions is called *boundary*. The edge or face, where the next element will be added, is referred to as *focus*. Relevant information is mainly the type of newly encountered elements and the position of new vertices. But it may also happen, that vertices of the new element are already known laying on the boundary. In that case it is needed to explicitly describe how the boundary is *joined*. Aiming for data with low entropy, a suitable choice of next focus means to avoid join operations.

The following nomenclature shall be established for meshes consisting of solid elements.

Border face A face, which is adjacent to only one solid element.

Border edge An edge, which is adjacent to at least one border face.

Slot count The number of unprocessed faces adjacent to a given edge.

Zero-slot A face, which is adjacent to a face in the boundary via an edge with slot count of zero.

The general traversal process is illustrated in Fig. 8 for a simple cubic mesh. Each step of the algorithm starts by choosing a focus face (red) among the incomplete faces (light red). This choice is defined by the *traversing strategy*. Then a new solid element is added to the processed region and the corresponding *data representation* of that element is stored in different data sequences.

Traversing strategy Adding one solid element adds in general several new potential focus faces, which means the traversing algorithm has to prioritize them to be processed one after another. The strategy we applied, is given in Ref. [6]. The idea is to prefer faces with many zero-slots, because they are the most complete. That way the creation of cavities, as intermediate state of the space growing process, shall be avoided. If there is no face with zero-slots in the boundary, a face with one or more border edges is chosen.

Each step of processing the mesh will add one further solid element, which is adjacent to the focus face. Before any information is stored explicitly, the implicitly known information is exploited via *focus expansion*. The focus gets extended by boundary faces, which are connected to the focus face through an edge with slot count of zero and by that known to be part of the new element (green in Fig. 8). This way only the data of new faces needs to be described explicitly.

Data representation of traversal The relevant information which encounters during the traversal is consecutively appended to different sequences. In Table 3 the sequences and their content are listed, which we explain in the following.

Apart from the typical entries for element configuration (HEXA, TETRA, WEDGE), the value ROOF is chosen if the opposite face of the focus face in a hexahedron- or wedge-configuration is already known, but none of the connecting faces in between. This cannot be detected by focus expansion and therefore we need to describe explicitly which face of the boundary is the one, we should connect with.

In general there will be vertices in the new element, which are not known yet. Then the operation ADD is executed and the difference between the quantized coordinates and their quantized prediction is stored in the offset sequence. We employ two prediction rules mentioned in Sec. 3.1: For tetrahedra, we use the center of the known face as in Ref. [4], while for hexahedra and wedges, we apply the parallelogram rule as in Ref. [6].

For each new edge the edge degree is appended to the edge degree sequence. Following the rules given in Ref. [6], we derive if the new edge is a border edge, and, if this derivation is not possible, explicitly store this information in border.

If there is an edge, which is known as part of the boundary, but not as part of the currently processed element, JOIN will be written into operation and some further specification into join operation. If one of the vertices of the edge is known to be part of the currently processed element, we append either LOCAL START or LOCAL END to join operation, depending on the position of the known vertex. Otherwise we append GLOBAL to join operation.

To fully describe the join operation, we use the sequences edge slots, vertex-ID, and vertex position. The join operation splits each joined edge into two parts, each having a different slot count. How to split the existing slots into these two parts is written into edge slots. If the entry in join operation was GLOBAL, one of the vertices has to be encoded explicitly, which is done by writing its position in the offset sequence into vertex-ID. Knowing one vertex of the edge, the second vertex is encoded by the edge's position in the edge-list of the known vertex into the sequence vertex position.

When all elements are processed, the sequences get ciphered into binary streams, which are then compressed via the Lempel-Ziv-Markov chain algorithm (LZMA) using its public domain implementation LZMA SDK [2]. These compressed streams will be concatenated and written into a file after a header containing some general information, e.g. the number of elements and the size of the quantization space. When decompressing this procedure is reversed; at first the streams are split and unzipped, then they get interpreted.

sequence	content	item size in bit
element configuration	HEXA, TETRA, WEDGE or ROOF	2
wedge configuration	orientation of wedge, when focus is quadrangle	1
operation	ADD or JOIN	1
offset	offset from the prediction of vertices	$3 \cdot n$
edge degree	edge degree	$\min_{\text{seq}}(\text{degrees})$
border	if edge is border	1
join operation	LOCAL START, LOCAL END or GLOBAL	2
edge slots	where to split slots, when join	$\min_{\text{seq}}(\text{slots})$
vertex-ID	starting vertex, when global join	$\min_{\text{seq}}(\text{IDs})$
vertex position	position of edge in vertex' edge-list	$\min_{\text{seq}}(\text{positions})$

Table 3: Specification of the sequences built during the traversal, $\min_{\text{seq}}(\text{seq}) = \text{int}(\log_2(\max(\text{seq})) + 1)$

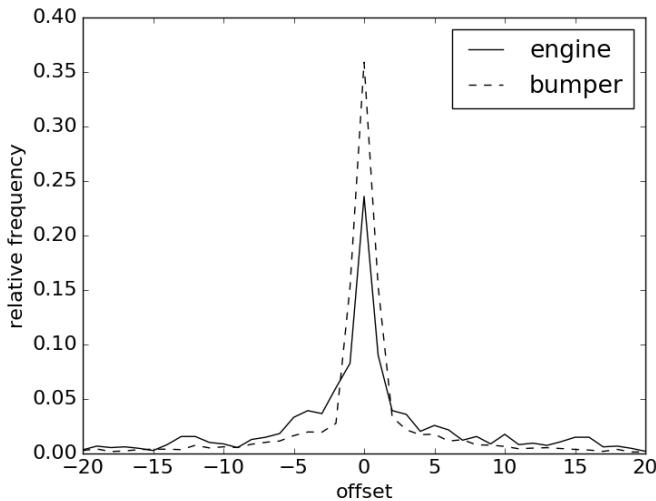


Fig. 9: Normed frequency graph of quantized offsets (12 bit)

3.3 Results

The algorithm is applied to a real world model of a Chrysler Neon, which is publicly available by the National Crash Analysis Center, Washington. We defined the raw data as reference for comparison as follows. The coordinates are quantized with bit size $n = 32$, so uncompressed size equals storage of coordinate as single precision float. The vertex-IDs are integers assumed to fit in 16 bit.

In Fig. 3 the results of lossless compression of the raw data are depicted for different parts of the mesh in comparison with ZIP. The proposed algorithm contains degree encoding in its different specifications for different mesh types as described in Sec. 3.2. While ZIP compresses the model parts to 79–86 % of the raw size, our proposed algorithm achieves around 40 %. This reveals, that ZIP handles the specifics when compressing mesh data far worse than our specialized algorithm.

In Sec. 3.1 we defined a target distribution to enhance the results of entropy compression. For comparison, the offset distribution of two different parts of the model is depicted in Fig. 9. For both parts, the approximate symmetry with a central peak at zero is quite obvious, which verifies the prediction algorithm. Compared to the bumper, the graph of the engine part exhibits additional substructures in the tails and is less symmetric.

More detailed characteristics are presented in Table 4, applying the most common quantitation of compression, the compression ratio, given by

$$\text{CR} = \frac{\text{raw size}}{\text{compressed size}}, \quad (4)$$

separately on vertex coordinate data, connectivity data, and in total.

For ZIP, the compression ratio of coordinate data is independent of the element type, while connectivity data shows some dependency. It is incapable to compress the coordinate data efficiently; nearly all compression is done for connectivity data, which is of course independent of the chosen quantization.

Considering the proposed algorithm, comparing connectivity compression between solid and polygon meshes is not meaningful as different algorithms are used. But coordinate compression is done quite similar, which allows to conclude on the regularity of the element geometry. In this point there is no outstanding difference, except from the engine part.

The maximal error introduced by the quantization with bit size $n = 12$ is $\frac{1}{4096}$ of the model dimension, i.e. about 1 mm for the full model and about 0.25 mm for the engine and bumper part. The compression ratios of the proposed algorithm are increasing exponentially reducing n . Of course there is no infinite increase and compressed data becomes more and more lossy.

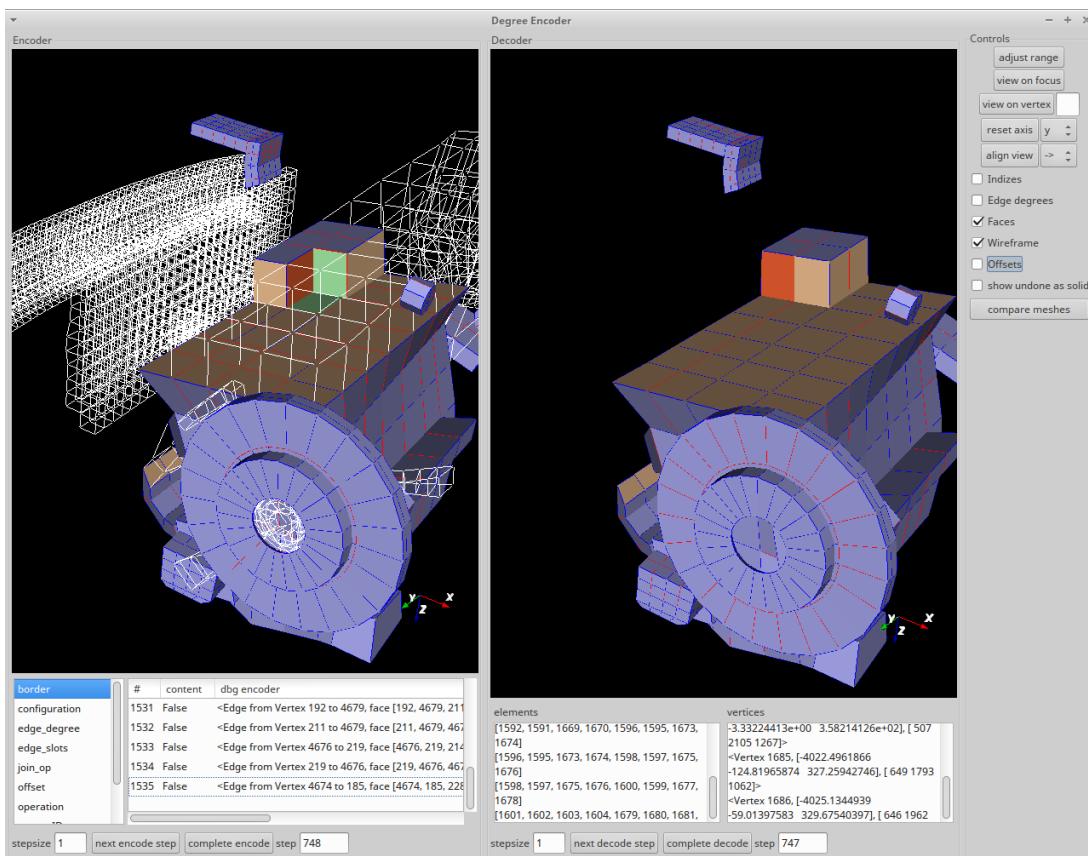


Fig. 10: Graphical user interface showing an intermediate state while processing the solid parts of the model of a Chrysler Neon. In the center is the not fully processed engine part.

		front bumper	engine	all solid parts	all shell parts	all parts
	no. of vertices	1,802	648	5,942	285,770	291,844
	no. of elements	1,092 hexa	359 solid	2,856 solid	273,674 shell	273,674 shell 66 bar
raw 32 bit	coordinate size	21.6	7.8	71.3	3,399.8	3,472.7
	connectivity size	17.5	5.7	45.7	2,149.4	2,195.2
	total size	39.1	13.5	117.0	5,578.6	5,697.3
ZIP 32 bit	CR _{coordinate}	1.03	1.00	1.02	1.02	1.02
	CR _{connectivity}	1.75	1.71	1.70	1.46	1.46
	CR _{total}	1.26	1.20	1.21	1.16	1.16
ZIP 16 bit	CR _{coordinate}	2.03	1.99	2.09	2.11	2.11
	CR _{connectivity}	1.75	1.71	1.70	1.46	1.46
	CR _{total}	1.89	1.84	1.92	1.81	1.81
pro- posed 32 bit	CR _{coordinate}	1.46	1.23	1.59	1.58	1.58
	CR _{connectivity}	30.18	8.61	20.24	39.81	39.03
	CR _{total}	2.53	1.90	2.47	2.52	2.52
pro- posed 16 bit	CR _{coordinate}	3.93	2.93	4.68	5.35	5.34
	CR _{connectivity}	30.18	8.61	20.24	39.81	39.03
	CR _{total}	6.31	3.93	6.53	8.09	8.06
pro- posed 12 bit	CR _{coordinate}	6.37	4.07	7.74	11.60	11.51
	CR _{connectivity}	30.18	8.61	20.24	39.81	39.03
	CR _{total}	9.56	5.02	10.10	16.07	15.89

Table 4: Compression ratio of parts of the Chrysler Neon model, sizes in kB

The front bumper part can be compressed much better than the engine, which is reflected in coordinate and connectivity compression. The relatively poor compression of the engine part is not representative for solid parts of the examined model. As the whole model consists mainly of shell elements, they are dominating the compression characteristics.

There are several reasons that the front bumper has a much better compression ratio than the engine. First, the smaller the mesh, the greater is the influence of the header on the compressed size. The compressed size of an empty solid element mesh is 309 byte. Second, the front bumper is modelled stringently with hexahedra and has no handles. That way the element configuration is always the same and no join operation is needed. The engine part contains also some wedges. Its connectivity is much more irregular and contains a handle, which enforces minimum one join operation. Third, also the geometry is much more irregular for the engine part, whereof follows that the prediction cannot work as fine as for the relatively smooth front bumper (cf. Fig. 9).

Using the degree encoding algorithm the connectivity of polygon meshes can be better compressed, than the one of solid element meshes. Polygon meshes are compressed based on the vertex degree, while solid element meshes employ their edge degrees, but both are compared to the raw data referencing vertex-IDs. Euler's polyhedron formula proves that the number of edges is always greater than the number of vertices, which explains the less efficient solid element approach. But it is also counter-productive to disassemble solid elements into polygons, because that introduces more redundancy into the data.

The discussed dependency of connectivity compression on the mesh type becomes vice versa for ZIP. As the degree of vertices in solid element meshes is higher in average than in polygon meshes, one specific vertex-ID occurs more often in the raw connectivity data. Redundancy like this, is the working point of entropy compression algorithms like ZIP.

Looking at the presented examples, it seems that connectivity compression is more efficient than coordinate compression, which justifies the effort put into compression of connectivity. But one has to keep in mind, that in general the coordinate compression is scaled by choosing an appropriate quantization

bit size n . If a larger loss is acceptable, n will be reduced, whereof follows a better compression of coordinates. If the resolution is critical, n will be raised or quantization will be omitted, i.e. the compression gets worse.

4 Summary and outlook

We considered two advanced compression techniques – data deduplication and mesh compression – and how to apply them to SDM models. In both cases, there is no off-the-shelf solution that can be used in the SDM scenario.

For data deduplication, we solved challenges such as choice of parameters, storage, deletion, data integrity, and encryption. We implemented our procedure in Python and incorporated it into the SDM client LoCo. The runtime performance is completely adequate for an SDM client. We measured the deduplication gain on several datasets. We achieve a deduplication gain of 75 % for mixed SDM data and 87–93 % for pure simulation models (which corresponds to deduplication rates of 4 and 8–14, respectively).

For mesh compression, our implementation is capable to handle all common element types, i.e., hexahedron, tetrahedron, wedge, shell and bar elements, while only the algorithm for meshes consisting of solid elements is described in detail. In first applications using 12 bit quantization quite promising compression ratios around 10 for solid element meshes, and greater for shell meshes, could be observed.

As the software is still in early development stage, it remains a future task to integrate mesh compression in SDM and postprocessing systems. Furthermore compressing element and vertex properties (e.g. stresses and strains) just like coordinates by quantization and prediction would be beneficial.

One can readily apply data deduplication to compressed meshes. However, it is possible that mesh compression decreases the deduplication gain, as small changes in a mesh may result in big changes in its compressed representation. In other words, changes may be amplified. In order to contain this amplification, one might apply mesh compression to logical parts of the model (such as a metal sheet) individually. In fact, the model is already partitioned into connected components during mesh compression. Ultimately, it is yet unclear whether mesh compression disrupts deduplication or not.

Acknowledgements

The work on data deduplication has been developed in the project VAVID (reference number: 01 IS 14005 C), which is partly funded by the German ministry of education and research (BMBF). Matthias Büchse would like to thank Heinrich Kießling for his support with the implementation of a few features and test cases.

The work on mesh compression was supported by the Federal Ministry for Economic Affairs and Energy (BMWi) in the project “W-PostSDM”. The Neon crash model is courtesy of FHWA/NHTSA National Crash Analysis Center.

References

- [1] <http://leveldb.org/>.
- [2] <http://7-zip.org/sdk.html>.
- [3] M. Ghosh. *Scaling Deduplication in Pcompress – Petascale in RAM*. Apr. 2014. URL: <https://moinakg.wordpress.com/2014/04/06/scaling-deduplication-in-pcompress-petascale-in-ram/>.
- [4] S. Gumhold, S. Guthe, and W. Straßer. “Tetrahedral mesh compression with the cut-border machine”. In: *In Proc. Visualization ’99*. 1999, pp. 51–58.

- [5] F. Guo and P. Efstathopoulos. "Building a High-performance Deduplication System". In: *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'11. Portland, Oregon: USENIX Association, 2011, pp. 25–25.
- [6] M. Isenburg and P. Alliez. "Compressing Hexahedral Volume Meshes". In: *GRAPHICAL MODELS*. 2002, pp. 284–293.
- [7] M. Isenburg and P. Alliez. "Compressing Polygon Mesh Geometry with Parallelogram Prediction". In: *IEEE Visualization* (2002), pp. 141–146.
- [8] M. Isenburg et al. "Geometry prediction for high degree polygons". In: *Proceedings of Spring Conference on Computer Graphics*. Budmerice, 2005, pp. 147–152.
- [9] R. M. Karp and M. O. Rabin. "Efficient randomized pattern-matching algorithms". In: *IBM Journal of Research and Development* 31.2 (Mar. 1987), pp. 249–260.
- [10] M. Lillibridge et al. "Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality". In: *Proceedings of the 7th Conference on File and Storage Technologies*. FAST '09. San Francisco: USENIX Association, 2009, pp. 111–123.
- [11] J. Paulo and J. Pereira. "A Survey and Classification of Storage Deduplication Systems". In: *ACM Comput. Surv.* 47.1 (June 2014), 11:1–11:30. ISSN: 0360-0300.
- [12] N. Stander and A. Basudhar. "LS-OPT® Status and Outlook". In: *14th International LS-DYNA Users Conference*. Livermore Software Technology Corporation. Detroit, June 2016.
- [13] M. W. Storer et al. "Secure Data Deduplication". In: *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability*. StorageSS '08. Alexandria, Virginia: ACM, 2008, pp. 1–10. ISBN: 978-1-60558-299-3.
- [14] M. Thiele, T. Landschoff, and A. J. Beck. "LoCo – An Innovative Process and Team Data Management Solution for Simulation". In: *NAFEMS European Simulation Process and Data Management Conference*. 2015.
- [15] C. Touma and C. Gotsman. "Triangle Mesh Compression." In: *Proceedings of Graphics Interface*. San Francisco, 1998, pp. 26–34.
- [16] A. Tridgell and P. Mackerras. *The rsync algorithm*. Tech. rep. TR-CS-96-05. The Australian National University, 1996.