



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Lossless Compression of Hexahedral Meshes

Peter Lindstrom, Martin Isenburg

November 13, 2007

IEEE Data Compression Conference
Snowbird, UT, United States
March 25, 2008 through March 27, 2008

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Lossless Compression of Hexahedral Meshes

Peter Lindstrom

pl@llnl.gov

Martin Isenburg

isenburg@llnl.gov

*Lawrence Livermore National Laboratory**

Abstract

*Many science and engineering applications use high-resolution unstructured hexahedral meshes to model solid 3D shapes for finite element simulations. These simulations frequently dump the mesh and associated fields to disk for subsequent analysis, which involves the transfer of huge volumes of data. To reduce requirements on disk space and bandwidth, we propose efficient schemes for lossless online compression of hexahedral mesh geometry and connectivity. Our approach is to use hash-based value predictors to transform the mesh connectivity list into a more compact byte-aligned stream of symbols that can then be efficiently compressed using conventional text compressors such as *gzip*. Our scheme is memory efficient, fast, and simple to implement, and yields 1–3 orders of magnitude reduction on a set of benchmark meshes. For geometry and field coding, we derive a set of local spectral predictors optimized for each possible configuration of previously encoded and thus available vertices within a hexahedron. Combined with lossless floating-point residual coding, this approach improves considerably upon prior predictive geometry coding schemes.*

1. Introduction

Numerical simulation is an integral component of many science and engineering disciplines, with applications ranging from computational fluid dynamics to structural mechanics. Usually such simulations involve discretizing a continuous function onto a mesh. For reasons of accuracy and flexibility, unstructured hexahedral meshes are the preferred representation [1], which has spawned a tremendous amount of research effort over the past decade on developing automated hexahedral meshing methods [2] for discretizing complex geometry with well-shaped elements.

In massively parallel simulations, it is common to write both the mesh and the fields attached to it to disk periodically, e.g. once every time step, regardless of whether the mesh connectivity and geometry change over time. The mesh is most often represented as an array of 3D vertex (aka. node) positions (the geometry) and a separate array of indices (the connectivity) into the vertex array to represent the hexahedral elements (aka. cells or zones), with eight indices per hexahedron. Assuming equal numbers of vertices and hexahedra, 64-bit double precision for the geometry, and 32-bit integer indices for connectivity, this representation equates to 56 bytes per hexahedron for the function domain alone. As meshes may range up to millions of hexahedra, writing just the mesh to disk once may require gigabytes of space.

*Work performed under the auspices of the U.S. DoE by LLNL under Contract DE-AC52-07NA27344.

Data compression techniques specialized for meshes [3] offer a means to alleviate the I/O demands of large simulations. Such methods usually build a separate mesh data structure that is traversed from element to adjacent element in some deterministic order. Only a handful of symbols are needed to specify how each element is attached to the previously encoded partial mesh. The mesh geometry is often aggressively quantized to a dozen or so bits per coordinate to allow simple residual coding via a small set of linear predictors.

Unfortunately many of these prior techniques are not applicable to compression of simulation data. First, both memory and CPU time are extremely scarce resources in numerical simulation, and the time and memory needed to first construct a mesh data structure are often not available. For example, the methods proposed in [4, 5] require memory on the order of several hundreds to over a thousand bytes per hexahedron. Second, for checkpointing purposes and accurate analysis, the compression scheme must be lossless [6, 7]. Not only does this requirement rule out quantization, but it also implies that the geometry and connectivity arrays may not be reordered so that the simulation state can be perfectly recovered, e.g. for debugging purposes, to avoid numerical drift due to out-of-order floating-point operations, and to keep external references to the mesh valid. Even the “orientation” of each hexahedron, i.e. the order in which its eight vertex indices are listed, must remain intact. Until recently [8, 9] mesh compressors have relied on reordering to maximize data reduction by avoiding explicit coding of the permutations of elements. (Note that reordering for compression purposes may be acceptable as a one-time preprocess before the simulation is run.) Finally, the compression scheme must be online. Shortage of time and disk space eliminates the possibility to make one or more analysis passes over the mesh, or to first dump it uncompressed to disk. An online scheme integrates easily with applications by accepting and coding each vertex and element as they are output. From a pragmatic standpoint, such transparent and lossless compression is crucial for widespread adoption.

Faced with these requirements, we here present a simple yet effective lossless compression scheme for hexahedral meshes. We exploit the regularity in the combinatorial structure of the mesh, as well as the nearly invariably coherent orderings of vertices and elements produced by mesh generators. Rather than encoding how to attach elements to the advancing front between visited and not-yet-visited elements, we directly compress the integer-valued mesh connectivity list via hash-based value prediction [10]. Furthermore, instead of adopting sophisticated context modeling and source coding techniques, we similarly to [11, 12] rely on a general purpose text compressor such as `gzip` and focus on transforming the input into a byte-aligned stream of easy-to-compress symbols. Our method has a fixed (but configurable) memory footprint, runs in linear time, and is easily integrated with most applications with minimal code changes. Though our technique is generally less effective than state of the art, it is considerably faster and much more memory efficient and simple to implement. Moreover, our method trivially handles nonmanifold and degenerate elements and preserves the ordering of both vertices and elements. Hence it is completely lossless.

2. Connectivity compression

Hexahedral meshes usually exhibit a large amount of regularity in both geometry and connectivity that can be exploited by a compressor. Unfortunately this regularity does not emerge directly in the connectivity list. For example, running `gzip` on the raw list of 32-bit indices usually results in poor compression, in part because the indices are wider than the byte-level granularity on which `gzip` operates, but more importantly because each index

occurs at most a few times, preventing recurring patterns to form. Hence a transform that can expose the regularity to such a compressor is needed.

Contrary to tetrahedral elements, hexahedral elements tile space uniformly, and the dual of a uniform hexahedral grid is also hexahedral. These properties imply that, aside from the “curving” of space needed to mesh the boundary, the interior can predominantly be tiled with a combinatorially (and even geometrically) regular grid, in which each vertex has eight incident elements. As a consequence, most hexahedral meshes have roughly equal numbers of vertices and hexahedra.

Hexahedral meshes are with few exceptions generated by automated methods that tile the interior of a prescribed quadrilateral mesh boundary [2]. Such mesh generators tend to order elements by tracing out strands or sheets of hexahedra such that consecutive elements share a face. Because vertex and element generation usually occur in tandem, and since on average each interior hexahedral element is paired with one vertex, there is often substantial correlation between vertex and element index. Finally, in this mesh growing process, it is natural to *orient* consecutive adjacent elements consistently, i.e. so that adjacent elements agree which of the eight vertices within each element is, e.g., the “bottom-left-far” one.¹ The benchmark meshes in [Figure 1](#) have a largely consistent orientation, as is indicated by large uniformly colored regions of faces. Here all first faces (e.g. formed by the first four vertices) within the hexahedra are assigned the same color, and so on.

For systematically and consistently ordered meshes, it is possible to separate the $H \times 8$ index array of H hexahedra into eight linear streams of H indices each (see [Table 1](#)), and to compress them largely independently. This strategy, which is a significant departure from previous work on hexahedral mesh compression [4, 5], is the one we take in this paper. Our main approach is to use the last few indices as context for a value predictor, to employ byte-aligned variable-length residual coding when there is a misprediction, and to compress the resulting byte stream using a general purpose byte stream compressor such as `gzip` or `bzip2`. Because we do not reconstruct the mesh from the index list, we are not burdened with the difficulties of maintaining a mesh data structure, or handling nonmanifold or degenerate elements (e.g. with one face collapsed to an edge, forming a wedge element), but simply losslessly compress the integer index array. In the subsections below, we describe each of the components of our compressor.

2.1. DFCM index prediction

[Table 1](#) shows a piece of an index list for a regular grid. Our goal is to encode this list row by row, from left to right. Each row holds the vertices of an element that span the three spatial dimensions. Though the example in [Table 1](#) is overly simplistic, the runs of constant strides $v_i^j - v_{i-1}^j$ in each column j are prevalent also in many unstructured meshes. This suggests as index predictor the linear extrapolation $p_i^j = 2v_{i-1}^j - v_{i-2}^j$. A more general stride-based approach is the *differential finite context method* (DFCM) [13], which has been used successfully for trace file and floating-point compression [7, 12]. The basic DFCM predictor is a hash table that maps a set of recent strides to the current, predicted stride. No collision resolution is employed; insertions overwrite past hash entries. DFCM accounts for non-linearities and variations in strides by learning recurring—though not necessarily constant—stride patterns.

¹The 24 possible orientations of a hexahedron are determined by which face and vertex are specified first. Note that a globally consistent orientation is not possible unless the mesh connectivity is entirely regular.

v_i^0	v_i^1	v_i^2	v_i^3	v_i^4	v_i^5	v_i^6	v_i^7
1	2	17	18	257	258	273	274
2	3	18	19	258	259	274	275
3	4	19	20	259	260	275	276
4	5	20	21	260	261	276	277
5	6	21	22	261	262	277	278
6	7	22	23	262	263	278	279
7	8	23	24	263	264	279	280
8	9	24	25	264	265	280	281
9	10	25	26	265	266	281	282
10	11	26	27	266	267	282	283
11	12	27	28	267	268	283	284
12	13	28	29	268	269	284	285
13	14	29	30	269	270	285	286
14	15	30	31	270	271	286	287
15	16	31	32	271	272	287	288
17	18	33	34	273	274	289	290
18	19	34	35	274	275	290	291

Table 1. The first 17 lines of the ‘grid16’ connectivity list.

```

unsigned
dfcm(const unsigned* v, unsigned i) // predict vertex v[i]
{
    unsigned a = v[i - 8] - v[i - 16]; // corresponding vertex stride
    unsigned b = v[i - 1] - v[i - 9]; // previous vertex stride
    unsigned h = ((b << 6) + a) & 0xfff; // hash index
    unsigned p = hash[h] + v[i - 8] + a; // prediction
    if (p == v[i]) // is prediction correct?
        conf[h] = true; // if so, set confidence bit
    else {
        if (conf[h]) // is confidence bit set?
            conf[h] = false; // if so, clear confidence bit
        else
            hash[h] += v[i] - p; // otherwise update hash
    }
    return p; // return prediction
}

```

Listing 1. DFCM value predictor.

The free parameters in a DFCM predictor are the hash function and table size used, and the decision what to store in the table. We found hash tables as small as $2^{12} = 4,096$ entries to work sufficiently well. In choosing a hash function, we note that when a prediction fails, often the other indices within the same element are also be mispredicted, as happens for example after the element traversal hits a dead end at a boundary. We therefore use as context not only the previous stride in the same column of the index list, but also incorporate in the hash function the most recent stride from the previous column, thereby involving another index from the same element. Variations in this secondary stride diverts the hash lookup to different parts of the hash table to avoid collisions.

Listing 1 shows the actual hash function used, with the argument v holding the linearized index list in row-major order. Note that only (some of) the last 16 indices from the index list are used as context and need to be buffered. To shorten the warm-up time of the zero-initialized hash table, we store in it not the predicted stride, as in the original DFCM method, but rather the difference in strides, anticipating constant strides (as opposed to rarely occurring zero-strides). (This approach is equivalent to initializing the d^{th} entry with d in standard DFCM, but only for strides d smaller than the hash size.) This way we correctly predict all constant stride sequences without first having to learn them. One may thus view our hash entries as learned correctors to the linear extrapolation p_i^j .

In our basic scheme, we use one DFCM hash table per column to predict the eight vertex indices of a hexahedron. In a single byte we encode using one bit per vertex whether the corresponding index was correctly predicted. Any mispredicted indices are appended verbatim. Assuming 32-bit integer indices, this scheme yields a maximum compression of 32:1. We outline below several modifications and improvements to this basic scheme.

2.2. Confidence bit

As pointed out in [13], in constant-stride subsequences such as nested loops the single difference in stride due to loop restart usually causes two mispredictions: once when the counter first wraps, and once when the constant stride re-emerges since it has then been replaced by the wraparound stride in the hash table. Similar wraparounds frequently occur in hexahedral meshes, e.g. when reaching the mesh boundary or a previously generated element. To guard against such double whammies, we add a single-bit confidence counter to each hash entry that reflects whether the associated prediction was last successful. Following a misprediction the hash entry is updated only if the confidence bit is not set. This technique improved compression of all of our benchmark meshes, and on average by 17%.

2.3. Variable-length byte coding

An obvious modification to the scheme is to use variable-length residual coding for small or frequent residuals (i.e. differences between actual and predicted indices). As is common, we remap the signed residuals to ordinals by magnitude: $(0, -1, +1, -2, +2, \dots)$. We then encode these ordinals using a variable number of bytes.

Several byte-aligned coding schemes have been proposed, including fixed-to-variable codes such as radix-256 Huffman and variable-to-fixed codes such as Anh and Moffat’s carryover-12 scheme [14]. Perhaps the simplest fixed-to-variable code is the static *byte code* (see, e.g., [14]), in which an integer is broken down into seven value bits followed by a continuation bit that signals whether more value bits are needed to represent the integer. Though potentially far from optimal, we chose this code because it is simple and fast to construct, and it preserves the bit pattern in frequent (perhaps large) residuals that can be exploited by a subsequent `zlib` coding phase. Compared to using raw fixed-length indices, this modification resulted in a 72% improvement in compression on average.

2.4. Reprediction

Failed DFCM predictions are often poorly correlated with the actual indices, for two reasons. First, when the failure is due to a hash collision, the predicted and actual values may be arbitrarily far apart. Second, even if such residuals are not large, they tend not to exhibit any particular regularity, and therefore do not compress well. Therefore we propose using an alternative predictor to compute residuals when the DFCM predictor fails.

Because vertex indices are often regular and localized with respect to the sequence of hexahedra, we would expect that the index of a neighboring vertex would serve as a fair (though rarely perfect) predictor. Since we maintain only two hexahedra as context with no additional adjacency information, we limit the candidate neighbors to those either in the same or previous hexahedron as the current vertex v_i^j . Depending on the local index j of v_i^j between one and four neighbors are available: those adjacent vertices (i.e. joined by an edge) in the same hexahedron that have already been encoded, and the corresponding j^{th} vertex v_{i-1}^j in the previous hexahedron.

To choose which neighbor to use as prediction, we apply this same neighbor predictor to the eight most recently encoded vertex indices and compute residuals. We then use as prediction for the current vertex the neighbor in the direction of smallest mean absolute residual. This heuristic produces not only small residuals on average, but also regular ones whenever strides between adjacent vertices are regular, as is often the case. This scheme furthermore worked consistently and considerably better than simpler approaches, such as using a fixed, data-independent priority of directions, and always improved compression over computing residuals from the DFCM prediction, on average by 14%.

2.5. General purpose compression

So far we have described the transform phase only, which alone yields 16:1 compression on average. Dramatic improvement in compression is obtained by passing this compact byte stream through a byte-based compressor. For simplicity and speed, we chose the widely available `zlib 1.2.3` compressor on which `gzip` is based. We used the default `zlib` settings with 24 KB input and 8 KB output buffers. This improved the median compression from 16:1 to 84:1, and ratios as high as 3,800:1 were observed for near-regular meshes.

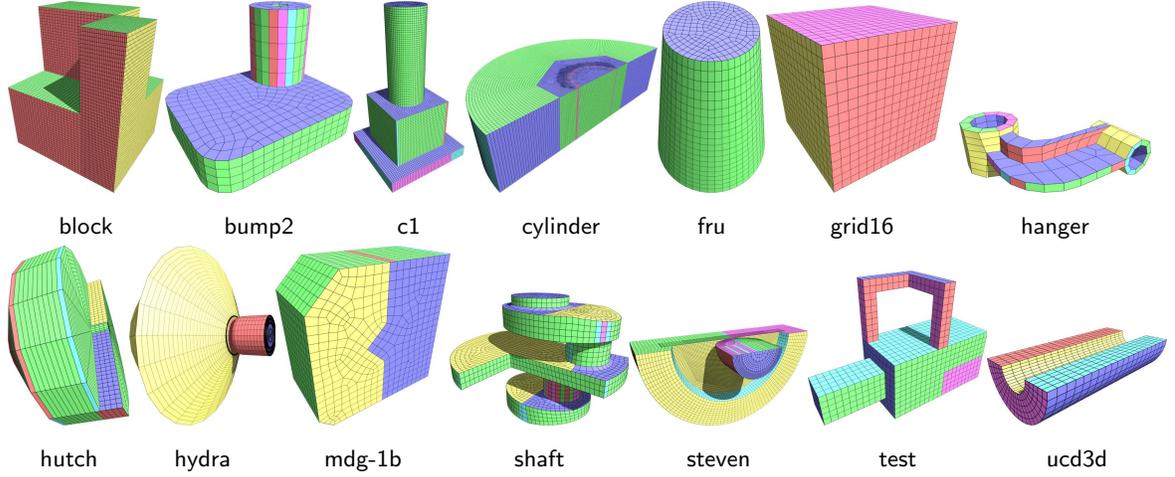


Figure 1. Meshes used in our experiments. The face colors indicate element orientation.

mesh	V	H	V_{corr}	H_{corr}	without zlib			with zlib		
					bytes	bph	time	bytes	bph	time
block	101,401	93,750	99.47%	95.98%	97,890	8.35	0.020	790	0.07	0.021
bump2	1,665	1,189	64.97%	27.25%	5,871	39.50	0.001	3,642	24.50	0.002
c1	78,618	71,572	96.98%	89.59%	105,613	11.80	0.018	13,394	1.50	0.023
cylinder	500,055	482,900	86.82%	71.38%	1,214,434	20.12	0.228	181,713	3.01	0.272
fru	5,124	4,360	94.91%	84.95%	6,903	12.67	0.002	1,666	3.06	0.002
grid16	4,096	3,375	99.01%	93.24%	3,649	8.65	0.001	88	0.21	0.001
hanger	382	171	28.14%	4.09%	1,315	61.52	0.001	1,000	46.78	0.001
hutch	8,790	8,172	81.44%	64.43%	23,583	23.09	0.005	8,862	8.68	0.007
hydra	98,357	141,960	94.25%	89.50%	237,256	13.37	0.045	47,897	2.70	0.055
mdg-1b	4,510	3,710	82.89%	57.28%	9,721	20.96	0.002	3,620	7.81	0.003
shaft	9,218	6,883	70.65%	35.44%	29,908	34.76	0.006	15,960	18.55	0.008
steven	96,030	81,832	96.52%	90.82%	112,829	11.03	0.022	20,025	1.96	0.025
test	3,198	2,386	57.87%	42.04%	12,621	42.32	0.003	6,085	20.40	0.004
ucd3d	2,646	2,000	97.23%	79.85%	2,449	9.80	0.001	76	0.30	0.001

Table 2. Connectivity compression results. V_{corr} is the fraction of correctly predicted vertex indices; H_{corr} is the fraction of hexahedra with all indices correctly predicted; bph is bits per hexahedron.

2.6. Results

We here present results of running our connectivity coder on fourteen meshes in their original orderings (available at <http://www.cc.gatech.edu/~lindstro/data/hexzip/>) on a Linux PC with two 3.2 GHz Intel Xeon CPUs, 2 GB of RAM, and a 10 KRPM SCSI disk. Table 2 lists statistics for the meshes and the performance of the DFCM predictor in terms of correctly predicted vertex indices. Considering that no effort was made to optimize the vertex or element order, the prediction accuracy is remarkably high for such a simple scheme, which is also reflected in the compression rates. Good compression is achieved for all meshes but ‘hanger’, which has few elements (171) and little regularity to tease out.

We compare our scheme with a stripped down implementation of Isenburg and Alliez’ `chvm` coder [4], which is the best hexahedral mesh compressor that we know of. For calibration we also compare with `bzip2` and `gzip` applied directly to the index list. All methods begin with a memory resident index list (not included in the memory footprint) and write

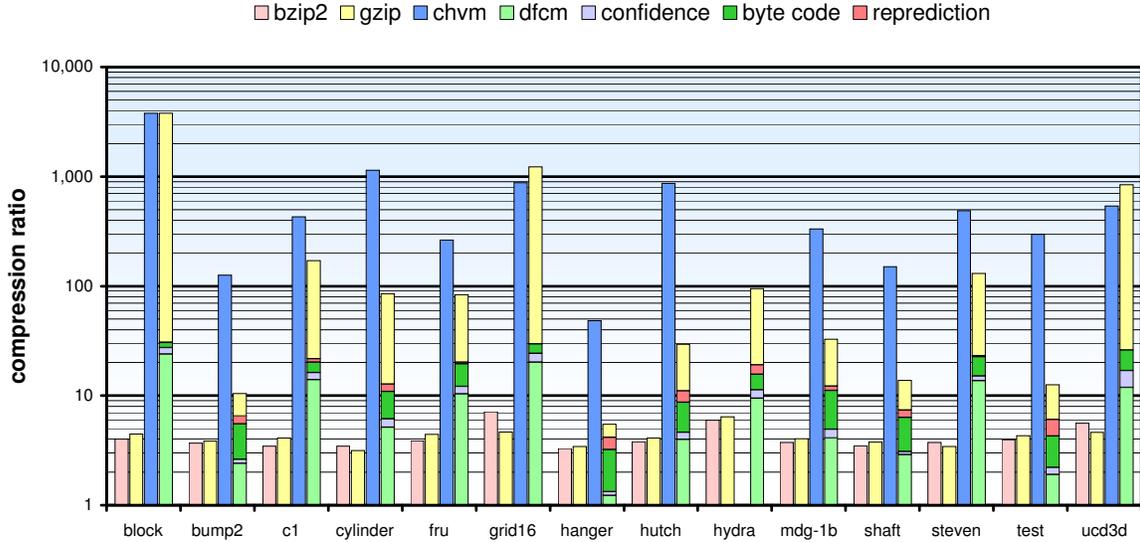


Figure 2. Connectivity compression rates for several meshes and methods. Results for the new method are broken down into five components, including the final gzip compression phase.

mesh	compression ratio				time (seconds)				memory footprint (KB)			
	bzip2	gzip	chvm	new	bzip2	gzip	chvm	new	bzip2	gzip	chvm	new
block	4.02	4.40	3,780	3,800	0.805	0.260	0.753	0.021	7,348	279	53,613	427
bump2	3.67	3.84	126	10.4	0.011	0.005	0.010	0.002	7,348	279	1,635	427
c1	3.45	4.09	428	171	0.670	0.225	0.637	0.023	7,348	279	41,828	427
cylinder	3.45	3.15	1,140	85.0	4.912	2.239	4.523	0.272	7,348	279	265,862	427
fru	3.88	4.41	262	83.7	0.031	0.013	0.039	0.002	7,348	279	3,126	427
grid16	7.12	4.63	876	1,230	0.022	0.010	0.026	0.001	7,348	279	3,029	427
hanger	3.29	3.43	48.3	5.47	0.003	0.001	0.002	0.001	7,348	279	1,672	427
hutch	3.78	4.03	863	29.5	0.059	0.030	0.064	0.007	7,348	279	6,132	427
hydra	5.97	6.33	-	94.8	4.298	0.342	-	0.055	7,348	279	-	427
mdg-1b	3.75	4.00	332	32.8	0.027	0.013	0.029	0.003	7,348	279	3,147	427
shaft	3.49	3.73	150	13.8	0.051	0.026	0.062	0.008	7,348	279	4,630	427
steven	3.75	3.44	487	131	0.721	0.356	0.724	0.025	7,348	279	47,834	427
test	3.95	4.31	296	12.5	0.019	0.009	0.019	0.004	7,348	279	1,586	427
ucd3d	5.58	4.62	537	842	0.015	0.006	0.015	0.001	7,348	279	1,582	427

Table 3. Compression, wall clock execution time, and memory usage (according to GNU memusage) for several methods. chvm [4] was not able to compress the ‘hydra’ mesh due to element degeneracies.

their output to a file. Figure 2 and Table 3 summarize the results. As is evident, our scheme compresses significantly better than bzip2 and gzip, and sometimes even better than chvm. Not surprisingly, our method cannot always compete with chvm—which by permuting vertices, elements, and orientations does not losslessly encode the index list—and occasionally fares much worse. On the other hand, our lossless method is 20–30 times faster than chvm and uses very little memory. Although both gzip and our scheme make use of the zlib library, we gain in speed by rapidly and effectively reducing the amount of data to be compressed. As a result, the zlib phase adds only about 20% to the running time.

Our compressor can be implemented using a few simple arrays. With 12-bit DFCM hash indices, its memory footprint is 427 KB: 132 KB for the hash tables, 32 + 262 KB for the zlib I/O buffers and internal state, plus a few more bytes to hold the last two hexahedra.

By comparison, `chvm` uses as much as 260 MB of RAM, which is prohibitive for memory starved applications like numerical simulations. Finally, our implementation of encoder and decoder transforms (without `zlib`) is only 200 lines of C++ code, which is insignificant compared to the more than 10,000 lines in our implementation of the `chvm` encoder alone.

2.7. Limitations

The main drawback of our method is the reliance on coherently ordered vertex and element arrays. Among the several mesh generators we have used, we have not yet encountered an incoherently ordered mesh, however some processing tasks may inadvertently scramble these arrays. To evaluate the sensitivity to ordering, we performed three experiments in which we randomly permuted vertices, hexahedra, and orientations. These experiments show that, at 2.4:1 average compression, randomizing the vertex order is far more damaging than scrambling the element order or orientation, which both allow for 5:1 compression. In case of incoherent input, preliminary results indicate that reordering can be very effective, and sometimes allows improving compression well beyond the ratios presented here.

2.8. Discussion

Our preconditioning scheme has been designed around the assumption that a compressor like `zlib` will remove any remaining redundancy in the byte stream. We here consider alternatives in case such a compressor, for whatever reason, is not available. The purpose of this discussion is two-fold: (1) to further improve the scheme in this situation, and (2) to dismiss the need for extra complexity in the transform phase whenever `zlib` is available.

As evidenced in [Table 2](#), it is quite common that all eight vertices of an element are correctly predicted. If this likelihood exceeds $\frac{1}{8} = 12.5\%$, as is the case for all but one of our benchmark meshes, it is beneficial to use a single bit to flag all-correct predictions and to conditionally omit the per-vertex flags. Such per-element flags can be packed into a single byte by encoding eight elements at a time. Without `zlib` this change boosts the maximum compression from 32:1 to 256:1 and improves the average compression by 94%. However, this change surprisingly has a net negative effect on `zlib`, in spite of a near 2:1 decimation of the byte stream, and hence we omitted this step from our main scheme.

Although residuals due to misprediction are mostly small, some not so small residuals occur with high frequency. To more efficiently encode those residuals, a *recency transform* such as move-to-front [15] may be inserted before the byte coding stage, which effectively reduces the code length of frequent residuals. Without `zlib`, such a transform improves compression by 8%, though it interferes with `zlib` by constantly “relabeling” symbols.

Finally, the use of byte code limits residual compression to 4:1 by requiring codewords to be at least eight bits long. We applied a coding scheme similar to Anh and Moffat’s world-aligned coder [14], which packs several residuals as short as a single bit each into a 32-bit word. Together with a per-element flag and recency transform, this further improved compression by 18%, however at a significant penalty in code complexity. This is due to the fact that this scheme does not emit one codeword per input symbol, which may indefinitely delay residuals and prevent them from being encoded and interleaved with the corresponding prediction flags. Addressing this requires potentially significant buffering of prediction flags and/or periodic flushing of buffered residuals in both encoder and decoder, complicating the implementation. Finally, such a bit packing approach hurts the byte-level regularity in the stream, which ultimately has a negative impact on `zlib`.

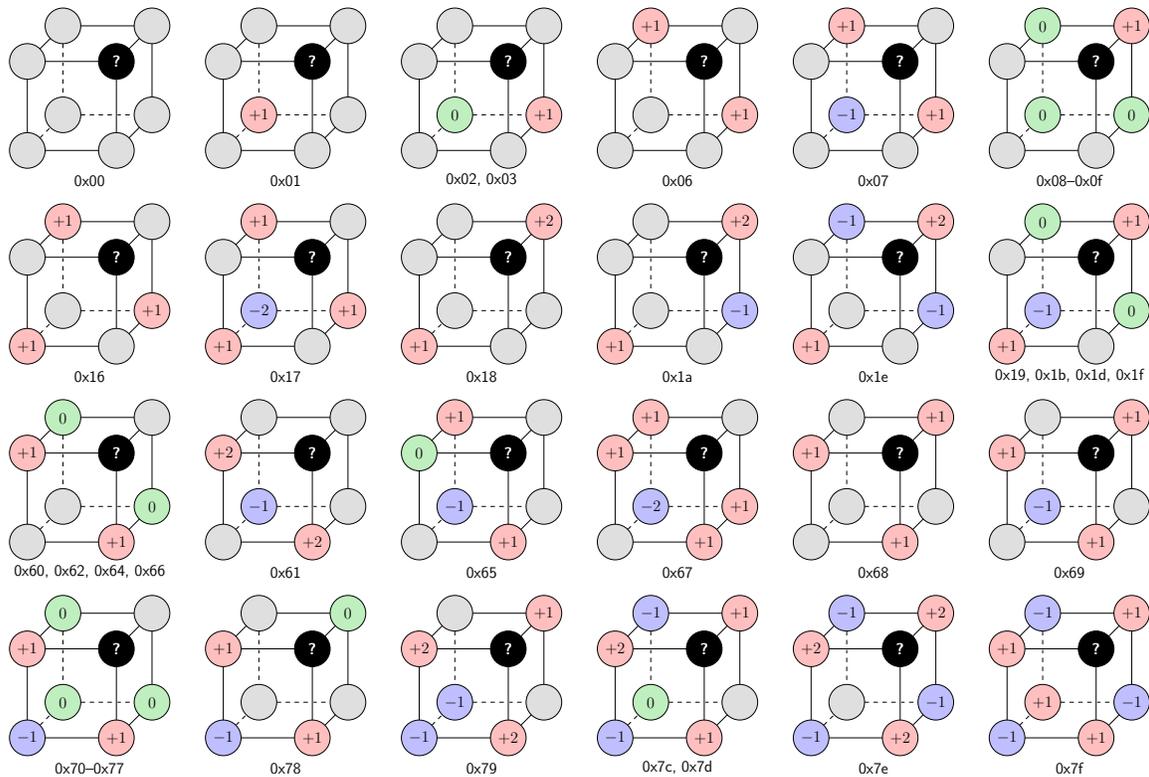


Figure 3. Spectral predictor weights (structurally equivalent cases have been omitted). Gray indicates unknown samples; green are either known or unknown with zero weight; blue have negative weight; and red have positive weight. Each weight is to be normalized by the sum of weights within a stencil.

3. Geometry compression

For completeness we here briefly outline a novel scheme for lossless compression of vertex coordinates and any other vertex-centered floating-point data. A full treatment of this problem and detailed results are deferred to an extended version of this paper.

As is common [3, 4], we use predictive coding to compress vertices in their order of reference from the index list, which on decoding specifies where in the array to place each decompressed vertex. We maintain a flag with each vertex to specify whether it has been compressed.² As a consequence of following this ordering, we usually end up with several different configurations of encoded and not-yet-encoded vertices within a single element, which is the only adjacency information we maintain and can exploit for prediction. To make the best possible use of available neighbors, we employ *spectral prediction* [16] by pre-computing “optimal” rational weights for all $2^7 = 128$ possible configurations (Figure 3). Our derivation shows *parallelogram* [4] and *Lorenzo* prediction [6, 17] to be special cases (0x70 respectively 0x7f in Figure 3) of spectral prediction.

Similar to the case of index compression, we use a single bit to signal perfect predictions and compress scalars in groups of eight. As in Section 2.3, we also order residuals by magnitude and byte code the ordinals. The resulting stream, which is kept separate from the index stream, is then compressed with `zlib`.

²A streaming implementation [9] would maintain flags and geometry only for the active front of vertices.

Using the meshes from Section 2.6 stored in double precision, we obtain a median lossless compression of 6.3, and as much as 950 times reduction of the ‘*block*’ geometry. This amounts to a 60% average improvement over using the same coding scheme but the much smaller set of spectral predictors presented in [4]. We also more than double the compression over running `gzip` on the raw floating-point array, and attribute these results to the fact that most traversals result in an abundance of 3D Lorenzo predictions, which offer outstanding prediction of the often regularly shaped hexahedral elements. Finally, we note that our geometry compression scheme is both fast and memory efficient, and requires as little as 384 bytes for the 128×8 three-bit spectral weights.

4. Conclusion

We have presented the first truly lossless compression scheme for coding connectivity and geometry in unstructured hexahedral meshes. Our approach combines byte-aligned predictive coding with a subsequent `zlib` compression phase. This simple combination results in excellent compression at a minimal CPU and memory cost. Our scheme exploits the regularity of the structure and ordering of hexahedral mesh elements. For incoherently ordered meshes, future work will investigate reordering strategies to further improve compression.

References

- [1] S. E. Benzley, E. Perry, K. Merkley, and B. Clark, “A comparison of all-hexahedral and all-tetrahedral finite element meshes for elastic and elasto-plastic analysis,” in *International Meshing Roundtable*, 1995, pp. 179–191.
- [2] S. J. Owen, “A survey of unstructured mesh generation technology,” in *International Meshing Roundtable*, 1998, pp. 239–267.
- [3] P. Alliez and C. Gotsman, *Recent Advances in Compression of 3D Meshes*. Springer, 2006, pp. 3–26.
- [4] M. Isenburg and P. Alliez, “Compressing hexahedral volume meshes,” *Graphical Models*, vol. 65, no. 4, pp. 239–257, 2003.
- [5] S. Prat, P. Gioia, Y. Bertrand, and D. Meneveau, “Connectivity compression in an arbitrary dimension,” *The Visual Computer*, vol. 21, no. 8–10, pp. 876–885, 2005.
- [6] P. Lindstrom and M. Isenburg, “Fast and efficient compression of floating-point data,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1245–1250, 2006.
- [7] M. Burtscher and P. Ratanaworabhan, “High throughput compression of double-precision floating-point data,” in *IEEE Data Compression Conference*, 2007, pp. 293–302.
- [8] M. Isenburg, P. Lindstrom, and J. Snoeyink, “Streaming compression of triangle meshes,” in *Symposium on Geometry Processing*, 2005, pp. 111–118.
- [9] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Shewchuk, “Streaming compression of tetrahedral volume meshes,” in *Graphics Interface*, 2006, pp. 115–121.
- [10] Y. Sazeides and J. E. Smith, “The predictability of data values,” in *ACM/IEEE International Symposium on Microarchitecture*, 1997, pp. 248–258.
- [11] M. Isenburg and J. Snoeyink, “Binary compression rates for ASCII formats,” in *International Conference on 3D Web Technology*, 2003, pp. 173–178.
- [12] M. Burtscher, “VPC3: A fast and effective trace-compression algorithm,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1, pp. 167–176, 2004.
- [13] B. Goeman, H. Vandierendonck, and K. de Bosschere, “Differential FCM: Increasing value prediction accuracy by improving table usage efficiency,” in *International Symposium on High-Performance Computer Architecture*, 2001, pp. 207–216.
- [14] V. N. Anh and A. Moffat, “Inverted index compression using word-aligned binary codes,” *Information Retrieval*, vol. 8, no. 1, pp. 151–166, 2005.
- [15] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, “A locally adaptive data compression scheme,” *Communications of the ACM*, vol. 29, no. 4, pp. 320–330, 1986.
- [16] L. Ibarria, P. Lindstrom, and J. Rossignac, “Spectral predictors,” in *IEEE Data Compression Conference*, 2007, pp. 163–172.
- [17] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak, “Out-of-core compression and decompression of large n -dimensional scalar fields,” *Computer Graphics Forum*, vol. 22, no. 3, pp. 343–348, 2003.