Fast and Efficient Compression of Floating-Point Data

Peter Lindstrom
LLNL

Martin Isenburg
UC Berkeley

Abstract

Large scale scientific simulation codes typically run on a cluster of CPUs that write/read time steps to/from a single file system. As data sets are constantly growing in size, this increasingly leads to I/O bottlenecks. When the rate at which data is produced exceeds the available I/O bandwidth, the simulation stalls and the CPUs are idle. Data compression can alleviate this problem by using some CPU cycles to reduce the amount of data that needs to be transfered. Most compression schemes, however, are designed to operate offline and try to maximize compression, not online throughput. Furthermore, they often require quantizing floating-point values onto a uniform integer grid, which disqualifies their use in applications where exact values must be retained.

We propose a simple and robust scheme for lossless, online compression of floating-point data that transparently integrates into the I/O of a large scale simulation cluster. A plug-in scheme for data-dependent prediction makes our scheme applicable to a wide variety of data sets used in visualization, such as unstructured meshes, point sets, images, and voxel grids. We achieve state-of-the-art compression rates *and* compression speeds, the latter in part due to an improved entropy coder. We demonstrate that this significantly accelerates I/O throughput in real simulation runs. Unlike previous schemes, our method also adapts well to variable-precision floating-point and integer data.

CR Categories: E.4 [Coding and Information Theory]: Data compaction and compression

Keywords: high throughput, lossless compression, file compaction for I/O efficiency, fast entropy coding, range coder, predictive coding, large scale simulation and visualization.

1 Introduction

Data sets from scientific simulation and scanning devices are growing in size at an exponential rate, placing great demands on memory and storage availability. Storing such data uncompressed results in large files that are slow to read from and write to disk, often causing I/O bottlenecks in simulation, data processing, and visualization that stall the application. With disk performance lagging increasingly behind the frequent doubling in CPU speed, this problem is expected to become even more urgent over the coming years.

A large scale simulation may run on a cluster of hundreds to thousands of supercomputer nodes that write the results of each time step to a shared file system for subsequent analysis and visualization [24]. Typically this involves storing large amounts of single- or double-precision floating-point numbers that represent one or more variables of simulation state per vertex/cell. When the CPU speed with which the simulation can be updated exceeds the available I/O bandwidth, the simulation stalls and the CPUs are idle.

Data compression strategies have the potential to combat this problem. By making use of excess CPU cycles, data can be compressed and uncompressed to reduce the number of bytes that need to be transferred between memory and disk or across file systems, effectively boosting I/O performance at little or no cost while reducing storage requirements.

The visualization community has developed compression schemes for unstructured data such as point sets [6, 3], triangular [27, 18], polygonal [19, 13], tetrahedral [11, 2], and hexahedral [14] meshes and for structured data such as images and voxel grids [8, 12]. However, most of these schemes are designed to maximize compression rate rather than data throughput. They are commonly applied as an offline process after the raw, *uncompressed* data has already been stored on disk. In order to maximize effective throughput, one must consider how to best balance compression speed and available I/O bandwidth, and at the same time support sufficiently efficient *decompression*. While higher compression rates improve effective bandwidth, this gain often comes at the expense of a slow and complex coding scheme.

Furthermore, prior methods often expect that vertex positions and field values can be quantized onto a uniform integer grid for efficient (but lossy) predictive compression. This modifies the original data as the non-linear precision of floating-point numbers cannot be preserved. In many science and engineering applications, however, exact values must be retained, e.g. for checkpoint dumps of simulation state and for accurate analysis and computation of derived quantities such as magnitudes, curls, fluxes, critical points, etc. The use of uniform quantization is also prohibited for data sets that exploit the non-linearity of the floating-point representation to allocate more precision to important features by specifically aligning them with the origin. Quantization can also change geometric relationships in the data (e.g. triangle orientation, Delaunay properties). Finally, scientists are often particular about their data and will simply refrain from using a compression scheme that modifies their data.

To address these needs, we propose a novel and surprisingly simple scheme for fast, lossless, online compression of floating-point data using predictive coding. It provides a well balanced trade-off between computation speed and data reduction and can be integrated almost transparently with standard I/O. Our scheme makes no assumption on the nature of data to be compressed, but relies on a plug-in scheme for computing data-dependent predictions. It is hence applicable to a wide variety of data sets used in visualization, such as unstructured meshes, point sets, images, and voxel grids. In contrast to many previous schemes, our method naturally extends to compression of adaptively quantized floating-point values and to coding of integer data.

We present results of lossless and lossy floating-point compression for scalar values of structured 2D and 3D grids, fields defined over point sets, and for geometry coding of unstructured meshes. We compare our results with recent floating-point compression schemes to show that we achieve both state-of-the-art compression rates *and* speeds. The high compression speed can be attributed in part to the use of an optimized, high-speed entropy coder, described here. As a result, our compressor is able to produce substantial increases in effective I/O rate for data-heavy applications such as large scale scientific simulations.

2 Related Work

This paper is primarily concerned with lossless floating-point compression and we now discuss prior work in this area. While our scheme extends to lossy compression of quantized float-point data and integer coding, covering the extensive work done in these areas is beyond the scope of this paper.

One approach to lossless float compression is to expand the non-linear floating-point representation to a wider linear integer representation, and to use standard compression schemes for uniformly quantized data. Usevitch [29] proposes expanding single-precision floats to large, 278-bit integers scaled by a least common exponent for coding JPEG2000 floating-point images. Similarly, Trott et al. [28] suggest expanding single-precision floats to common-exponent double-precision numbers whose 52-bit mantissas are assumed to be sufficient for representing the range of the single-precision data. Liebchen et al. [20] take a hybrid approach by choosing a suitable quantization level for MPEG audio data, applying integer compression on the quantized data, and compressing floating-point quantization residuals using Lempel-Ziv coding. The audio compressor by Ghido [10] makes a similar analysis pass over the data to discover its range and intrinsic precision to eliminate redundant bits, which due to limited sampling accuracy often occur in audio data. Gamito and Dias [9] propose a lossless wavelet coding scheme for use in JPEG2000 that separates sign, exponent, and mantissa, and that identifies regions of constant sign and exponent for efficient mantissa compression.

2.1 Streaming Floating-Point Compression

The latter three approaches [20, 10, 9] are not applicable in a streaming I/O setting as they require multiple passes over the data. Streaming compression, where data is compressed as it is written, avoids excessive delay due to buffering, is memory efficient and therefore scalable, and integrates easily with applications that produce (and consume) streams of data. Recent techniques for streaming compression of geometric data [12, 17, 15] compress vertex coordinates and field values using predictive coding. To operate losslessly on floating-point data these schemes need compatible predictive coders. We here review three floating-point compression schemes that are suitable for streaming compression. Later, we will compare our new compressor with these methods.

RKB2006 [22] The scheme by Ratanaworabhan et al. is noteworthy for its generality and independence of a geometric structure. This method compresses any linear stream of data by constructing a hash key from the last few sample differences in an attempt to find recurring patterns in the data. This allows geometry-free prediction, which works well if the data is traversed in a coherent manner so as to expose patterns, but it is not clear how well this scheme generalizes to coding of unstructured data (e.g. meshes) that has no natural traversal order. Prediction residuals are computed via an exclusive or operation, and are encoded using a fixed-width leading-zero count followed by raw transmission of all trailing bits, which makes for efficient I/O.

EFF2000 [7] The main difference in the method by Engelson et al. lies in the predictor used. Instead of hashing, values in time-varying data are predicted using 1D polynomial extrapolation of corresponding values in previous time steps. As in [22], residuals are computed by treating the binary representation of actual and predicted floats as integers. Two's complement integer subtraction results in a compressible run of leading zeros or ones. The drawback of both techniques is that they can not exploit the non-uniform distribution of leading bit counts, and that they are wasteful when the number of precision bits is not a power of two.

ILS2005 [16] Our previous scheme for single-precision floating-point compression tends to give the best compression rates compared to other schemes at the expense of higher computational complexity. The idea is to separate and compress in sequence the difference in sign, exponent, and mantissa between a float and its prediction using context-based arithmetic coding [31]. A successful prediction of the exponent, for example, is used as context for coding of mantissa differences. While effective at eliminating redundancy in the data, the implementation is riddled with conditionals and overly complicated bit manipulations, requires entropy coding of 3–5 symbols per float (even though many of these symbols are incompressible), uses up to 500 different contexts, and maintains probabilities for as many as 20,000 distinct symbols. Moreover, extending this scheme to double- and variable-precision floating-point numbers and integer data would require careful design decisions.

By contrast, the new algorithm presented here is more general in the sense that it compresses floating-point and integer values of any precision, and is also much simpler in the sense that it requires fewer operations per compressed value and fewer lines of code. As a result it is significantly faster and more memory efficient while yielding comparable compression rates. We now describe our method in detail and will return to comparisons with prior methods in Section 5.

3 Floating-Point Compression Algorithm

Our compression algorithm has been designed for IEEE floating-point numbers [1], although should be easily generalizable to similar formats. An IEEE single (double) precision number is made up of a sign bit s, an $n_e = 8$ (11) bit exponent e, and a $n_m = 23$ (52) bit mantissa m that generally represent the number

$$(-1)^s 2^{e-2^{n_e-1}-n_m+1}(2^{n_m}+m) \tag{1}$$

From here on, we will use the term "float" to generically refer to single- or double-precision floating-point numbers.

Our float compressor is not dependent on a particular prediction scheme or data type. To give this discussion context and focus, we will assume that the data to be compressed is a 3D regular grid of single- or double-precision floating-point scalar values; compression of other data types is discussed in Section 5. For completeness, we here also describe a prediction scheme for use with structured data.

In brief, our method works as follows. The data is traversed in some coherent order, e.g. row-by-row, layer-by-layer, and each value to be compressed is first predicted from a subset of the already encoded data, i.e. the data available to the decompressor. The predicted and actual values are transformed to an integer representation during which the least significant bits are optionally truncated if lossy compression is desired. Residuals are then computed and partitioned into entropy codes and raw bits, which are transmitted by the fast entropy coder discussed in Section 4.

While the steps of our algorithm are quite simple, efficient implementation requires some care. Therefore, we include source code to document each of the main steps. We begin by discussing the predictor used in our regular grid compressor.

3.1 Prediction

For regular grids we use the Lorenzo predictor [12], which is a generalization of the well-known parallelogram prediction rule [27] to arbitrary dimensions. The Lorenzo predictor

```
// compress 3D array of scalars
void compress(const FLOAT* data, front& f, int nx, int ny, int nz)
{
  f.advance(0, 0, 1); for (int z = 0; z < nz; z++) {
    f.advance(0, 1, 0); for (int y = 0; y < ny; y++) {
      f.advance(1, 0, 0); for (int x = 0; x < nx; x++) {
        FLOAT pred = f(1, 0, 0) - f(0, 1, 1) +  // Lorenzo prediction
                     f(0, 1, 0) - f(1, 0, 1) +
                     f(0, 0, 1) - f(1, 1, 0) +
                     f(1, 1, 1);
        FLOAT real = *data++;                   // fetch actual value
        real = encode(real, pred);              // encode difference
        f.push(real);                           // put on front
      }
    }
  }
}
```

Listing 1: Data prediction and compression loop for 3D grids.

estimates a hypercube corner sample from its other, previously encoded corners by adding those samples that are an odd number of edges from the sample and subtracting those that are an even number of edges away. As only immediate neighbors need to be maintained for prediction, the compressor (and decompressor) must not keep track of more than an $(n-1)$-dimensional front (slice) from the n-dimensional data [12]. Previously encoded samples $f_{x-i,y-j,z-k}$ relative to the current sample $f_{x,y,z}$ are indexed as `f(i, j, k)` in Listing 1 by representing the front as a circular array.

To bootstrap the predictor and allow boundary samples to be predicted, one usually lowers the dimension of the Lorenzo predictor, so that the first layer is predicted using 2D prediction and the first row using 1D prediction. The first sample encoded is predicted as zero. Unfortunately, on 3D data this results in eight different predictors and hence eight conditionals in the inner loop, which degrade performance. We make the observation that $(n-1)$-dimensional Lorenzo prediction is equivalent to n-dimensional prediction with the n^{th} dimension samples set to zero. Hence, by padding the data set with one layer of zeros in each dimension, a single n-dimensional predictor can be used for all samples. Instead of copying and padding the entire data set, this padding can be done efficiently only to the front. The calls `f.advance` in Listing 1 perform this zero padding and advance the front by one layer, row, or column. In case of lossy compression, where we allow truncation of the floats, we must update the front (the `f.push` call) with the lossily encoded samples since those are the only samples available to the decompressor.

By default our compressor performs prediction using floating-point arithmetic. The order of operations and the precision used must match exactly between compressor and decompressor. This may be difficult to achieve due to compiler optimizations, roundoff policies, availability of extended precision (as on Intel architectures), and other platform dependent differences in floating-point arithmetic. In such cases, one may perform predictions using integer arithmetic at a small cost in compression rate by first mapping the floats to their binary representation. This mapping, which is applied to both predicted and actual samples regardless of how prediction is done, will be discussed next.

3.2 Mapping to Integer

We could compute prediction residuals via floating-point subtraction, however this might cause underflow with irreversible loss of information that precludes reconstruction of the actual value. Instead, as in [7, 22], we map the predicted and actual floats p and f to their sign-magnitude binary integer representation. On platforms implementing sign-magnitude integer arithmetic, we could now simply compute integer residuals via subtraction, however most current platforms implement two's complement arithmetic. To address this, we map the sign-magnitude representation to unsigned

```
// encode actual number 'real' given prediction 'pred'
FLOAT encode(FLOAT real, FLOAT pred)
{
  UINT r = forward(real);             // monotonically map floats to their...
  UINT p = forward(pred);             // ... unsigned binary representation
  if (p < r) {                        // case 1: underprediction
    UINT d = r - p;                   // absolute difference
    unsigned k = msb(d);              // find most significant bit k
    encode(zero + (k + 1), model);    // entropy code k
    encode(d - (1 << k), k);          // code remaining k bits verbatim
  }
  else if (p > r) {                   // case 2: overprediction
    UINT d = p - r;                   // absolute difference
    unsigned k = msb(d);              // find most significant bit k
    encode(zero - (k + 1), model);    // entropy code k
    encode(d - (1 << k), k);          // code remaining k bits verbatim
  }
  else                                // case 3: perfect prediction
    encode(zero, model);              // entropy code zero symbol
  return inverse(r);                  // return possibly quantized value
}
```

Listing 2: Predictive floating-point coding scheme.

integers by flipping either the most significant bit (for positive floats) or all bits (for negative floats). The result is a monotonic mapping of floats to unsigned integers that preserves ordering and even linearity of differences for floats with the same sign and exponent. This approach is also similar to [16], however we benefit by allowing a carry to propagate from mantissa to exponent in case p and f are close but separated by an exponent boundary, which would be signaled as a large misprediction in [16].

In case lossy compression is desired, we discard some of the least significant bits during the mapping stage. This can be thought of as logarithmic rather than uniform quantization, which in our experience is the quantization preferred by scientists. They often describe the precision of their data as the number of decimal digits in scientific notation.

3.3 Residual Computation and Coding

Once the actual and predicted values f and p have been mapped to integers, we apply a two-level compression scheme to the integer residual $\hat{d}$ (Listing 2). Using one symbol (and probability) per residual is not practical in our scheme for two reasons. First, because of the potentially large range $(-2^n, +2^n)$ of residuals for n-bit data, the probability tables would become prohibitively large. Second, because there are generally many more possible residuals than actual floats in a data stream, most residuals are expected to appear only once, making probability modeling unreliable at best. To address this problem, we use a two-level scheme that groups residuals into a small set of intervals. A residual can then be represented by interval number and position within the interval. We considered using the optimal two-level scheme by Chen et al. [4], which partitions a distribution so as to minimize the coding cost, but opted for a static and simpler scheme. Observing that most residual distributions are geometric and highly peaked around $\hat{d} = 0$, grouping residuals into variable-size intervals $\pm[2^k, 2^{k+1})$ makes for both a simple and effective scheme. This approach, equivalent to coding the number of leading zeros of $|\hat{d}|$, is essentially the one taken by [7, 22], although our scheme differs in one important aspect: the manner in which the first-level intervals are coded.

Formally, we define the integer residual $\hat{d}$ as:

$$\hat{d} = \hat{f} - \hat{p} = s(2^k + m) \tag{2}$$

where $s \in \{-1, 0, +1\}$ encodes the sign of $\hat{d}$, $0 \leq k < n$ is the position of the most significant bit of $|\hat{d}|$, and m is a length-k bit string. k can be computed quickly either by repeated right shifting or using the `bsr` Intel assembly instruction. Whereas both [7, 22] would encode the tuple $\langle s, k \rangle$ using a fixed number of bits, we exploit the non-uniform distribution of k and use entropy coding. If $\hat{d} = 0$, we entropy code only a single symbol $g = 0$. Otherwise, we first entropy code a symbol $g = s(k+1)$, followed by transmitting verbatim the remaining k bits representing m. (Note that k may be zero, e.g. when $\hat{d} = 1$, in which case no additional bits are transmitted.) Whereas we could fold $\langle s, k \rangle$ into n distinct symbols via modular arithmetic, as in [7, 16], or using exclusive-or differencing, as in [22], we use all $2n+1$ symbols and rely on the range coder to (nearly) eliminate the cost of coding large, infrequent residuals. As we shall see later, using entropy coding to compress g can considerably improve the overall compression rate at little or no expense in speed.

While the raw bit stream $\mathcal{M}$ could be transmitted independently of the symbol stream $\mathcal{G}$, interleaving and synchronizing the two streams is a non-trivial problem considering the stream $\mathcal{G}$ produced by our range coder contains symbols of "fractional" bit length. Moreover, our range coder is sufficiently fast that coding raw bits does not pose a significant overhead. In general, these raw bits do not have much regularity that could be exploited for further compression. One notable exception is the case when the data intrinsically has less than full precision, in which case additional entropy coding can remove redundant bits [10, 16].

4 Fast Entropy Coding

Range coding [21] is an efficient variation on arithmetic coding [31] that outputs data in byte increments. As in arithmetic coding, an interval $[l, l+r)$ is maintained that uniquely represents a string of encoded symbols. During encoding of one of a set of possible symbols, the interval is partitioned such that symbols are assigned non-overlapping portions of the interval in proportion to their probability, and the interval is then narrowed to the sub-interval corresponding to the encoded symbol. The process repeats for the next symbol. Any number within the final near-infinite-precision interval then encodes an entire symbol sequence.

In practice, fixed-point integer arithmetic is used to represent a subset of $[0, 1)$. To avoid working with infinite precision, the most significant bits of the interval can be output whenever they agree in both l and $l+r$. One problem arises when r becomes small but l and $l+r$ straddle a bit (or byte, in case of range coding) boundary. To avoid running out of finite precision for r, two solutions have been proposed: (1) Output a zero bit and later correct it once it is determined that a carry would have turned this bit into a one. This may in the worst case require buffering many bits. This approach is implemented in Schindler's range coder [23], which was used in our previous compressor [16]. (2) Since encoder and decoder are synchronized and agree on the value of r, they can both detect this condition and handle it by simply reducing r just enough that a carry can no longer occur. This computationally more efficient method was first proposed by Subbotin [26], and is the one used in our new compressor. We improve upon Subbotin's implementation by splitting the two conditions for outputting a byte and reducing the range r, and by making the observation that if no bytes can be output and the integer $r < 2^{16}$, the subsequent reduction of r implies that the top two bytes, and only the top two bytes, can *always* be output. Because these tests are performed for every encoded symbol, whether entropy coded or transmitted raw, this seemingly trivial improvement can have a measurable impact.

The code for our range coder is shown in Listing 3. Not included here is the code for probability modeling, which is done by the quasistatic probability modeler from [23].

```
// encode a symbol s using probability modeling
void encode(unsigned s, model* m)
{
  range /= m->tot;             // tot = sum of pmf
  low += range * m->cdf[s];    // cdf = cum. distribution function P(x < s)
  range *= m->pmf[s];          // pmf = probability mass function P(x = s)
  update(s, m);                // update probabilities
  normalize();                 // normalize interval
}

// encode an n-bit number s : 0 <= s < 2^n <= 2^16
void encode(unsigned s, unsigned n)
{
  range >>= n;                 // scale interval
  low += range * s;            // adjust lower bound
  normalize();                 // normalize interval
}

// normalize the range and output data
void normalize()
{
  while (((low ^ (low + range)) >> 24) == 0) {
    putbyte();                 // top 8 bits of interval are fixed;...
    range <<= 8;               // ... output them and normalize interval
  }
  if ((range >> 16) == 0) {
    putbyte();                 // top 8 bits are not fixed but range...
    putbyte();                 // ... is small; fudge range to avoid...
    range = -low;              // ... carry and output top 16 bits
  }
}

// output most significant byte
void putbyte()
{
  putchar(low >> 24);          // output top 8 bits
  low <<= 8;                   // shift out top 8 bits
}
```

Listing 3: Fast range coder used in our compressor. The range coder makes use of an external probability modeler that periodically (e.g. every 1K symbols) updates the `pmf` and `cdf` arrays.

5 Results

We evaluated our compressor against our own implementations of the three streaming schemes [7, 16, 22] and the generic zlib compressor (the scheme used in Unix gzip). We compressed 2D and 3D single- and double-precision data sets from the fluid dynamics simulation code Miranda [5] and the last time step of the hurricane Isabel data used in the Visualization 2004 contest [30]. We also compressed a large point set from an atomistic simulation of shock propagation, as well as the vertex coordinates of benchmark triangle meshes *lucy* and *david* (see [17]) and tetrahedral meshes *torso* and *rbl* (see [15]). Whereas the original scheme by Engelson et al. [7] uses 1D temporal prediction, we used Lorenzo prediction on the grid data for all schemes but [22], whose main distinguishing feature from [7] is its hash-based predictor. We did this because we did not have access to multiple time steps for all data sets, and also to factor out the impact of different data predictors and their dependence on temporal versus spatial resolution. For all schemes but [22], we used as predictor the previous sample for the partially coherent point stream, parallelogram prediction [27] for the triangle meshes, and the base triangle midpoint [11] for the tetrahedral meshes. Our experiments were run on a dual 3.2 GHz Intel Xeon with 2 GB of main memory and a Seagate Cheetah 10K.6 Ultra 320 SCSI disk.

5.1 Compression Rates

Results of lossless compression on several quite different data sets (Figure 1) are presented in Table 1 and Figure 2. The Miranda Rayleigh-Taylor simulations involve two fluids of different density that initially are separated into mostly homogeneous regions. Hence the density fields have low entropy and compress well. Most of the other fields span negative and positive numbers, resulting in a large number of different exponents (nearly all 32 respectively 64 bits are used for the single- and double-precision data). We note that the hurricane data uses the value 10^{35} to indicate land at ground level (roughly 0.4% of all values). We did not

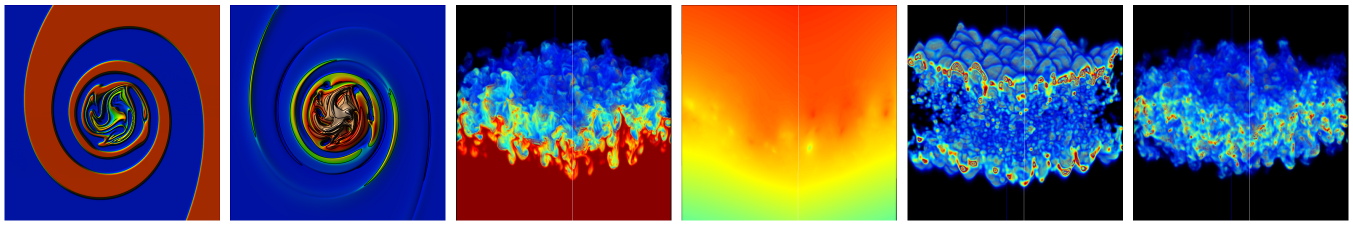
(a) 2D Density (b) 2D Vorticity (c) 3D Density (d) 3D Pressure (e) 3D Diffusivity (f) 3D Viscocity

Figure 1: Visualizations of 2D data (as pseudocolored height fields) and 3D data (volume rendered) used in our experiments.

data set								compressed size (MB) and compression time (seconds)									
name	unique (%)	entropy (bits)	range (bits)	min	max	size (MB)	time (sec)	zlib		[RKB2006]		[EFF2000]		[ILS2005]		new scheme	
m2d density	3.89	3.49	21.83	8.7E-01	1.2E+00	19.6	0.71	1.6	0.86	4.3	**0.49**	4.4	0.56	**1.3**	1.08	1.3	0.56
m2d vorticity	99.20	22.25	31.05	-1.4E+02	2.5E+01	19.6	0.71	18.4	2.14	**11.8**	**1.21**	15.5	1.29	12.9	2.22	13.8	1.49
m3d density	7.67	5.16	23.60	1.0E+00	3.0E+00	364.5	12.81	50.4	17.55	100.5	9.06	96.3	**8.48**	35.7	19.03	**35.5**	9.25
m3d pressure	27.29	23.91	31.06	-3.7E+00	2.3E+03	364.5	12.80	229.2	99.76	95.6	9.31	87.9	**8.87**	**40.1**	18.79	40.4	9.96
m3d diffusivity	36.87	23.19	30.02	0.0E+00	6.8E+00	364.5	12.68	297.6	42.90	250.8	19.09	239.3	**15.02**	**198.8**	31.92	203.0	18.47
m3d viscocity	50.07	24.86	28.59	8.6E-15	2.9E+00	364.5	12.62	314.0	46.09	249.4	18.95	246.1	**14.68**	209.2	32.66	**207.5**	19.45
h3d temp	65.70	23.54	31.56	-7.7E+01	1.0E+35	95.4	3.77	75.8	14.56	59.3	4.64	53.0	**4.27**	**44.1**	8.04	44.1	5.06
h3d pressure	81.82	24.13	31.58	-3.4E+03	1.0E+35	95.4	3.78	82.3	12.00	64.3	5.14	52.9	**4.87**	**45.0**	7.78	45.2	5.34
h3d x velocity	84.18	24.18	31.55	-5.3E+01	1.0E+35	95.4	3.89	86.1	11.27	67.4	6.22	63.3	**4.59**	**54.5**	8.86	55.4	5.44
h3d y velocity	84.32	24.18	31.55	-4.6E+01	1.0E+35	95.4	3.83	84.5	11.42	67.1	5.74	62.3	**5.04**	**53.5**	8.64	53.8	5.53
h3d z velocity	86.82	24.24	31.54	-3.2E+00	1.0E+35	95.4	3.87	88.4	10.76	85.6	8.50	76.9	**5.29**	**68.9**	9.83	69.1	6.65
M3d density	40.14	18.84	52.59	1.0E+00	3.0E+00	288.0	11.28	136.8	41.91	160.3	11.63	121.6	**10.94**	-		**105.2**	11.63
M3d pressure	100.00	25.17	63.00	-2.2E+00	2.2E+00	288.0	11.20	272.6	35.18	237.3	**14.91**	225.1	16.59	-		**208.4**	17.20
M3d x velocity	100.00	25.17	63.00	-2.2E+00	2.3E+00	288.0	10.83	275.6	32.30	230.4	**14.73**	215.1	15.91	-		**197.7**	16.84
M3d y velocity	100.00	25.17	63.00	-2.1E+00	2.3E+00	288.0	10.54	275.1	32.19	223.1	**14.27**	215.2	15.16	-		**197.7**	16.65
M3d z velocity	100.00	25.17	63.00	-5.2E+00	9.0E+00	288.0	10.32	275.5	32.62	226.6	**14.74**	213.7	16.05	-		**196.8**	16.14
a3d x position	61.10	23.82	31.01	-4.8E-02	4.6E+02	107.7	7.07	84.3	21.18	76.0	7.88	78.8	**7.61**	**67.3**	12.88	68.6	9.07
a3d y position	45.90	23.32	26.99	3.7E-02	2.1E+03	107.7	7.08	65.9	30.76	60.4	6.97	56.4	**6.31**	47.0	10.49	**46.9**	7.73
a3d z position	61.68	23.84	27.48	9.1E-05	4.6E+02	107.7	7.46	94.6	19.86	82.6	9.00	86.1	**8.25**	**75.7**	13.80	78.2	9.93
a3d y velocity	64.65	23.87	30.96	-1.5E-01	1.4E-01	107.7	7.30	95.7	19.88	93.8	10.07	99.1	**9.65**	**84.3**	14.93	87.6	9.92
a3d temp	64.91	23.94	27.41	3.0E-03	7.1E+03	107.7	6.69	95.7	19.76	91.6	10.27	95.9	**8.34**	**84.6**	15.02	84.6	10.31
a3d energy	3.45	18.57	21.79	-3.6E+00	-2.7E+00	107.7	7.15	77.9	38.59	74.1	7.98	71.8	**7.01**	60.8	12.66	**60.5**	8.30
lucy	61.39	24.38	31.09	-6.1E+02	1.2E+03	160.5	-	137.8	-	99.5	-	90.0	-	**73.6**	-	77.8	-
david$_{1mm}$	25.23	17.08	31.11	-4.4E+03	1.8E+03	322.5	-	144.9	-	155.7	-	163.4	-	**108.6**	-	131.9	-
torso	84.72	18.48	31.08	-2.7E+02	5.8E+02	1.9	-	1.7	-	1.5	-	1.5	-	**1.3**	-	1.3	-
rbl	71.90	20.14	25.99	1.48E+00	3.6E+02	8.4	-	7.1	-	5.8	-	5.6	-	**4.7**	-	4.8	-

Table 1: Compression results for the Miranda (m2d, m3d, M3d) and hurricane (h3d) structured grid data, the atom (a3d) point set, the lucy and david triangle meshes, and the torso and rbl tetrahedral meshes. All data sets except M3d are represented in single precision. The [ILS2005] scheme operates on single precision only, hence the missing values. The mesh compression timings are entirely dominated by connectivity coding, hence we do not include them here. The range measures (the logarithm of) the number of floating-point values between the minimum and maximum value. Note that the first-order entropy is limited by the total number of samples in a data set.

specialize the compressors to ignore these values.

As can be seen, our compression rates are comparable to those of [16] and significantly better than both [7, 22]. We achieve lossless reductions in the range 1.4–15 and on average a compressed size of 10.7 bits/float on the Miranda single-precision data and 18.0 bits/float on the hurricane data. Using the previous sample to predict the next, our compression results on the point data are dictated by the geometric coherence of the data stream. The atom data set is bucketed and roughly organized along one axis, but is locally not particularly coherent. We achieve an average compression of 1.5. More sophisticated point compression techniques based on local point reordering and higher-order prediction would likely improve compression rates. On the double-precision Miranda data, the lossless reduction is only 1.4–2.7, with an average size of 40.3 bits/double.

Double-precision floating point data is more challenging to compress as the increase from 23 to 52 mantissa bits adds 29 low-order bits to each value. It is well known that predictive coding mainly "predicts away" high-order bits so that the relative reduction rate decreases as low-order bits are added [13, 16]. Of course, it is possible that the low-order bits exhibit some predictable pattern, e.g. when some or all 29 additional low-order bits are everywhere zero, as would be the case if single-precision floating-point numbers were cast to double precision. A similar situation arises in Marching Cubes isosurface extraction from regular grids, for which two of three coordinates of each vertex have much less precision than can be represented in floating-point, resulting in predictable (though not necessarily all zero) low-order bits. (Even scanned surfaces, including lucy and david, are typically extracted from a volumetric representation.) Arguably such data sets should use an integer rather than floating-point representation, although for simplicity or for other reasons it is common practice to use floating-point. Contrary to [16], which uses entropy coding for all bits of the residual, our new coder sacrifices such potential compression gains for speed by storing these repeated low-order bits in raw and uncompressed form. However, the massive data sets from scientific simulation that motivated our work on high-speed compression, as well as the tetrahedral meshes, rarely exhibit significant low-order redundancy, as also evidenced by our results.

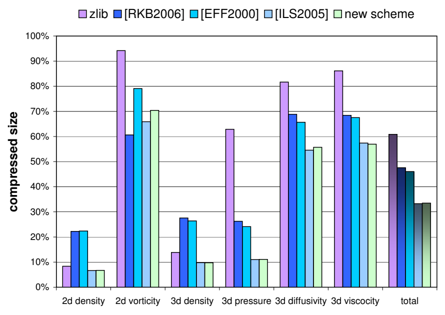

Figure 2: Compression **rates** for the Miranda single-precision data.

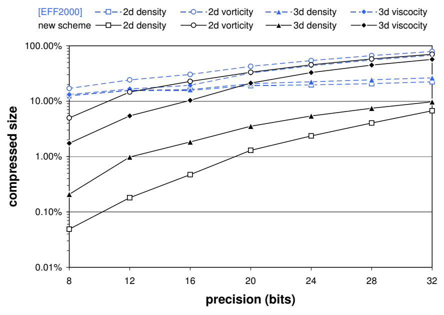

Figure 3: Lossy compression rates for our scheme (black) and [7] (blue). Note that the vertical scale is logarithmic.

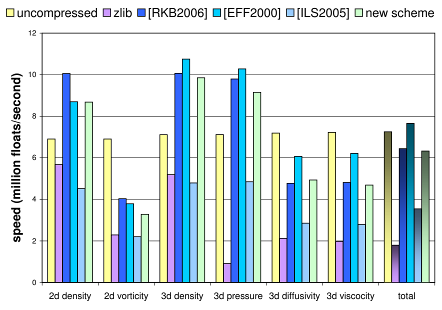

Figure 4: Compression **speed** for the Miranda single-precision data.

5.1.1 Lossy Compression

Figure 3 shows that our scheme gracefully adapts to decreasing levels of precision when discarding the least significant mantissa (and eventually exponent) bits. For n bits of precision, the schemes [7, 22] minimally require $\log_2 n$ bits to code the number of leading zeros, whereas our scheme can exploit the combination of low entropy in the leading-zero count and the elimination of the low-end mantissa bits that are most difficult to predict and compress.

5.2 Compression Speed

Figure 4 shows the speeds of compressing from memory to disk in millions of floats per second, including disk write time. (Because of the simplicity of our method, its decompression speed is similar to its compression speed.) We also include the raw I/O performance of dumping the data uncompressed using a single `fwrite` call. Timings correspond to the median of five runs. Whereas our compressor is slightly slower than the less effective compressors [7, 22], it is nearly twice as fast as [16] while producing similar compression rates. However, in more I/O-intensive scenarios, such as in massively parallel simulations dumping data to the same high-performance file system (as is common), the improved compression of our method over [7, 22] results in a net gain in effective throughput. We integrated our compression code with Miranda's dump routines and ran performance tests on 256 nodes of LLNL's MCR supercomputer. Achieving on average a lossless reduction of 3.7 on 75 GB of data dumped, the overall dump time was reduced by a factor of 2.7 over writing the data uncompressed.

5.3 Entropy Coding

We compared the raw throughput of our range coder and Schindler's [23] by (1) passing raw bytes through it with no compression and (2) entropy coding byte sequences. In both cases, the source data was the uncompressed floating-point data sets used in our experiments. Timing results show that our coder is 40% faster for raw transmission and 28% faster for entropy coding. Meanwhile, the inefficiency of our coder due to loss of precision and range reduction is only 26 bytes of overhead for 1.5 GB of coded data. Its raw throughput is only 20% less than an `fwrite` call, while its entropy coding throughput of 20 MB per second, which includes probability modeling and I/O time, compares favorably with state-of-the-art entropy coders [25].

6 Conclusion

We have described a simple and robust method for lossless compression of floating-point data based on predictive coding. The main characteristics of our method are (1) effective predictive coding for floating-point data, (2) efficient arithmetic and robustness of implementation by treating the floating-point number as an unsigned integer, (3) high-speed adaptive range coding of leading zeros in residuals, and (4) transmission of raw bits whenever we cannot expect compression to result in much gain. Our scheme provides high compression rates without sacrificing computational efficiency, thereby delivering high throughput when used in a typical large scale simulation environment where I/O bandwidth is an especially precious resource.

We achieve compression rates nearly as good as those of [16], but at twice the speed and using a much simpler implementation. Although our compression speeds are slightly slower than those of [7] and [22], our compression rates are significantly higher, resulting in higher overall throughput and smaller files in bandwidth-limited environments.

Our fast range coder may also prove useful outside the context of lossless floating-point compression. It is notably faster than Schindler's range coder [23] in both scenarios: when actually making use of entropy coding to compress data and when merely including raw uncompressible bits into the bit stream. The achieved throughput for entropy coding, including probability modeling and I/O time, compares favorably with the state of the art.

References

[1] IEEE 754: Standard for Binary Floating-Point Arithmetic, 1985.

[2] U. Bischoff and J. Rossignac. TetStreamer: Compressed Back-to-Front Transmission of Delaunay Tetrahedra Meshes. *Proceedings of Data Compression Conference*, 93–102. 2005.

[3] D. Chen, Y.-J. Chiang, and N. Memon. Lossless Compression of Point-based 3D Models. *Proceedings of Pacific Graphics*, 124–126. 2005.

[4] D. Chen, Y.-J. Chiang, N. Memon, and X. Wu. Optimal Alphabet Partitioning for Semi-Adaptive Coding of Sources of Unknown Sparse Distributions. *Proceedings of Data Compression Conference*, 372–381. 2003.

[5] A. W. Cook, W. H. Cabot, P. L. Williams, B. J. Miller, B. R. de Supinski, R. K. Yates, and M. L. Welcome. Tera-Scalable Algorithms for Variable-Density Elliptic Hydrodynamics with Spectral Accuracy. *SC '05: Proceedings of the ACM/IEEE Conference on Supercomputing*, 60. 2005.

[6] O. Devillers and P.-M. Gandoin. Geometric Compression for Interactive Transmission. *Visualization'00 Proceedings*, 319–326. 2000.

[7] V. Engelson, D. Fritzson, and P. Fritzson. Lossless compression of high-volume numerical data from simulations. *Proceedings of Data Compression Conference*, 574–586. 2000.

[8] J. Fowler and R. Yagel. Lossless Compression of Volume Data. *Symposium on Volume Visualization*, 43–50. 1994.

[9] M. N. Gamito and M. S. Dias. Lossless Coding of Floating Point Data with JPEG 2000 Part 10. *Applications of Digital Image Processing XXVII*, 276–287. 2004.

[10] F. Ghido. An efficient algorithm for lossless compression of IEEE float audio. *Proceedings of Data Compression Conference*, 429–438. 2004.

[11] S. Gumhold, S. Guthe, and W. Strasser. Tetrahedral Mesh Compression with the Cut-Border Machine. *Visualization'99 Proceedings*, 51–58. 1999.

[12] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak. Out-of-core compression and decompression of large n-dimensional scalar fields. *Proceedings of Eurographics'03*, 343–348. 2003.

[13] M. Isenburg and P. Alliez. Compressing Polygon Mesh Geometry with Parallelogram Prediction. *Visualization'02 Proceedings*, 141–146. 2002.

[14] M. Isenburg and P. Alliez. Compressing Hexahedral Volume Meshes. *Graphical Models*, 65(4):239–257, 2003.

[15] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Shewchuk. Streaming Compression of Tetrahedral Volume Meshes. *Proceedings of Graphics Interface*, 115–121. 2006.

[16] M. Isenburg, P. Lindstrom, and J. Snoeyink. Lossless Compression of Predicted Floating-Point Geometry. *Computer-Aided Design*, 37(8):869–877, 2005.

[17] M. Isenburg, P. Lindstrom, and J. Snoeyink. Streaming Compression of Triangle Meshes. *Proceedings of 3rd Symposium on Geometry Processing*, 111–118. 2005.

[18] F. Kälberer, K. Polthier, U. Reitebuch, and M. Wardetzky. FreeLence - Coding with free valences. *Eurographics'05 Proceedings*, 469–478. 2005.

[19] A. Khodakovsky, P. Alliez, M. Desbrun, and P. Schroeder. Near-Optimal Connectivity Encoding of 2-Manifold Polygon Meshes. *Graphical Models*, 64(3-4):147–168, 2002.

[20] T. Liebchen, T. Moriya, N. Harada, Y. Kamamoto, and Y. A. Reznik. The MPEG-4 Audio Lossless Coding (ALS) Standard – Technology and Applications. *119th Audio Engineering Society Convention*. 2005.

[21] G. N. N. Martin. Range encoding: an algorithm for removing redundancy from a digitized message. *Video and Data Recording Conference*. 1979.

[22] P. Ratanaworabhan, J. Ke, and M. Burtscher. Fast Lossless Compression of Scientific Floating-Point Data. *Proceedings of Data Compression Conference*, 133–142. 2006.

[23] M. Schindler. Range Encoder version 1.3, 2000. URL http://www.compressconsult.com/rangecoder/.

[24] P. Schwan. Lustre: Building a File System for 1,000-node Clusters. *Proceedings of the Linux Symposium*, 401–408. 2003.

[25] J. Senecal, M. Duchaineau, and K. I. Joy. Length-Limited Variable-to-Variable Length Codes For High-Performance Entropy Coding. *Proceedings of Data Compression Conference*, 389–398. 2004.

[26] D. Subbotin. Carryless Rangecoder, 1999. URL http://search.cpan.org/src/SALVA/Compress-PPMd-0.10/Coder.hpp.

[27] C. Touma and C. Gotsman. Triangle mesh compression. *Graphics Interface'98 Proceedings*, 26–34. 1998.

[28] A. Trott, R. Moorhead, and J. McGinley. Wavelets Applied to Lossless Compression and Progressive Transmission of Floating Point Data in 3-D Curvilinear Grids. *Proceedings of Visualization'96*, 385–358. 1996.

[29] B. E. Usevitch. JPEG2000 extensions for bit plane coding of floating point data. *Proceedings of Data Compression Conference*, 451. 2003.

[30] Visualization Contest Data Set, 2004. URL http://vis.computer.org/vis2004contest/data.html.

[31] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30(6):520–540, 1987.