Honours Project Report

Intra-frame Compression of Molecular Dynamics Simulations of Water

Keegan Carruthers-Smith
ksmith@cs.uct.ac.za

Supervised By:

Patrick Marais
patrick@cs.uct.ac.za

James Gain
jgain@cs.uct.ac.za

	Category	Chosen
1	Software Engineering/System Analysis	5
2	Theoretical Analysis	4
3	Experiment Design and Execution	5
4	System Development and Implementation	15
5	Results, Findings and Conclusion	15
6	Aim Formulation and Background Work	10
7	Quality of Report Writing and Presentation	10
8	Adherence to Project Proposal and Quality of Deliverables	10
9	Overall General Project Evaluation	6
Total marks		80

Department of Computer Science

University of Cape Town

2009

Abstract

Molecular dynamics (MD) simulations generate vast amounts of data. A typical 100-million atom MD simulation produces approximately 5 gigabytes of data per frame consisting of atom types, coordinates and velocities.

The main contribution of the report is the specification of an MD compressor which targets simulations with large amounts of water in it. Water is targeted since many MD simulations contain significant amounts of water molecules. It uses an approach based on predictive point cloud compressors, but with predictors tailored towards models of water.

A point cloud is a collection of points in 3D space. The method improves on existing point cloud compressors [Gumhold et al., 2005, Devillers and Gandoin, 2000] and MD compressor [Omeltchenko et al., 2000] when applied to MD data with significant amounts of water molecules.

There are six MD simulations the compression schemes are tested and compared against. Our water compression scheme performs best in general. At 8-bit quantisation the mean and average compression rate for water compression is 17.7%. The next best compression rate was gzip with a mean and average compression rate of 19.5% and 18.5% respectively. At 12-bit quantisation the mean and average compression rate for water compression is 29.8%. The next best scheme is gzip which has a mean and average compression rate of 37% and 36.5% respectively.

The report also presents techniques for compressing permutations. It presents a technique which has a worst case performance that is approximately the lower bound on the performance of a general permutation encoder. It also presents a technique which probabilistically performs better than the lower bound. When permutation information is discarded, our water compression scheme still performs best.

Contents

List of Figures

Chapter 1

Introduction

Molecular dynamics (MD) simulations generate vast amounts of data. A typical 100-million atom MD simulation produces approximately 5 gigabytes of data per frame consisting of atom types, coordinates and velocities. This will generate 17 terabytes of data a day if run for 35 000 steps with every 10^{th} frame being saved [Omeltchenko et al., 2000]. Generating this much data makes compression desirable useful for reducing storage space. A significant overhead in simulations is also transmitting the data from the cluster which performs the simulation to the researchers computer for analysis. Compression reduces the transmission time.

Many MD simulations contain significant amounts of water molecules in them. There are models on how water connect. Using this we have developed a method which compresses MD simulations. It uses a connectivity-based point cloud technique, but with predictors which are based on models of the structure of water.

The method is compared against other point cloud compressors, as well as the Omeltchenko et al. [2000] MD compressor.

This report concentrates on intra-frame compression. Intra-frame compression is where the frame of the MD simulation is compressed independently the other frames. This is versus inter-frame compression, where correlation between frames can be exploited. Inter-frame compression is covered in Julian Kenwood's report.

Research Question This report investigates the implementation and results of an intra-frame MD data compressor which targets water. This compressor is considered successful if:

- It can exploit water models.
- It can improve on other methods. Specifically point cloud compression methods.
- It can compress in an efficient manner. ($O(n^2)$ where n is the number of atoms)
- It can decompress in an efficient manner. ($O(n)$)

Thus, the research question is whether it is possible to implement an intra-frame compressor with the listed properties.

Outline In chapter two background information on MD file formats, compression, point cloud compression and water models are discussed. Chapter three details the design and implementation of the water compressor. Chapter four presents and discusses the results from the implementation. Finally chapter five concludes the report.

Chapter 2

Background

This section starts by introducing the common file format used for storing MD simulation data. It then moves on to the topic of compression which is relevant to the rest of the report. This includes entropy encoding, quantisation and predictive encoding. Point cloud compression algorithms are then presented, as well as a taxonomy of point cloud compressors. The water dimer model is also presented, with information on how it could be applied to existing predictive point clouding techniques.

2.1 Molecular Dynamics Format

There are several Molecular Dynamics Simulations formats in use. The most common format is the binary DCD file. DCD is also the default format used in Visual Molecular Dynamics (VMD) [vmd, 2009a]. VMD is a molecular visualisation program for displaying, animating, and analyzing molecular dynamics simulations. VMD also relies on an accompanying PDB file to display the DCD file.

There are multiple variations on the format of a DCD file, but they all store the points in effectively the same way. VMD can handle two major variants, CHARMm and X-PLOR, but VMD defaults to X-PLOR [vmd, 2009b].

The DCD header contains at least the number of atoms and frames in the simulations. Every atom is assigned an index i between 0 and $\#atoms - 1$. Each frame is then stored as an array of floating point values of size $3 \times \#atoms$. The location of atom i is then stored at indicies $3 \times i$ to $3 \times i + 2$ (storing the x, y and z coordinates). Atom i will always occur at index $3 \times i$ in every frame.

The DCD file only contains the positions of the atoms, with each item only being identified by its index in a frame. Thus the DCD file does not contain the atom's type. Instead this is contained in an accompanying PDB file.

2.2 Compression

Compression is the process of encoding information such that it uses less storage than the unencoded form.

Compression algorithms can be either lossy or lossless. Lossless compression algorithms do not "lose" any information during the encoding and decoding process. An example of lossless compression is the gzip data compressor [Gailly and Adler, 2009]. Lossy compression algorithms can "lose" information during the encoding and decoding stages, but give approximations of the original information. An example of a lossy compression algorithm is DivX for video compression [DivX, 2009].

2.2.1 Entropy Encoders

Entropy encoders attempt to optimally encode symbols as bits. A familiar example of an entropy encoder is the Huffman Encoder. Given n symbols $\{x_1, \ldots x_n\}$ each with corresponding probabilities $\{p_1, \ldots p_n\}$, Huffman Encoding assigns each x_i a binary codeword c_i such that: [Huffman, 1952]

1. $p_i < p_j$ implies that $|c_j| \leq |c_i|$ where $|c|$ is the length of the codeword c.
2. There is no c_i such that it is a prefix of c_j

Huffman Encoding generates an optimal encoding which has the two listed properties. Generation of the code takes $O(n \log n)$ time and $O(n)$ space.

Huffman Encoding has three main shortcomings. The first two problems are that it requires the probabilities of each symbol and it needs to store the mapping for decoding. Both of these problems are remedied by Adaptive Huffman Encoders [Drozdek, 2001, Vitter, 1987]. These start with every symbol having equal probability, and then proceed to efficiently update the probabilities as each symbol is encoded.

The final shortcoming is that each symbol is mapped to a prefix-free binary codeword. The amount of wastage caused by this could be as much as one bit per symbol. Arithmetic Coding is an alternative approach which can encode a symbol as a fractional amount of bits [Drozdek, 2001]. Arithmetic Coding achieves this by encoding the entire message as a rational number in $[0, 1)$. Each symbol x_i is assigned a disjoint subrange of $[0, 1)$ with size p_i.

Encoding is done by adjusting the lower and upper bound of what the encoded rational number will be. Initially the lower bound is $l = 0$ and the upper bound is $u = 1$. Then when encoding a symbol x_i which is assigned the range $[a, b)$ the bounds are adjusted to

$$l' = l + a * (u - l)$$
$$u' = l + b * (u - l)$$

Arithmetic coding has a similiar shortcoming to Huffman Encoding, in that it requires the probabilities of each symbol and needs to store the probabilities. This can be overcome by using an Adaptive Arithmetic Encoder [Drozdek, 2001]. Initially all symbols have equal probability and as each symbol is encoded the ranges are updated.

Implementations of Arithmetic Encoding usually separate out the model and the encoder. The model keeps track of what the probabilities of each symbol are, while the encoder writes out the rational. The encoder does not store the full rational, but instead writes out each bit of the binary decimal expansion.

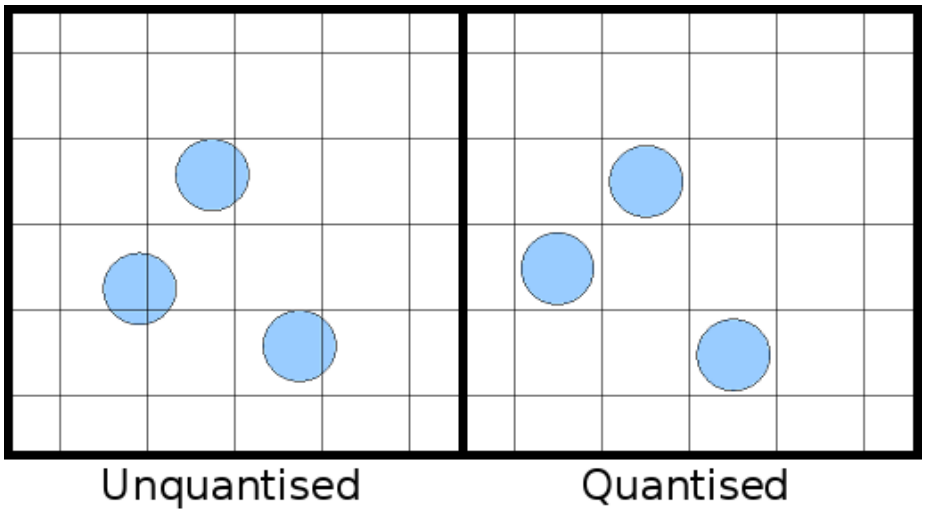

Figure 2.1: The effect of linear quantisation on points.

It can write the bit when the lower bound is such that the bit will not change. Encoding and decoding is $O(1)$.

The static model is just a look-up table of the probabilities, so can be implemented to be $O(1)$. The adaptive model uses Fenwick trees [Fenwick, 1994] to update and query frequencies in $O(\log n)$ time.

2.2.2 Quantisation

The data encoded by the algorithms in this report are usually stored as floating point values. A process called *quantisation* reduces the precision of a floating point value to an integer value [Alliez and Gotsman, 2005]. *Dequantisation* converts the integer value back into an approximation of the original floating point value. Quantisation is inherently lossy.

If the bounds of the data are known, then the range of the data can be uniformly divided up. Then each sample is assigned the index of the range it falls into [Drozdek, 2001]. This is called *linear quantisation*. By storing the bounds and the number of buckets the range is divided up into, original values can be approximated.

Linear quantisation of point coordinates involves quantising each coordinate separately and uniformly in Cartesian space. This can be visualised as creating an evenly-spaced grid over the dimensions of the space, and snapping the points to the closest grid point. Refer to Figure 2.1.

There are more complex methods reviewed by Alliez and Gotsman [2005]. These lie outside the scope of the report.

2.2.3 Predictive Coding

Predictive coders exploit inter-correlation of data to achieve compression [Drozdek, 2001]. If the data in the format is point positions then neighbouring

points are inter-correlated. So using the encoded neighbouring points, a predictive encoder would predict where the point is. The encoder would then encode the difference between the points location and its predicted location (known as the residual).

Linear quantisation preserves the inter-correlation of the points. So if the points are quantised first, then the residuals can be encoded using an entropy encoder such as an arithmetic coder. If the predictor is accurate, the residuals will mostly be small values. So the probability of the small values will be high, leading to good compression.

Using the residuals and the predictions the decoder can reconstruct the point locations.

2.3 Point Cloud Compression

A point cloud is a collection of point locations in 3D space. Research has concentrated on point clouds of surfaces since point clouds are usually generated from 3D scans of object surfaces. A surface is the "skin" of an object, i.e. the boundary of a solid object.

Every point cloud compressor in this report (except Chen et al. [2005]) first quantise the points using linear quantisation. In all the point cloud compressors reviewed, this is the only lossy step. Chen et al. [2005] gets around quantisation by treating the 32-bit floating point numbers as 32-bit integers.

Gumhold et al. [2005] create a tree of the points that exploit the knowledge that the points lie on a surface. They do not, however, use predictors which exploit this. Merry et al. [2006] extend this by using predictors which do exploit the rectilinear nature of surface scans.

Omeltchenko et al. [2000] compress point clouds from molecular dynamics simulations. To exploit the density of the point clouds, a space filling curve is created along the quantised positions. A space filling curve is one that touches every point in a discrete space. The space filling curve used by Omeltchenko et al. [2000] is based on indexing the cells created by an octree division. Then by encoding the difference between successive indicies which contain points, they generate consistently small differences leading to good compression rates.

Devillers and Gandoin [2000] compress meshes of 3D models. A mesh is a collection of vertices, edges and faces defining a 3D object. The method they use only considers point locations, so it can be used for point cloud compression. Their method exploits a property of kd-trees, instead of general geometric properties.

2.3.1 Predictors

A rooted spanning tree of the point is used in both Gumhold et al. [2005] and Merry et al. [2006]. The creation of the spanning tree depends on what predictor(s) are used and will be described in 2.3.2. For each point v in the spanning tree let v' be the point's parent and v'' be the point's grandparent.

Gumhold et al. [2005] lets the user choose between two predictors. The first is a *constant* predictor where v is predicted to be at v'. The second is a *linear* predictor where v is predicted to be at $v' + (v' - v'')$. So either v is in the same

place as its parent, or it is along the straight line (v', v'') with distance $|v' - v''|$ away from its parent.

Merry et al. [2006] use the same predictors as Gumhold et al. [2005] but with two additional predictors, the *left* and *right* predictors. The surface normal[1] n_v at v is heuristically determined from v, v'', v' and $n_{v'}$ (where $n_{v'}$ is the surface normal at v'). From this they determine what left and right are by using the plane determined by $n_{v'}, v'$ and v''. Another difference to Gumhold et al. [2005] is that instead of letting the user choose the predictor used, the best predictor is picked for each point. This requires the predictor used at each point to be encoded, which is discussed next.

2.3.2 Serialisation

Unlike meshes, point clouds have no connectivity information. So there is no obvious order to serialize the vertex information (generally residuals). By creating a spanning tree of the points, serialisation can occur by walking the graph. The tree also needs to be encoded so decoding can recover the tree to do the walk.

Gumhold et al. [2005] create a rooted spanning tree of the points. The points are first sorted along the x, y or z axis. The tree is then greedily created by picking the first point as a root and then adding each successive point to the vertex which minimises the residual. The tree is then serialized by entropy encoding the out degree of each vertex in breadth-first order.

Merry et al. [2006] also create a rooted spanning tree of the points, but in a way that favours "long runs" of the forward predictor. They need to encode the predictor used at each point, so encoding the tree as Gumhold et al. [2005] did will not work. Instead the tree is serialized by entropy coding the predictor used in depth-first order. By favouring long runs of the forward predictor they get better compression ratios of the encoding of the tree. To create the spanning tree, they first create a graph of the points where every edge with length less than L is added. L is the length of the longest edge in the minimum spanning tree of the complete graph of the points. The spanning tree is then created similarly to Prim's algorithm [Sedgewick, 2001, p. 457], but instead using a metric[2] that favours recently considered vertices and ones predicted well by the forward predictor.

Chen et al. [2005] uses a spanning tree with differential coding on the weights. So the point cloud is encoded as a tree, with edge weights being the difference vectors between points. They show that there exists a spanning tree which minimises the number of pairwise different edge weights, but also show that this is NP-Hard. They do, however, give an approximate algorithm which uses clustering and minimum spanning trees. The tree is serialized in a simple breadth-first order, recording the number of children for each node and the edge weights for each child.

[1] A surface normal at a point v is the normal of the plane tangent to the surface at v.

[2] A metric is a way to calculate the distance between vertices.

2.4 Decoding

When decoding the data, we may present to the user intermediate representations of the final decompressed data as we process it. This is a desirable property if the user can use the intermediate representations while (s)he waits for the final representation to be transferred/decompressed. Depending on whether we can do this or not gives us another way to classify encoders.

Progressive Encoder A progressive encoder is one that starts out streaming a coarse representation of a point cloud. It then streams out refinements. This is useful for transmitting the data over a network, as users can almost instantaneously see a coarse representation. This is then followed by more and more refinements over time as they arrive.

The use case of MD simulations require the whole file (or frame) to be decompressed. So progressive encoders have no advantage, and are judged purely on how well they compress the data.

Devillers and Gandoin [2000] provide an example of a progressive coder. The method is best explained in one dimension first. If you know the total number of points in $[a, b]$ (say x points) and the number of points in $[a, (a + b)/2]$ (say y points) then the number of points in $[(a + b)/2 + 1, b]$ is $x - y$. By sending out counts of deeper and deeper levels in breadth first order they progressively get a better representation. Encoding this with arithmetic coding gives us the desired compression ratios. To extend to 3 dimensions one uses a process similiar to creating a kd-tree: at each step subdivide along a different dimension.

Single-Rate Encoding A single-rate encoder is one which requires the whole stream to be available before the data can be decompressed. Omeltchenko et al. [2000], Gumhold et al. [2005] and Merry et al. [2006] are all single-rate encoders. In general single-rate encoders imply better compression.

2.5 Modelling Water

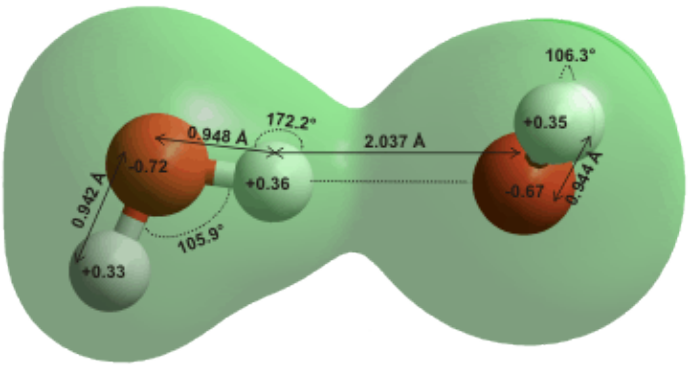

Figure 2.2: The most energetically favourable water dimer calculated from first principles. [Chaplin, 2009]

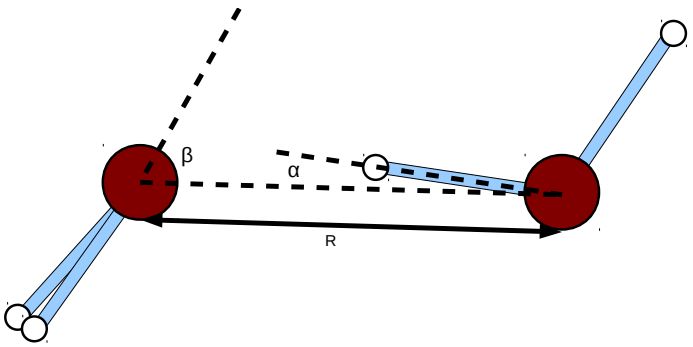

Figure 2.3: Water dimer angles. $R = 2.976$Å, $\alpha = 6 \pm 20°$, $\beta = 57 \pm 10°$.

The previously discussed predictors work well in the setting of general point clouds, but in a point cloud consisting of water molecules we can create predictors tailored towards water.

A water dimer is two water molecules loosely bonded by a hydrogen atom. This is the simplest model for hydrogen bonding in water, and as such has been extensively studied.

The oxygen which is loosely bonded to the hydrogen is expected to be 2.976Å away from the other oxygen. It is also expected that the vector created by the O-H bond to be roughly in line with the vector of the loose O-H bond [Chaplin, 2009]. See Figure 2.2 and Figure 3.1.

2.5.1 Using the Model in Point Cloud Compressors

Omeltchenko et al. [2000] only exploit the fact that there may be many dense clusters in MD simulations. It is more general than water compression, and how to modify it to use the relative layouts of the water molecules is unclear.

The predictors of Merry et al. [2006] and Gumhold et al. [2005] do not exploit the layout of water molecules. Using water dimer model prediction can be done given the positions of a water molecules atoms. The creation of the spanning tree would be different, since we know the expected degree of the verticies is at most two.

2.6 Taxonomy of Techniques

Refer to Figure 2.1 for an overview of the categorisations of the different papers. There are two underlying philosophies which the compression algorithms exploit: *scope* and *topology*.

Scope is whether the algorithm is exploiting *global* or *local* properties of the points. Merry et al. [2006] uses a local scope since it exploits nearby points to predict the location of the current point. Chen et al. [2005] uses a global scope, since it is trying to minimise a global property of the spanning tree.

Topology is whether the algorithm is trying to exploit connectivity or geometric information. Gumhold et al. [2005] exploit both, but the geometry more since it uses a predictor to predict the location based on other points location. Chen et al. [2005] just tries to choose the best possible spanning tree, so they exploit connectivity of points.

Attribute/Method	Omeltchenko et al. [2000]	Gumhold et al. [2005]
Structure	MD Simulation	Surface
Progressive	Single-rate	Single-rate
Quantisation	Yes	Yes
Connectivity	Space Filling Curve	Spanning Tree
Exploits	Dense/regular clusters	Regularity of scans
Scope	Global	Local
Topology	Geometry	Both
Attribute/Method	Chen et al. [2005]	Devillers and Gandoin [2000]
Structure	3D Model	Mesh
Progressive	Single-rate	Progressive
Quantisation	No	Yes
Connectivity	Spanning Tree	kd-trees
Exploits	Differential Coding	kd-tree properties
Scope	Global	Global
Topology	Connectivity	Neither
Attribute/Method	Merry et al. [2006]	
Structure	Surface	
Progressive	Single-rate	
Quantisation	Yes	
Connectivity	Spanning Tree	
Exploits	Rectilinear scans	
Scope	Local	
Topology	Both	

Table 2.1: Taxonomy of the different methods.

The other properties listed in the taxonomy are *structure*, *progressive*, *quantisation*, *connectivity* and *exploits*.

- *Structure* is what type of point cloud the compressor targeting.
- *Progressive* is whether the compressor is a single-rate encoder or progressive encoder.
- *Quantisation* is whether the compressor applies quantisation before compression.
- *Connectivity* is how relationships between points are derived or modelled.
- *Exploits* is what the compressor is targeting for good compression.

2.7 Summary

The water dimer model gives a good predictor for exploiting the local structure of water molecules. So, using a predictive point cloud encoder similiar to Merry et al. [2006] and Gumhold et al. [2005] should result in good compression of water molecules than previous methods.

Compressing the other atoms will require a more general technique. Using the best point cloud compressor on the other atoms combined with the water prediction scheme should result in better compression.

Chapter 3

Design and Implementation

This chapter outlines the design and implementation of our compressor and decompressor. The compression techniques investigated only cover intra-frame compression, where compression occurs per frame independently of the other frames. Reference implementations of other schemes, inter-frame compression as well as visualisation, were implemented as independent subsystems, as such the design needs to take into account these uses. Components outside of the scope of intra-frame compression will not be covered in detail.

The main contribution of this report is a predictive encoder targeting water molecules. The compressor is referred to as the Water Compressor.

3.1 Overview

Compression is done on the DCD/PDB format, which is a format for storing MD simulations. DCD files are explained in Section 2.1.

A high-level overview of how data flows in the compressors is illustrated by the following:

$$DCDFile \rightarrow Atoms \rightarrow QuantisedAtoms \rightarrow Compressor$$

The decompressor works in the reverse direction:

$$Decompressor \rightarrow QuantisedAtoms \rightarrow DCDFile$$

The decompressed DCD files are an approximation due to quantisation. Excluding this quantisation step, the schemes are lossless. Note that each step in the compression phase has auxillary information. The auxillary information is required for reconstruction.

Reference implementations of Devillers and Gandoin [2000], Gumhold et al. [2005] and Omeltchenko et al. [2000] are also implemented. These use the I/O and quantisation components. Julian Kenwood implemented Omeltchenko et al. [2000], while Keegan Carruthers-Smith implemented Devillers and Gandoin [2000] and Gumhold et al. [2005]. Visualisation uses the I/O and quantisation components as well and was implemented by Min-Young Wu.

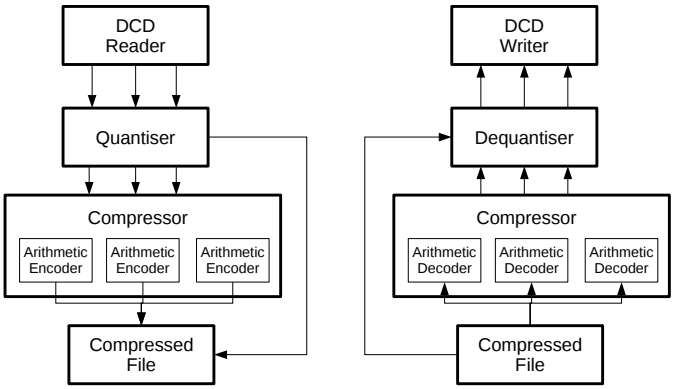

Figure 3.1: A high level overview of how data travels through the compressor and decompressor.

3.1.1 Dependencies

All development was done on Linux, but no Linux or POSIX specific APIs where used. The implementations should be cross platform, but this has not been tested.

C++ A language which produces fast and optimised code is necessary since run time is important in compression. All of the authors are competent in C++, thus all the components are written in C++. Development was done with GCC 4.x.

ANN This is a library for approximate nearest neighbor searching. It is used to speed up graph creation where locality of points is important, [Mount and Arya, 2006].

VMD DCD Loader Plugin This plugin is used to read and write DCD files, [vmd, 2009a].

3.2 Components

A component-based approach was used to design the compressor and decompressor, since the algorithm can be broken up into distinct steps.

The components are as follows:

Simulation I/O These components handle I/O to and from the simulation data files. The VMD DCD loader plugin is used to load and write the DCD files. A PDB reader for determining what each atom is in the DCD file is also implemented. DCD is used since it is the prevalent format for storing simulation data, and it is well supported in VMD.

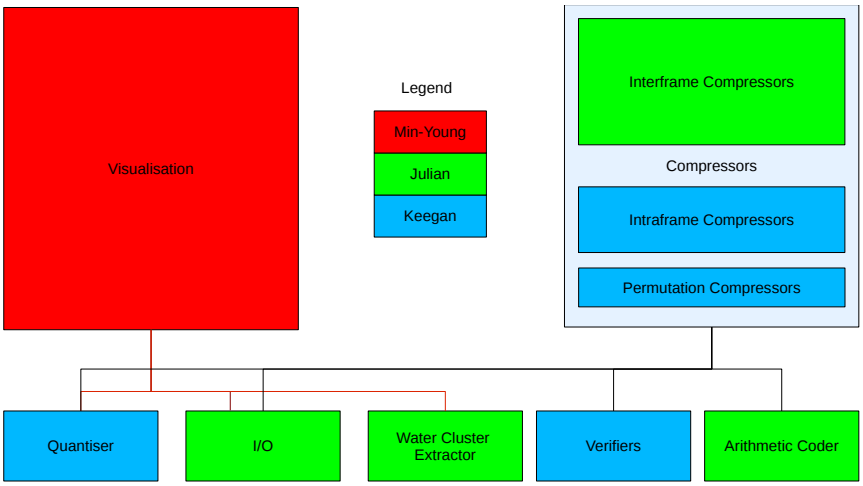

Figure 3.2: Breakdown of work amongst group members.

Arithmetic Coder The water compression scheme as well as the point cloud schemes use arithmetic encoding of the residuals and tree. The range of the residuals in the predictive schemes as well as the number of different residuals is not known beforehand. As such the usual model used in arithmetic encoding does not fit.

For the model we use a novel approach which is based the on adaptive arithmetic model. It uses similiar to the technique by Cormack and Horspool [1984] used in Adaptive Huffman Encoding. The adaptive model initially only has one symbol meaning "new" symbol. When encoding if there is a new symbol, the "new" symbols is encoded followed by a raw representation of the new symbol. By using a prefix tree and a modification of Fenwick trees [Fenwick, 1994], insertion of new symbols into the model is $O(n \log n)$ where n is the number of seen symbols. A prefix tree is a map data structure which gives $O(m)$ look-ups and insertions, where m is the length of the key string. It was used because it is the fastest possible string look-up structure.

The encoder by Omeltchenko et al. [2000] does not make use of an Arithmetic Coder, instead it uses its own entropy coder. Every other encoder uses the arithmetic encoder.

Quantiser The quantiser is responsible for quantising and dequantising a frame. The quantiser takes in as input a number b. The quantiser quantises each dimension independently using linear quantisation. The number of cells it divides each dimension into is 2^b. So after quantisation each coordinate becomes a b-bit integer.

Linear quantisation is chosen since the water compressor (as well as the other predictive compressor) requires the locality of the points to be preserved. Quantisation is also necessary since small differences in predictive residuals should be treated as the same for better entropy encoding.

Converting the floating point values to integers introduces error. But for the predictive schemes to be effective it is necessary. This is the only step which is

lossy. In the literature if the only lossy step is quantisation, the algorithm is still referred to as lossless.

Permutations When decompressing we need to recover the atoms in the same order they are listed in the original DCD file. The water compressor as well as the other predictive point cloud schemes do not recover the original order. How permutations are encoded is explained in Section 3.3.

Compressors The compressor component is different for each compressor. It implements the encoding and decoding of the quantised frames. The intra-frame compressors which are implemented are the water compressor and reference implementations of Omeltchenko et al. [2000], Gumhold et al. [2005] and Devillers and Gandoin [2000].

Verifiers Verifiers are for testing whether a compression scheme is working. Given a DCD file and a decompressed DCD file, if the files match when quantised then the compressor is considered functional. There is the possibility of floating point error causing a mismatch when the files are in fact equivalent. This happens infrequently, so the frames which do not match can be inspected manually to see if they are the same.

3.3 Compressing Permutations

Compression algorithms explored in this report do not necessarily preserve the order of the points when decompressed. For example in Devillers and Gandoin [2000]

$$(1,2),(1,3),(0,0),(2,3),(2,2),(0,2),(1,1)$$

when decompressed becomes

$$(0,0),(1,1),(0,2),(1,2),(1,3),(2,2),(2,3)$$

When decompressing we need to recover the original order of the points, since the index of a point indicates which atom it is in the DCD file format.

We experimented with five different types of permutation compressors. Each permutation compressor takes in a permutation of $[0,1,\ldots,N-1]$ and encodes it to the file. Let the permutation be $[P_0,P_1,\ldots P_{N-1}]$. The decompressor then recovers the permutation.

Null Null does not write anything to the file. The decompressor just outputs the order permutation $[0,1,\ldots,N-1]$. A compressor which uses the null compressor does not preserve the order. As such the null is only used for reference.

Naïve Naïve writes the permutation list straight to the file. Each index is stored as an unsigned 32-bit integer. Every other permutation compressor should do better than this.

Delta Delta transforms the permutation into a list L, where $L_0 = P_0$ and $L_i = P_i - P_{i-1}$ for $0 < i < N$. Arithmetic encoding is then done on L, which is then written to file.

Decompression gets back the list L. Since $P_0 = L_0$ and $P_i = L_i + P_{i-1}$ for $0 < i < N$ we can recover the original permutation. Since each P_i depends only on previous i's, calculating P_i in increasing order of i will recover P.

Interframe Interframe tries to exploit the correlation of the permutation lists between frames. Interframe transforms the permutation P into a list L. Let P' be the permutation from the previous frame. If it is the first frame, let $P' = [0, 1, \ldots, N-1]$. Then $L_i = P_i - P'_i$ for $0 \leq i < N$. Arithmetic encoding is used on L and it is then written to file.

Decompression recovers the list L. Since $P_i = L_i + P'_i$, P can be recovered. Note that this method requires frames to be decompressed in order.

Using this scheme causes an intra-frame encoder to become interframe. The objective of this report is constructing an intra-frame compressor, but this is still experimented with since the permutation encoding ended up being a significant portion of the compressed file. The results indicated that this scheme did not perform best, as such the best encoders remain intra-frame.

Optimal Each P_i occurs only once, so the frequency of each P_i is known per encoding. Optimal uses a static arithmetic coder with a frequency of one for each P_i.

A static arithmetic encoder model uses a Fenwick Tree [Fenwick, 1994] to update the symbol table after each encoding. This takes $O(\log(N))$ per update. In the optimal permutation compressor we can get this down to $O(1)$ by using the fact that we only ever encode each P_i once.

To encode a symbol i the arithmetic encoder requires S_{tot}, S_i and S_{i-1}. S_{tot} is the sum of all the frequencies. S_i is the sum of the frequencies for all the symbols less than or equal to i. Since the frequency of each index is one, $S_i = i$ and S_{tot} is the number of permutations left. We need a way to map permutation indicies to symbols which is quick to update once an index has been encoded. The following algorithm is used:

```
init(permutation):
  number_of_symbols = permutation.size
  for i in 0, 1, ..., permutation.size - 1:
    index_to_symbol[i] = i
    symbol_to_index[i] = i

pop_index(index):
  number_of_symbols--

  symbol = index_to_symbol[index]
  index2 = symbol_to_index[number_of_symbols]

  index_to_symbol[index2] = symbol;
  symbol_to_index[symbol] = index2;

  return symbol;
```

	index_to_symbol				*symbol_to_index*				P_i *left*				*Value*
	0	1	2	3	0	1	2	3	0	2	3	1	
pop_index(0)		1	2	0	3	1	2			2	3	1	0
pop_index(2)		1		0	3	1					3	1	2
pop_index(3)		0			1							1	0
pop_index(1)													0

Table 3.1: Trace of values for the optimal encoder when encoding the permutation $[0, 2, 3, 1]$.

```
pop_symbol(symbol):
  index = symbol_to_index[symbol]
  pop_index(index)
  return index
```

To encode P_i, we encode *pop_index*(P_i). To decode, we return *pop_symbol*(*val*) where *val* is a value returned from the arithmetic decoder. An example of encoding is given in Table 3.1.

The optimal permutation encoder is optimal in the worst case. If we have M indicies left to encode, each unseen index has probability $\frac{1}{M}$ of occurring and every seen index has probability 0 of occurring. So for each i we have that $S_{tot} = M, S_i = i, S_{i-1} = i - 1$. This gives a probability of $\frac{i-(i-1)}{M} = \frac{1}{M}$.

If there are M indicies left, the lower-bound on the number of bits to encode an index is $\log_2(M)$. So if P has N indicies a lower-bound on the number of bits to encode P is

$$\sum_{i=1}^{N} \log_2(i) \tag{3.1}$$

Each index can fit into $\lceil \log_2(N) \rceil$ bits, so this is used as an effective upper-bound. See Table 3.2 for values of Equation (3.1) in bytes. Notice how as the size of the permutation increases, the gains made by the optimal scheme decrease.

Permutation Size	1	2	10	1000	10 000	100 000
Lower-Bound	0.00	0.12	2.72	1066.17	14807.26	189588.02
Effective Upper-Bound	0.00	0.25	5.00	1250.00	17500.00	212500.00
Percentage Gain	0.00	33.33	29.47	7.94	8.33	5.70

Table 3.2: Number of bytes needed for storing permutations of different sizes. See Equation (3.1) for the lower bound. The effective upper-bound is $\lceil \log_2(N) \rceil / 8$.

The optimal encoder is optimal since on its worst permutation it does better than any other encoder, but most of the encoders have a low probability of encountering a bad permutation. There may also be inter-correlation amongst subsets of the permutation. So probabilistically the *delta* encoder will do better than the *optimal* encoder. If the permutations between frames are correlated, then the *interframe* encoder will likely do better than the *optimal* encoder.

3.4 Water Compressor

This section details the algorithm of the water compressor and is the main contribution of this report. The encoder is implemented as a compressor component and uses the components detailed in the Section 3.2.

The compressor processes each quantised frame in order. This encoder is a predictive intra-frame encoder, and as such only compresses using information contained in the frame.

The algorithm is similiar to the predictive point cloud compressors of Gumhold et al. [2005] and Merry et al. [2006]. The following is a high level overview of the algorithm:

```
CompressFrame(list_of_atoms_in_frame, arithmetic_encoder):
  list_of_water_molecules = FindWater(list_of_atoms_in_frame)
  list_of_other_atoms = FindNonWater(list_of_atoms_in_frame)
  compress(list_of_other_atoms, arithmetic_encoder)
  water_graph = CreateGraph(list_of_water_molecules)
  spanning_tree = CreateSpanningTree(water_graph)
  TreeSerialise(spanning_tree, arithmetic_encoder)
```

Similarly decompression uses this algorithm:

```
DecompressFrame(arithmetic_decoder):
  list_of_other_atoms = decompress(arithmetic_decoder)
  list_of_water_atoms = TreeDeserialise(arithmetic_decoder)
  return list_of_other_atoms + list_of_water_atoms
```

The compression of non-water atoms simply involves writing out the quantised positions of the frame. This is done for two reasons: the permutation information is stored for "free" and the number of other atoms is expected to be small.

An explanation of each step follows, along with motivation of why each step is done.

3.4.1 Finding Water Molecules

The PDB file associated with a DCD file contains information on which atoms a given atom is bonded to. Using the PDB file the encoder finds all oxygen atoms which are bonded to two hydrogen atoms. This section was implemented in the Simulation I/O component.

The implementation returns two lists: a list of water molecule indicies and a list of other atom indicies. Every water molecule is specified using the indicies in the frame of its corresponding oxygen and hydrogen atoms. Every index which is not in a water molecule is stored in the second list.

The split between water molecules and other atoms is done so that compression of water molecules and the other atoms can be done using different encoders. The water compression predictors target water atoms, so a more general compression method is used on the other atoms.

3.4.2 Water Predictors

The predictors are based on the models of water dimers explained in the background chapter. There are three predictors used, a constant predictor and two hydrogen predictors.

When prediction is applied a tree is made of the water molecules. Then each molecule is predicted from its parent in the tree. Let O, H_1, H_2 be the predictions for where the water molecule will be and let O', H_1', H_2' be the water molecules parent.

Constant Predictor The constant predictor predicts O to be at O'. This predictor is included since occasionally the hydrogen bond is broken (molecule information remains static from the start of the simulation). If the bonds are broken the hydrogen predictors are far less accurate than the constant predictor.

Hydrogen Predictor The hydrogen predictor predicts along $O' - H_1'$ or $O' - H_2'$. We simplify the model in Figure 3.1 and assume α is 0°. Then O is predicted to be at $O' + 2.976\frac{O'-H_i'}{|O'-H_i'|}$.

In all three prediction schemes H_1 and H_2 are predicted to be at O. This is since their is a large variability in β in Figure 3.1.

3.4.3 Creating the Water Graph

Creating the spanning tree is $O(E \log E)$, where E is the number of edges in the graph. When creating the spanning tree, each molecule's predictions are compared to all of its neighbours. If the graph created is the complete graph, E is $O(N^2)$, where N is the number of water molecules. However, this is too inefficient for large scale simulations.

So the purpose of graph creation is to efficiently minimise the degree of the verticies (and hence E), while maintaining the neighbours which predict well against the predictors.

The neighbours which predict well are related to the spatial locality of the oxygens. So the approximation used is to add edges between all molecule pairs where the oxygens are within 3Å. The degree of a vertex is then a small number (typically less than five) and the number of edges created is usually $O(N)$.

The naïve implementation of graph creation would compare each molecule against every other, yielding a $O(N^2)$ algorithm. Again this is too slow. Instead a kd-tree [Cormen et al., 2001] is used to create the graph. A kd-tree is a space-partitioning data structure which allows quick queries based on locality of points. The kd-tree is populated with the oxygen positions. Then using ANN's approximate fixed radius search one can find all other oxygens within approximately 3Å. Creating the graph requires $O(N \log N)$ time. The fixed radius search takes approximately $O(\log N)$ time. The search is done per molecule, so the overall time complexity for graph creation is $O(N \log N)$. The algorithm is as follows:

```
createGraph(list_of_water_molecules):
  graph = graph of water molecules
  kdtree = KDTree(list_of_water_molecules.O_pos)
```

```
  for cur_mol in list_of_water_molecules:
    radius = 3 angstroms
    for mol in kdtree.fixed_radius_search(cur_mol.O_pos, radius):
      graph.addEdge(cur_mol, mol)
  return graph
```

3.4.4 Creating the Spanning Tree

The order in which the points are serialised, and which predictor is used for each point is not obvious from the graph created in the previous section. A directed tree is created from the graph since each molecule is predicted by one other molecule. The edge $O' \to O$ indicates that O is predicted using the position of O'.

The graph is not necessarily connected, so a spanning tree is created for each component. A root vertex is arbitrarily selected. Then each tree in the forest is connected to the root with the constant predictor. Each edge $O' \to O$ is labelled with what predictor is used to predict O from O'.

For each component a root is arbitrarily picked. The tree is then created by using a modification of Dijkstra's shortest path algorithm [Dijkstra, 1959]. The priority queue stores potential edges, and sorts by the residual from the prediction. When a potential edge $O' \to O$ is removed from the queue, it is added to the spanning tree if O is not already in it. The algorithm is illustrated by the following:

```
create_spanning_component(graph, tree, component_root, root):
  priority_queue q
  q.add(0, component_root -> root, constant)
  while q not empty:
    residual, v' -> v, p = q.pop()

    if v not in tree:
      tree add edge v' -> v with predictor p

    predictions = { constant_predictor(p),
                    h1_predictor(p),
                    h2_predictor(p) }

    for u in graph[v].children:
      if u in tree:
        continue

      residual, predictor = smallest_residual(u, predictions)

      q.add(residual, v -> u, predictor)
```

This algorithm is $O(E \log E)$ since for every edge in the graph, a potential edge is added to the queue. In the implementation an optimisations is used which reduces the complexity to approximately $O(N \log E)$. In this optimisation the smallest seen residuals is kept for each molecule. Then a potential edge $v \to u$ is only added if the residual is less than the current smallest residual for u.

3.4.5 Tree Serialisation

This stage encodes the predictions and spanning tree. The predictions are encoded in breadth first order. The spanning tree is also encoded since it is required to reconstruct what each molecule is predicted from. The BFS then starts at the root of the spanning tree and for each molecule in the BFS the following happens:

```
process(O' -> O, predictor):
  prediction  = prediction(O', predictor)
  O_residual  = water_molecule.oxygen - prediction
  H1_residual = water_molecule.hydrogen1 - water_molecule.oxygen
  H2_residual = water_molecule.hydrogen2 - water_molecule.oxygen

  residual_encoder.encode({ O_residual, H1_residual, H2_residual })
  permutation_encoder.encode(water_molecule.index)

  for pred, child in O.children:
    tree_encoder.encode(pred)
    bfs_queue.push(O -> child, pred)
  tree_encoder.encode(sentinel)
```

The root does not have a predictor, so by default the constant predictor will predict the position to be $(0, 0, 0)$.

Deserialisation The tree deserialisation is the reverse of the serialisation. It populates a quantised frame with the following algorithm:

```
q = queue()
q.push(NULL, constant_predictor)

while q not empty:
  parent, predictor = q.pop()

  prediction  = predict(parent, predictor)
  O_position  = residual_decoder.decode() + prediction
  H1_residual = residual_decoder.decode() + O_position
  H2_residual = residual_decoder.decode() + O_position

  index = permutation_decoder.decode()
  water_molecules[index].oxygen    = O_position
  water_molecules[index].hydrogen1 = H1_position
  water_molecules[index].hydrogen2 = H2_position
```

Since this is done for each atom once per frame, decoding takes $O(N)$ time.

3.5 Summary

This section presented the design of our components and our water compression scheme. Our water compression scheme is similiar to those by Merry et al.

[2006] and Gumhold et al. [2005], but with predictors tailored towards water. The tree serialisation is similiar to Merry et al. [2006], except with Breadth First traversal instead Depth First. The spanning tree creation is also different to both, using a more optimal (but slower) algorithm for minimising residuals.

The complexity of encoding in our water compression scheme is $O(n + n \log n + e \log e + n) = O(e \log e)$ where $e = O(n^2)$ and n is the number of water molecules. Empirically and due to the theoretical models of water, e is typically less than $5 \times n$, and so $e = O(n)$. So the encoding time is $O(n \log n)$. The decoding time is $O(n)$.

Chapter 4

Results

4.1 Experiment Design

Testing was done on an Intel Quad Core 2.66GHz machine with 4GB of RAM. Each encoding scheme was tested against 6 DCD files at 8-bit and 12-bit quantisation. 12-bit is the most common level of quantisation used in other point cloud compressors. 8-bit was also tested since this is the lowest acceptable level of quantisation [Wu, 2009].

Each encoding scheme was tested with all the permutation encoders listed in Section 3.3. When showing the result of an encoding scheme, the *null* permutation encoder is shown. The best performing permutation encoder, excluding *null*, for that run is also included.

Two additional compressors are also tested. These are straight quantisation (referred to as *quant*) and quantisation followed by the data compressor gzip (referred to as *gzip*).

The encoder by Gumhold et al. [2005] has two possible predictors to use, the *constant* predictor and the *linear* predictor. The results here indicate only the use of the *constant* predictor since it always did better than the *linear* predictor. This is due to a non-linear relationship between points in molecular dynamics data, unlike rectilinear scans.

4.2 Datasets

Six different DCD files were used in the experiments. Every DCD file excluding MSCL was generated by Julian Kenwood using the Nanoscale Molecular Dynamics (NAMD). A summary of each file is given in Table 4.1.

- **Small Water** is a simulation containing 233 water molecules. The simulation is run at room temperature. The file contains every frame of the simulation.
- **Water** is a simulation of 32 301 water molecules. The simulation is run at room temperature (25°C). The file contains every 50^{th} frame of simulation.
- **Hot Water** is the same simulation as *Water*, but at 1 000°K.

Name	*Size*	*Atoms*	*Frames*	*% Water*	*Comment*
smallwater	0.8	699	100	100%	Small water box
rabies	11	464 099	2	68%	Rabies virus with two areas of water
hiv	38	16 470	200	93%	HIV in water
hotwater	222	96 603	200	100%	Water at 1 000°K
water	244	96 603	220	100%	Water at room temperature
mscl	382	111 016	300	61%	Mechanosensitive Channel

Table 4.1: Description of DCD files used in compression tests. *Size* is the size of the DCD file in megabytes. *Atoms* is the number of atoms per frame. *Frames* is the number of frames in the DCD file. *% Water* is the percentage of atoms which are part of a water molecule.

- **HIV** is a simulation of a part of the Human Immunodeficiency Virus(HIV) surrounded by water. It is simulated at room temperature (25°C). The file contains every 50^{th} frame of simulation.
- **Rabies** is a simulation of the rabies virus. The simulation is run at room temperature. The file contains every frame of the simulation.
- **MSCL** is a simulation of a protein separating two volumes of water. The protein is called Mechanosensitive Channels. The output rate is unknown, but inspection of the simulation in VMD indicates erratic movement of the atoms. Hence the output rate is probably low.

The two most important attributes of the simulation are the temperature and the number of atoms. The number of atoms affects the run time of the schemes per frame. The temperature affects the accuracy of the water model. The higher the temperature, the worse the expected performance of the water compressor. This is since the performance of our water compressor is directly related to how accurate the water dimer model is. At high temperature the accuracy of the water dimer model decreases.

4.3 Compression Results

For each run the compression size and time were recorded. The decompression time where also tested, but are not listed. This is since every schemes excluding Devillers and Gandoin [2000] has linear decompression complexity. Devillers and Gandoin [2000] has $O(n \log n)$ decompression complexity, which is the same as the compression complexity, so times are similiar. The results for 8-bit and 12-bit compression are presented separately.

4.3.1 8-bit Quantisation

Refer to Table 4.2 for the 8-bit compression ratios. The results indicate that in general our water compression scheme performs best. *Water*, *hotwater* and *hiv* all contain large amounts of water. This accounts for the significant gain over the other schemes that water compression achieves.

Our water compression scheme performs poorly on *smallwater* even though it contains only water molecules. Table 4.4 indicates that this occurs since the

DCD	atoms	quant	gzip	omel	gumhold		dg		watercomp		
					null	*best*	*null*	*best*	*null*	*best*	*gain*
smallwater	699	25.35	**12.30**	30.00	19.34	27.66	21.74	26.45	18.65	21.75	−9.45
water	96 603	25.00	21.47	27.72	11.20	26.00	12.86	24.10	*9.24*	**14.63**	6.84
hotwater	96 603	25.00	19.08	26.43	10.48	25.56	11.31	22.67	*9.99*	**15.33**	3.76
rabies	464 099	25.00	**13.86**	27.78	*9.28*	25.69	10.17	22.05	14.18	19.44	−5.59
hiv	16 470	25.02	24.12	27.96	13.89	26.98	15.81	26.67	*12.10*	**16.26**	7.86
mscl	111 016	25.00	20.17	27.69	*11.22*	25.89	12.41	23.39	15.94	**19.13**	1.04

Table 4.2: Compression rates at 8-bit quantisation. The best ratio is bolded. The *null* permutation values are also included, but are not considered since they do not restore the correct order of points. *Best* is the permutation encoder which did the best out of *delta*, *interframe*, *naïve* and *optimal*. *Gain* is the amount of compression ratio gained against the next best scheme.

constant predictors are used nearly as often as the hydrogen predictors, hence the hydrogen predictors are not predicting well. Additionally the file itself is very small causing the overhead of the arithmetic encoder's symbol tables to be significant.

Surprisingly *gzip* performed best in two cases, *smallwater* and *rabies*. At 8-bit quantisation the points are written to file and are byte-aligned, causing gzip to more likely pick up correlation among the points. In the two cases where gzip performed well, the quantised data would of had lots of repetition and correlation.

If the ordering of the points is not important, then our water compression scheme performs best (see the null columns). *Gumhold* performs better in *rabies* and *mscl*, both of which contain significantly less water molecules than the other test cases. Since there are less water molecules, *watercomp*'s hydrogen predictors would of not been as effective leading to the worse compression.

The reason *gumhold* does not perform best in *rabies* and *mscl* with ordering is that our water compression scheme stores a smaller permutation. *Watercomp*'s permutation size is the number of water molecules while *gumhold*'s permutation size is the number of atoms.

Refer to Figure 4.1 for the 8-bit compression speeds. This graph indicates that our water compression algorithm performs relatively poorly in terms of speed. This is due to the $O(e \log e)$ spanning tree creation step. All other algorithms are $O(n \log n)$ or $O(n)$ where n is usually an order of magnitude less than e. However, the algorithm is still efficient since it is sub $O(n^2)$.

The worst running time for our water compression algorithm is 693 seconds on the *water* test case. This further indicates that it may be worse relatively, but still acceptable. The algorithm performs best in terms of compression ratio, so the trade-off is valid.

4.3.2 12-bit Quantisation

See Table 4.3 for the 12-bit compression ratios. The results indicate that our water compression scheme performs best for all the DCD files. The gain over the next best scheme increased in all cases when compared to the 8-bit quantisation results. This is due to more accurate quantisation causing the hydrogen predictors to be more accurate.

Our water compression scheme also performs best when ordering is not

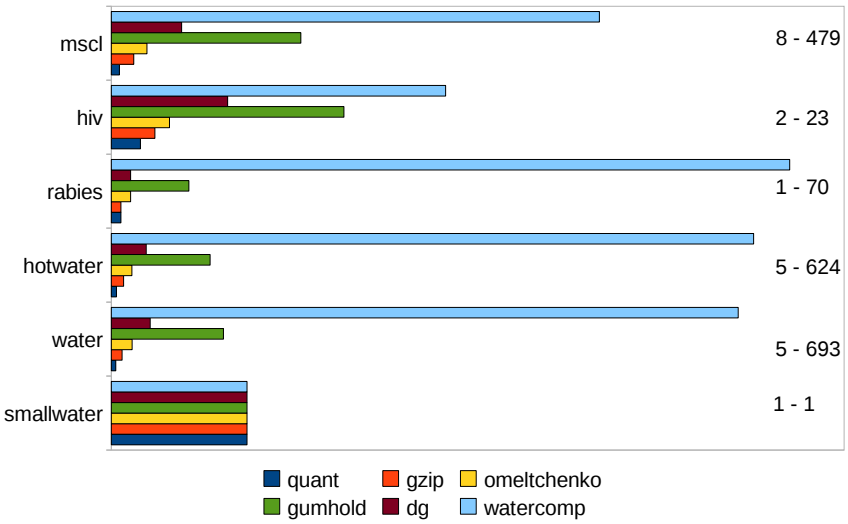

Figure 4.1: The relative time taken for each compression scheme at 8-bit quantisation. The numbers on the right indicate the quickest and longest time in seconds for the each case.

DCD	atoms	quant	gzip	omel	gumhold		dg		watercomp		
					null	*best*	*null*	*best*	*null*	*best*	*gain*
smallwater	699	37.83	37.39	43.86	32.75	41.07	34.99	39.69	*32.60*	**35.20**	2.19
water	96 603	37.50	37.26	40.23	23.62	38.40	26.50	37.74	*20.84*	**26.23**	11.03
hotwater	96 603	37.50	36.82	38.92	22.95	38.03	25.19	36.54	*21.50*	**26.89**	9.65
rabies	464 099	37.50	32.92	40.27	*21.64*	38.00	24.12	36.00	25.89	**31.03**	1.89
hiv	16 470	37.51	37.52	40.51	26.30	39.39	29.25	40.10	*24.49*	**28.65**	8.86
mscl	111 016	37.50	37.19	40.19	*23.63*	38.22	26.06	37.04	28.01	**31.22**	5.82

Table 4.3: Compression rates at 12-bit quantisation. The best ratio is bolded. The *null* permutation values are also included, but are not considered since they do not restore the correct order of points. *Best* is the permutation encoder which did the best out of *delta*, *interframe*, *naïve* and *optimal*. *Gain* is the amount of compression ratio gained against the next best scheme.

important (see the null column). Similarly *gumhold* performs better in *rabies* and *mscl*. The reason for this can also be attributed to both files containing less water molecules. If the ordering of the points is not important, the best scheme is Gumhold et al. [2005] (see column gumhold null).

12-bit quantisation is not byte-aligned which causes the quantised data to appear completely different to its 8-bit version. This is most likely the cause for *gzip* performing poorly on *smallwater* and *rabies* compared to the 8-bit results.

See Figure 4.2 for the 12-bit compression speeds. This graph is nearly identical to the 8-bit graph (4.1). The difference is that our water compression scheme has run times closer in speed to the other schemes. This occurs because the run-time is dominated by the spanning tree creation, which is not affected by the granularity of the quantisation.

The worst increase in run time for our water compression scheme from 8-bit to 16-bit is on MSCL. It increased by 21 seconds, from 479 seconds to

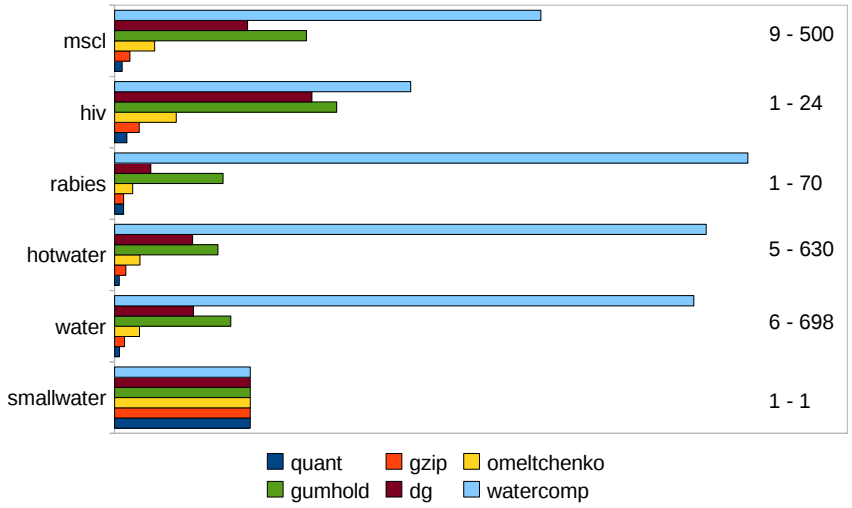

Figure 4.2: The relative time taken for each compression scheme at 12-bit quantisation. The numbers on the right indicate the quickest and longest time in seconds for the each case.

500 seconds. The increase in time is probably due to the increased number of different residuals due to the higher precision. If the number of residuals increases, the performance of the arithmetic encoder decreases (but the decrease is logarithmic, which can be largely ignored).

4.4 Prediction Results

Refer to Table 4.4 for statistics on prediction and water clusters. The number of water clusters is the number of components created at the graph creation stage. The *constant* and *hydrogen* columns indicate the number of molecules predicted by the respective predictors.

The graph creation stage adds an edge between all atoms which are approximately within 3Å (once quantised). This causes a number of components to occur in the graph. Each component corresponds to a cluster of water atoms. The number is important since the hydrogen predictors only work on the molecules within a cluster. The joining of clusters is done naïvely, so if the cluster number is high (relative to the number of water molecules) this will negatively affect compression.

The results indicate that the difference in quantisation has a significant affect on the number of water clusters. This is expected since the maximum increase in distance between two atoms is related to how high the quantisation is. In this case atoms which where 3Å away from each other, once quantised where further than 3Å away. This is an indicator that 8-bit quantisation may be too aggressive. Since the accuracy exponentially increases as the quantisation level increases, values above 12-bit quantisation are expected to have similiar number of water clusters.

DCD	Clusters			Constant			Hydrogen		
	Mean	*Min*	*Max*	*Mean*	*Min*	*Max*	*Mean*	*Min*	*Max*
smallwater - 8	23	19	27	98	81	112	135	121	152
smallwater - 12	22	18	27	99	88	106	135	127	145
water - 8	39	8	1 086	1 879	951	9 192	30 323	23 009	31 250
water - 12	10	1	568	446	189	4 658	31 756	27 543	32 012
hotwater - 8	2 007	6	3 486	14 704	838	20 086	17 614	12 115	31 363
hotwater - 12	1 305	1	1 434	8 394	184	8 712	23 815	23 489	32 017
rabies - 8	8 334	8 321	8 334	50 691	50 682	50 691	55 219	55 210	55 219
rabies - 12	5 016	5 011	5 016	26 437	26 429	26 437	79 472	79 464	79 472
hiv - 8	12	5	199	216	94	1 291	4 892	3 817	5 014
hiv - 12	9	3	190	139	85	1 160	4 969	3 948	5 023
mscl - 8	1 033	728	1 341	7 423	5 755	8 542	15 068	13 945	16 732
mscl - 12	593	534	668	4 092	3 744	4 327	18 396	18 160	18 743

Table 4.4: Statistics from our water compressor at 8 and 12 bit quantisation. *Clusters* is the number of components created in the graph creation stage. *Constant* is the number of molecules predicted with the constant predictor. *Hydrogen* is the number of molecules predicted with either hydrogen predictor. *Mean*, *Min* and *Max* are the respective values over the frames.

As expected, temperature affected the results. *Hotwater* and *water* are both the same simulation, except run at 1 000°K and 25°C respectively. The bonds between water molecules at the higher temperature are not as stable leading to *hotwater* to have significantly more water clusters (around 100 times more). This lead to the hydrogen predictor to be used a lot less and the constant predictor more. Surprisingly this did not affect the compression results significantly. At both quantisation levels *water* only performed 0.7% better than *hotwater*. Further inspection indicates the number of clusters is insignificant compared to the number of water molecules. This accounts for the small difference in ratios.

A good indicator on how well the compression performed is the percentage difference between the mean number of constant predictions and the mean number of hydrogen predictions. If the number is small, it means that the constant predictor is used approximately the same number of times as the hydrogen predictors. This indicates that the hydrogen predictors did not successfully predict well. Comparing these values with the values in the results shows that the closer constant and hydrogen predictor use is, the worse the water compressor will perform. *Rabies* and *smallwater* are the worse performers in the results, and they have the closest constant and hydrogen predictor means. While *water* and *hiv* did the best and have very different constant and hydrogen predictor means.

4.5 Permutation Results

Studying the differences between the null and best columns in Tables 4.2 and 4.3 indicate that a significant portion of the compression is lost due to the encoding of the permutation. In some cases encoding the permutation causes the scheme to perform worse than merely quantising the file.

Figure 4.3 shows the relative performance of each permutation encoder with our water compressor at 12-bit quantisation. The best permutation encoder is the *delta* encoder. The pathological cases which cause poor performance are

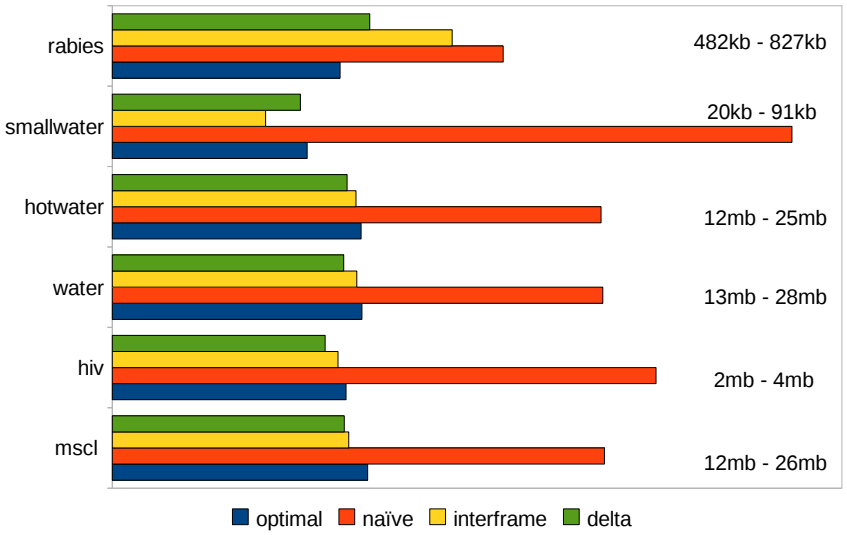

Figure 4.3: The relative performance of the permutation encoders on different DCD files. The encoder used is our water compressor at 12-bit quantisation. The numbers on the right indicate the smallest and biggest size of the encoded permutation.

unlikely, which is why the *delta* encoder performed well.

The most stable encoder is the *optimal* encoder. This is since the size of the encoding is related to the size of the permutation (see Equation 3.1).

4.6 Summary

From the results it can be seen that our water compression scheme performs best. This is also the case for when the original order of points is not needed. Generally the larger the amount of water in the file, the better the gain is on other schemes. In some cases a gain of up to 11% was made.

The number of times the hydrogen predictor is used versus the number of times the constant predictor is used is a good indicator of how well our water compression scheme performs. If the hydrogen predictor is used significantly more, then our water compression scheme makes good gains on other schemes.

The speed of our water compressor can be up to three times slower than the next slowest scheme. The slowdown is a constant factor, but the choice on which encoder to use is a compression ratio vs compression time trade-off.

The permutation information requires a significant portion of the compressed file. In some of the references scheme it caused the compression to be worse than simply quantising the file. The best permutation encoder was the *delta* encoder, while the most stable was the *optimal* encoder.

Chapter 5

Conclusion

Our water compression scheme compresses better than existing point cloud compressors (Section 4.6), since it effectively exploits the water-dimer model (Section 2.5 and 3.4.2). Our water compression scheme performed best at both 8-bit and 12-bit quantisation. At 8-bit quantisation the mean and average compression rate for water compression is 17.7%. The next best compression rate was gzip with a mean and average compression rate of 19.5% and 18.5% respectively. At 12-bit quantisation the mean and average compression rate for water compression is 29.8%. The next best scheme is gzip which has a mean and average compression rate of 37% and 36.5% respectively.

Our water compression technique performed best on simulations with large amounts of water. In some cases the gain made on the next best scheme was 11%.

The permutation information consumes a significant portion of the compressed file. A permutation encoder called *optimal* was shown to take approximately $\sum_{i=1}^{N} \log_2(i)$ where N is the size of the permutation (Section 3.3). The formula is a lower bound on the worst performance of a permutation encoder.

Probabilistically the worst case permutation will not occur when using the *delta* encoder. The results indicate that this encoder performs best in practice.

The complexity of encoding in our water compression scheme is $O(n + n \log n + e \log e + n) = O(e \log e)$ where $e = O(n^2)$ and n is the number of water molecules. Empirically and due to the theoretical models of water, e is typically less than $5 \times n$, and so $e = O(n)$. So the encoding time is $O(n \log n)$. The decoding time is $O(n)$.

5.1 Future Work

Parallelism Intra-frame compressors work independently of each frame in a simulation. This permits parallelisation over frames. One of the negative factors of our water compression scheme is the slower run time. The run time was slower by a constant factor. A parallel implementation would reduce this constant factor (and possibly make it less than one).

General Molecular Dynamics Predictors Running an approximate simulation and using the positions it obtains as predictors should give good general

predictors. The predictors will work on all atom types and should be more accurate than the water-dimer model. This approach also does not need to store the permutation information.

Better joining of components in spanning tree Our water compressor implementation joins up the components by adding an edge to a root. This is currently done in a naïve way, causing a wider range of less used residuals. A better scheme would cause the component connections residuals to be minimised. Possible ways include using a minimum spanning tree of the components or encoding the components separate to the errors. Adapting the minimum spanning tree approach by Chen et al. [2005] is a good candidate.

Point Cloud Compressor which preserve ordering A significant portion of the compressed data involved storing the permutation. A point cloud compressor which preserves the ordering is likely to perform better than our water compression scheme. The scheme will also have application outside of MD compression, since many point clouds have information associated with each point.

Bibliography

Vmd - visual molecular dynamics, theoretical and computational biophysics group, 2009a. http://www.ks.uiuc.edu/Research/vmd/.

Vmd - x-plor dcd format, 2009b. http://www.ks.uiuc.edu/Research/vmd/plugins/molfile/dcdplugin.html.

P. Alliez and C. Gotsman. *Recent Advances in Compression of 3D Meshes*. 2005. ISBN 3-540-21462-3.

M. Chaplin. Water molecule structure, 2009. http://www1.lsbu.ac.uk/water/molecule.html.

D. Chen, Y. Chiang, and N. Memon. Lossless compression of point-based 3D models. In *Pacific Graphics*, pages 124–126, 2005.

G. Cormack and R. Horspool. Algorithms for adaptive Huffman codes. *Information Processing Letters*, 18(3):159–165, 1984.

T. Cormen, C. Leiserson, R. Rivest, and C. Stein. Introduction to algorithms, 2001.

O. Devillers and P. Gandoin. Geometric compression for interactive transmission. In *Proceedings of the conference on Visualization'00*, pages 319–326. IEEE Computer Society Press Los Alamitos, CA, USA, 2000.

E. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

DivX. DivX.com, 2009. http://www.divx.com/.

A. Drozdek. *Elements of Data Compression*. Brooks/Cole Publishing Co, Pacific Grove, CA, USA, 2001. ISBN 053438448X.

P. Fenwick. A new data structure for cumulative frequency tables. *Softw. Pract. Exper*, 1994.

J. Gailly and M. Adler. The gzip home page, 2009. http://www.gzip.org/.

S. Gumhold, Z. Kami, M. Isenburg, and H.-P. Seidel. Predictive point-cloud compression. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, page 137, New York, NY, USA, 2005. ACM.

D. Huffman. A method for the construction of minimum-redundancy codes. *Kibern. Sb.*, 3:79–87, 1952.

B. Merry, P. Marais, and J. Gain. Compression of dense and regular point clouds. In *Afrigaph '06: Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 15–20, New York, NY, USA, 2006. ACM. ISBN 1-59593-288-7.

D. M. Mount and S. Arya. Ann: A library for approximate nearest neighbor searching, 2006. `http://www.cs.umd.edu/~mount/ANN/`.

A. Omeltchenko, T. Campbell, R. Kalia, X. Liu, A. Nakano, and P. Vashishta. Scalable I/O of large-scale molecular dynamics simulations: A data-compression algorithm. *Computer Physics Communications*, 131(1):78–85, 2000.

R. Sedgewick. *Algorithms in C*. Addison-Wesley Professional, 2001.

J. S. Vitter. Design and analysis of dynamic huffman codes. *J. ACM*, 34(4): 825–845, 1987. ISSN 0004-5411.

M. Wu. Water Compression Visualisation, 2009. Honours Thesis.