Efficient Compression of Molecular Dynamics Trajectory Files

Patrick Marais,*[a] Julian Kenwood,[a] Keegan Carruthers Smith,[a] Michelle M. Kuttel[a] and James Gain[a]

We investigate whether specific properties of molecular dynamics trajectory files can be exploited to achieve effective file compression. We explore two classes of lossy, quantized compression scheme: "interframe" predictors, which exploit temporal coherence between successive frames in a simulation, and more complex "intraframe" schemes, which compress each frame independently. Our interframe predictors are fast, memory-efficient and well suited to on-the-fly compression of massive simulation data sets, and significantly outperform the benchmark BZip2 application. Our schemes are configurable: atomic positional accuracy can be sacrificed to achieve greater compression. For high fidelity compression, our linear interframe predictor gives the best results at very little computational cost: at moderate levels of approximation (12-bit quantization, maximum error $\approx 10^{-2}$ Å), we can compress a 1–2 fs trajectory file to 5–8% of its original size. For 200 fs time steps—typically used in fine grained water diffusion experiments—we can compress files to ~25% of their input size, still substantially better than BZip2. While compression performance degrades with high levels of quantization, the simulation error is typically much greater than the associated approximation error in such cases.

DOI: 10.1002/jcc.23050

Introduction

The rapid increase in available computational power at high-performance computing centers over the last decades has revolutionized computational science. However, the concomitant massive increase in the data produced by scientific simulation—the so called "data deluge"—has created another set of problems. In the case of molecular dynamics (MD), simulation trajectory files can now record the motion of millions of atoms over nanoseconds to microseconds, producing terabytes of data. Efficient data compression methods are thus needed, not only for storage of these huge data sets, but also to facilitate remote access to large data files for local analysis.

An effective MD compression scheme should exhibit the following characteristics:

- *High compression*. The ratio of the compressed trajectory file size to the original uncompressed file size should be as small as possible, as achieving significantly smaller file sizes is the primary purpose of compression.
- *Low run times*. The computational cost of both compression and decompression should be minimal. For instance, the particular case of compression for once-off transmission makes no sense if compression times dominate over the original uncompressed transmission time. However, for some applications, it is worth spending more time on compression (often executed once on a more powerful machine) than on decompression (often executed multiple times on less powerful machines), if this will yield a better compression or faster decompression times.
- *Streaming*. Ideally, an MD compression scheme should allow for compression of MD frames, as they are produced. This has multiple benefits: the simulation and compression can be executed in parallel, the original uncompressed data set can be held on secondary storage/disk, allowing for much larger file sizes, and different subsets of the data can be compressed independently in parallel.
- *Configurability*. It should be possible to sacrifice accuracy for higher compression, when required. For instance, visualization of MD trajectories typically requires less accuracy than analysis.
- *Simplicity*. Implementations of complex algorithms tend to be less robust and harder to extend and maintain.

Although data compression is a well-studied problem, with a number of schemes aimed at the compression of domain-specific data (such as 3D models, images, and videos[1,2]), the compression of MD trajectory files has not yet been explored to any great extent. In most cases, the only available methods are generic compression applications such as bzip2[3]—a popular and highly efficient public domain compressor used in many scientific computing applications. MD compression methods developed thus far consider neither the computational cost of compression nor the specific structure of the MD data files. Often, the entire data set needs to be collected before compression can begin, and the order of atoms in the trajectory file may be sacrificed. The preservation of point ordering across frames is essential for many postsimulation analyses of MD trajectories. In addition, little attention is paid to the characteristics of the input data or run time, though

P. Marais, J. Kenwood, K. Carruthers Smith, M. M. Kuttel J. Gain
Department of Computer Science, University of Cape Town, Cape Town, South Africa
E-mail: patrick@cs.uct.ac.za

Contract/grant sponsor: National Research Foundation of South Africa.

these factors can have a significant effect on the final utility of the algorithms. Structural properties that might be exploited are solvent water (which comprises a large proportion of many simulated biological systems) and temporal coherence (the positions of atoms tend not to vary significantly across successive frames). Such domain-specific information can potentially be leveraged to improve compression ratios.

Thus far, methods using principal component analysis (PCA) have been the primary focus of data compression schemes for MD.[4,5] PCA enables dimensionality reduction by discarding redundant information—this allows atom locations to be represented with fewer coordinates, at the cost of accuracy. The Essential Dynamics MD compression algorithm proposed by Meyer et al.[4] compresses MD trajectory files to 5–20% of the input file size, with a corresponding root-mean squared (RMS) error of 0.5–0.1 Å. However, this method does not support streaming, because all MD frames must be generated prior to compression. File sizes are hence limited by the available physical memory. In addition, the authors do not provide run times for their algorithm, so it is hard to assess the practicality of their compression scheme for very large data sets. Run times are, however, unlikely to be good, as PCA methods are expensive: an Eigen problem is solved, requiring the inverse of a large matrix system. It should also be noted that aggregate measures like RMS tend to hide large but transitory spikes in the error, which are very likely with the extreme dimensionality reduction imposed by PCA.

Kumar and Tu[5] also use PCA for compression of MD trajectories but focus on compressing the trajectory of each atom separately. In their method, each trajectory is considered over a contiguous subset of frames, and a PCA is computed for this sequence. The reduced representation is then further compressed by means of a Discrete Cosine Transform.[1] Their quoted RMS errors are in the range of 0.2–0.45 Å. However, the authors only discuss the compression step. For decompression, a set of basis 3-vectors and a mean 3-vector for each trajectory is required, of which there may be millions. Furthermore, each frame block will require a new set of vectors. The cost of storing this side information will severely impact compression performance as the block size shrinks.

The existing extensive literature on point cloud compression in the area of Computer Graphics[2,6,7] has potential, as yet unexploited, application to MD trajectories: an MD frame can be viewed as a 3D "point cloud" of atomic positions, with additional metadata (such as velocity, atom ID, etc.) attached to each point. The simplest approach to compressing volumetric point clouds is to use an octree—a data structure that decomposes a 3D volume into a series of nested cuboids. The MD frame compressor proposed by Omeltchenko et al.[8] follows this approach, using an octree to store the atom positions, and then traversing the octree using a "Z" (space-filling) curve to minimize prediction error between consecutive point samples. Frames are compressed independently, so no attempt is made to exploit temporal coherence in this method. Results indicate an average compression to 12% of the original file size at a quantization interval of $\approx 10^{-3}$ Å with this method. Unfortunately, this scheme sacrifices the numerical order of atoms in the MD file: the uncompressed file will not list the atoms in the same order in which they were compressed.

The point cloud compressor proposed by Devillers and Gandoin[6] uses a kd-tree to "progressively" encode a general 3D point cloud. The kd-tree is a generalization of the octree, which allows for nonuniform splits of each parent node. The algorithm traverses the kd-tree and outputs a sequence of numbers representing the point count in each tree cell. Only the count of one cell of a split parent cell need be encoded: the other can be immediately inferred from the parent point count. An arithmetic encoder[1] is used to optimally compress the number of bits needed for each number in the sequence. Unfortunately, in this scheme the entire data stream needs to be decompressed to get back the original point set. While the compression results are good, this again comes at the cost of point ordering, because the scheme will not produce points in the same order in which they were compressed.

Point cloud schemes usually use quantization to enable high compression ratios: each point coordinate is allocated to a specific bin in a three-dimensional grid. The number of bins is determined by the available "bit budget" per coordinate. For example, 12 bits per coordinate allows for $2^{12} = 4096$ bins, which uniformly subdivides the min/max range of that coordinate. By comparison, a standard IEEE single precision "float" requires 32 bits of storage. Quantization makes the compression process inherently "lossy," but lossless compression schemes result in lower compression ratios and are only used if every bit must be precisely reconstituted. In the case of MD data, the input is already quantized to the precision permitted by the floating point representation and the precision at which data needs to be represented is context-dependent. We take the view that additional quantization is often acceptable, as seems to be the implicit assumption of all the compression papers we have surveyed. In contrast to the PCA methods discussed earlier, quantization of the space gives an absolute maximum error bound that will be never exceeded. For example, the MD compression scheme of Omeltchenko has a maximum error bound of 10^{-3} Å, whereas the Essential Dynamics paper does not report on errors below 0.1 Å, and these RMS errors only permit compression ratios of around 14–22% for the protein structures they explored.

After quantization, additional data compression of point clouds can be achieved using the current frame data to predict the location of points that have not yet been processed. A good prediction scheme will result in small prediction errors, which can be losslessly compressed by means of "entropy coding."[1] The schemes most commonly used to compress points clouds are tailored toward the compression of an implied 3D mesh surface.[2,7] In these cases, sensible prediction schemes can be devised that roughly approximate the implied surface and lead to small prediction errors and correspondingly good compression. However, the molecules in an MD simulation are typically space-filling and thus not suited to such surface predictors.

In this work, we investigate whether properties of common MD trajectory files—the large proportion of solvent water in many simulated systems and temporal coherence between

frames—can be exploited to achieve better compression. Specifically, we develop and compare two classes of lossy, quantized compression scheme: "interframe" predictors that exploit temporal coherence between successive frames in a simulation (using low-order polynomial predictors followed by adaptive arithmetic entropy coding), and more complex "intraframe" schemes that compress each frame independently and exploit the fairly rigid geometry of water molecules in MD simulations. (The water compressor compresses each frame independently, because water generally exhibits poorly correlated structure across time.) In general, our proposed schemes exhibit high compression ratios, low run times, streaming, configurability, and simplicity.

We benchmark our algorithms against a number of published methods: our optimized version of Omeltchenko et al.,[8] Devillers and Gandoin,[6] as well as BZip2.[3]

We evaluate our methods on a number of trajectory files incorporating a variety of structures at different quantization levels and data sampling rates, to ensure broad coverage of likely trajectory files and identify the most appropriate compression schemes. We are particularly interested in the compression of simulations with small time steps, because the limitations of existing compressors have largely meant that, currently, such simulations are substantially subsampled prior to compression. Our results show higher compression ratios and faster run times than both computationally expensive point cloud compressors and current schemes designed specifically for MD compression.

Methods

We focus on compression of atom positional data, in keeping with prior work. Our compression algorithms begin with quantization of the atomic coordinates as a first step and then proceed to lossless compression of this quantized data. We use a simple linear quantizer[1] along each coordinate axis, as follows. Given a real value $x \in [a, b]$, and a quantization budget of q bits, x is mapped to an integer bin index, $n \in [0, 2^q - 1]$, according to

$$n = \left\lfloor \frac{x - a}{b - a}(2^q - 1) + 0.5 \right\rfloor \quad (1)$$

where $\lfloor \cdot \rfloor$ is the floor operator. To recover a real number x from a bin number n, we evaluate

$$x = a + n\left[\frac{b - a}{2^q - 1}\right] \quad (2)$$

The maximum quantization error for q bit quantization is a function of the simulation bounding box size. More specifically,

$$\text{max\_error} = \frac{1}{2^{q-1}}\text{box\_diagonal} \quad (3)$$

where *box_diagonal* is the diagonal length of the simulation bounding box. When providing compression results we always indicate the maximum quantization error and bounding box size.

Our compression algorithms produce quantized "bin numbers" for each positional coordinate as output. These bin numbers are not uniformly distributed over the entire range of possible quantized numbers. Instead, they tend to cluster in a way that can be usefully exploited by an entropy coder. Arithmetic coding[1] allows one to closely approach the theoretical entropy bound of the bin number sequence. A single arithmetic coder is used to compress the bin numbers for *x*, *y*, and *z* quantized values, in turn. As each bin number is encoded, new bits are added to a growing bit string that corresponds to the encoded data stream.

Arithmetic encoder

We have chosen to implement a modified form of "adaptive arithmetic coding."[9] Our implementation is based largely on publicly available code.[10] Those parts of the code which handle the symbol mapping and frequency calculations needed in arithmetic coding have been replaced with our own data structures. The principal justification for our modifications is to reduce the size of the symbol table, because many bin values are unlikely to be used. This, in turn, reduces the size overhead associated with arithmetic coding.

A symbol's frequency is stored in a "Fenwick tree"[11] for efficient updates and fast access. In the following pseudocode, the table array represents the data stored in the Fenwick tree.

We need to store two mappings: a mapping from symbol numbers to data for decoding and a mapping from data to symbol numbers for encoding. The mapping from symbol number to data uses a simple array. We provide two procedures to access data by symbol and add new data—Algorithms 1 and 2.

Algorithm 1 findDataFromSymbol(dataTable, symbol)
return dataTable[symbol]

Algorithm 2 addSymbolToData(dataTable, data)
symbol ⇐ len(dataTable)
data[symbol] ⇐ data
return symbol

We use a data structure called a "trie"[12] to store the mapping from data to symbol numbers. The trie is structured like a tree and allows efficient mapping from byte strings to other data.

We describe two algorithms to retrieve a symbol from data mapping from the trie and add a new mapping to the trie.

Algorithm 3 findSymbolFromData(trie, data)
trieNode ⇐ trie
for all byte in data **do**
  if hasChild(trieNode, byte) == False **then**
    return − 1
  end if
  trieNode ⇐ child(trieNode, byte)
end for
return value(trieNode)

Algorithm 3 returns the symbol identifier if it is found otherwise it returns −1. The child(trieNode, byteValue) returns the byteValue'th child of trieNode, whereas hasChild(trieNode, byteValue) tests if trieNode has the requested child. value(trieNode) returns the value of trieNode's value variable.

Algorithm 4 mapDataToSymbol(trie, data, symbol)
trieNode ⇐ trie
for all byte in data **do**
  if hasChild(trieNode, byte) == False **then**
    child(trieNode, byte) ⇐ Trie()
  end if
  trieNode ⇐ child(trieNode, byte)
end for
value(trieNode) ⇐ symbol

Algorithm 4 associates data with a symbol id. Trie() is a function that creates a new Trie node. The initial trie node contains no children and the value stored at the node is initialized to −1. Algorithm 5 interacts with the arithmetic coder to output data. The arithmetic coder uses an "adaptive model."[9] so additional work is done when encountering new symbols.

Algorithm 5 output(data)
symbolID ⇐ findSymbolFromData(dataTrie, data)
if symbolID == −1 **then**
  mapDataToSymbol(dataTrie, data, numSymbols)
  encode(_NEW)
  updateFrequency(_NEW)
  writeRaw(data)
  numSymbols ⇐ numSymbols + 1
end if
if symbolID ≠ −1 **then**
  encode(symbolID)
  updateFrequency(symbolID)
end if

The encode(symbol) function requests the arithmetic coder to output the designated symbol. updateFrequency(symbol) corresponds to updating the frequency of the symbol in the Fenwick tree. writeRaw(data) is a method that we added to our arithmetic coder. It allows us to safely output data to the stream without interfering with the arithmetic coder. There are no other special implementation details attached to this function.

Finally, Algorithm 6 reads a symbol from the arithmetic coder:

Algorithm 6 read()
frequency ⇐ decode()
readSymbol ⇐ findByFrequency(frequency)
updateFrequency(readSymbol)
if readSymbol == _NEW **then**
  newSymbolData ⇐ readRaw()
  addSymbolToData(dataTable, newSymbolData)
  numSymbols ⇐ numSymbols + 1
  return newSymbolData
end if
if readSymbol ≠ _NEW **then**
  return findDataFromSymbol(dataTable, readSymbol)
end if

The decode function reads from the arithmetic coder. The read data must be mapped from a cumulative frequency to a symbol number. We exploit the properties of the Fenwick tree (efficient prefix sum calculation) with the findByFrequency(frequency) function returning the appropriate symbol id. readRaw() is analogous to the writeRaw() but used to read raw data.

Interframe prediction

For interframe prediction, we require only a small number of preceding frames to predict atom positions in the new, unprocessed, frame. This reduces the memory requirements of our algorithms, and opens the way for compression of streaming data. Once a prediction has been generated, the difference between each atom's true position and the prediction (a "delta") can be quantized and compressed using adaptive arithmetic encoding.

The frames comprising an MD simulation are usually temporally coherent. This suggests the use of low-order polynomial predictors to approximate the atom positions in the next frame, relative to the frames already processed. It is assumed that the frames are spaced equally in time.

The following predictors were implemented:

Polynomial. The polynomial predictor interpolates the current position and K previous atom positions, $\{(x, y, z)(t_i), i = 0,\ldots,K\}$, and extrapolates the position in the next frame. When $K = 0$ we have a constant predictor, which gives rise to "delta encoding."

Given a set of data points, $\{(t_0, y_0),\ldots,(t_K, y_K)\}$, we can define the Lagrange polynomial[13] as

$$F_y(t) = \sum_{j=0}^{K} y_j \ell_j(t) \tag{4}$$

where

$$\ell_j(t) = \prod_{i=0, i\neq j} \frac{t - t_i}{t_j - t_i} \tag{5}$$

Each atom coordinate will have a similar interpolant defined. We can optimize this computation by noting that the ℓ_i weights can be precomputed when we only evaluate $F()$ at a finite set of integral values. $F()$ will only be evaluated over the discrete set $[0,K]$. Typically, K is small (one or two for our work), so the storage overhead is minimal.

Spline. Higher order polynomial interpolants tend to oscillate. To counter this, we also implemented a spline predictor that only interpolates the first and last samples over the previous frames and smooths the intervening positional data. The spline curve can then be used to predict the atom position in the next frame. The (Bezier) spline interpolant[13] can be computed using:

$$F_y(t) = \sum_{i=0}^{K} \binom{K}{i} (1-t)^{K-i} t^i y_i \tag{6}$$

Similar interpolants can be defined for each coordinate function. Note that we have to scale the input parameter domain to ensure the interpolant values lie in the correct range: $t \leftarrow t/K$.

We can precompute the binomial coefficients, because we only consider a small set of integral values of K.

Note that no additional calculation is required for each atom, beyond the simple equations presented earlier.

Algorithm 7 Interframe Compression
file_header = {comp_params, f_params}
initialize AC state
write(stream, file_header)
while frames available **do**
  f = get_next_frame()
  frame_header = {bounding_box}
  write(stream, frame_header)
  for i = 1 to number_of_atoms **do**
    A = quantize(Atom[i], bounding_box)
    if no_prediction_frames **then**
      $\delta = A$
    else
      A^* = predict(pw[i], comp_params)
      δ = quantize(A^*) – A
    end if
    pw[i] = update_pred_window(pw[i], A)
    arithmetic_encode(stream, δ_x)
    arithmetic_encode(stream, δ_y)
    arithmetic_encode(stream, δ_z)
  end for
end while

Algorithm 8 Interframe Decompression
initialize AC state
{comp_params, f_params} = read(stream)
while stream has data **do**
  {bounding_box} = read(stream)
  $i = 1$
  while frame has data **do**
    δ_x = arithmetic_decode(stream)
    δ_y = arithmetic_decode(stream)
    δ_z = arithmetic_decode(stream)
    if no_prediction_frames **then**
      $A = \delta$
    else
      A^* = predict(pw[i], comp_params)
      A = quantize(A^*) $- \delta$
    end if
    pw[i] = update_pred_window(pw[i], A)
    Atom[i] = dequantize(A, bounding_box)
    $i = i + 1$
  end while
end while

The compression and decompression phases are summarized by the pseudocode presented in Algorithms 7 and 8. Here, pw[i] represents the prediction window for atom i and update_pred_window() adds and removes quantized atoms positions to the appropriate prediction window. Note that prediction is based on "quantized" atom coordinate values, because only these are available to both the compression and the decompression pass. The various mappings we referred to earlier have not been expanded in this simplified outline.

Intraframe water predictor

As water is an important component in many simulations, one cannot simply discard water molecules as a means of compression. However, water does display some structure, albeit very locally.[14] As this structure is local, our water compressor does not attempt to predict the movements of water molecules across frames: the sequence is compressed on a frame-by-frame basis.

Each frame is processed to separate out water molecules from nonwater constituents. The DCD/PDB input files enumerate all atoms and the bonds they form with neighbors. Using this information, we can generate a list of water molecules and a list of nonwater atoms. The water predictors do not attempt to compress nonwater constituents in any special way: these atoms are simply quantized and subjected to interframe delta encoding.

We use a water dimer model,[14] because this is easy to implement and imposes a simple *a priori* constraint on adjacent loosely coupled water molecules. This setup is illustrated in Figure 1. The oxygen, which is loosely bonded to the hydrogen, is expected to be $\approx$2.97 Å away from its neighboring oxygen. In addition to this distance constraint, the vector created by the "bound" O—H bond should be approximately lined up with the vector attached to the "loose" O—H bond.

The water encoding scheme operates by building a spanning tree of "neighboring" water dimers and then traversing the tree in a breadth-first fashion, compressing data as it proceeds. We use an approach similar to that of Merry et al.[7]: as we traverse the tree, we write out the index of the best predicting encoder, along with the prediction deltas.

The edges in a tree induce a parent–child relationship. As we proceed down each subtree, we use the molecular information in a parent node to predict the orientation and position of its children molecules.

Three water predictors are employed during tree traversal. In the following, O′, H1′, and H2′ represent parent water atom positions, and O, H1, and H2 the child atom positions we are attempting to predict.

Constant. O is predicted as O'. This deals with some cases were bonding information changes as the simulation progresses (information is fixed for the entire sequence);

H predictor1/2. The hydrogen predictor predicts O to be along $O' - H1'$ or $O' - H2'$, approximately. More specifically we assume $\alpha = 0$ and use $O' + 2.97 * \frac{O'-H'i}{\|O'-H'i\|}$ for $i = 1, 2$.

In all cases, we predict the new hydrogen locations as O, because there is too much variability in the β parameter to do otherwise.

A more detailed explanation of each stage is given as follows.

Algorithm 9 Graph Creation
create_graph(list_of_water_molecules):
  graph = graph of water molecules
  kdtree = KDTree(list_of_water_molecules.O_pos)
  for all cur_mol in list_of_water_molecules **do**
    radius = 3 Å
    for mol **in** kdtree.fixed_radius_search(cur_mol.O_pos, radius) **do**
      graph.addEdge(cur_mol, mol)
    end for
end for
return graph

Graph creation. we use a KD-tree for fast spatial lookups when associating O atoms in "neighboring" water molecules.

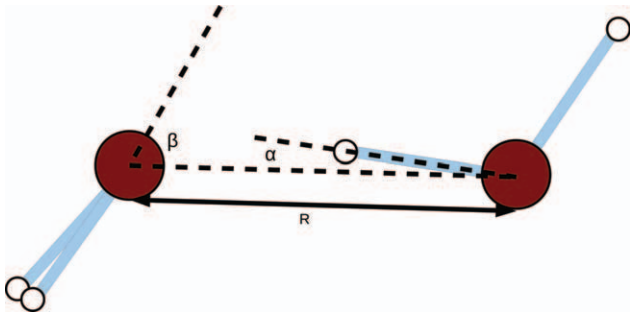

Figure 1. Simple water dimer model.

We can then add all water O atoms within a 3 Å radius as new graph edges and proceed until all water molecules have been processed. The pseudocode is shown in Algorithm 9.

Algorithm 10 Spanning tree creation

```
create_spanning_component(graph, tree, component_root, root):
    min_priority_queue q
    q.push((0, component_root → root, constant))
    while q not empty: do
      (residual, v′ → v, p) = q.pop()
      if v not in tree: then
        tree add edge v′ → v with predictor p
        predictors = {constant_predictor(p), h1_predictor(p), h2_predictor(p)}
        for all u in graph[v].children: do
          if u not in tree: then
            residual, predictor = smallest_residual(u, predictors)
            q.add((residual, v → u, predictor))
          end if
        end for
      end if
    end while
create_spanning_tree(graph):
  root = 0 // First molecule in file
  spanning_tree = tree()
  for all v in graph do
    if v not in spanning_tree then
      create_spanning_component(graph, spanning_tree, v, root)
    end if
  end for
```

Spanning tree creation. The above process connects water O atoms to form a graph, but there may be isolated components and there is no directionality to edges. We thus build a "minimum spanning tree," including the optimal predictor choice (which will minimize delta/residuals) as we proceed. We use a variant of well known Dijkstra's shortest path algorithm[15]—Algorithm 10. This will ensure that all components are linked into the final tree and that each edge connects to its closest predicted O atom neighbor.

Algorithm 11 Compression

```
compress(spanning_tree, water_molecules, root):
  bfs_queue = queue()
  bfs_queue.push(root → root)
  while bfs_queue not empty do
    v′ → v = bfs_queue.pop()
    predictor = spanning_tree.predictor_for(v′ → v)
    prediction = prediction(v′, predictor)
    water_molecule = water_molecules[v]
    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 u in spanning_tree.children(v): do
      predictor = spanning_tree.predictor_for(v → u)
      tree_encoder.encode(predictor)
      bfs_queue.push(v → u)
    end for
  end while
tree_encoder.encode(sentinel)
```

Compression. The spanning tree and per-edge predictor choices are then compressed. The edges are encoded in breadth first order, starting at the root of the tree and running the procedure outlined in Algorithm 11 until all edges in the breadth first queue have been processed. The adaptive arithmetic coder introduced earlier is used to compress both the prediction residuals (deltas) and the type of each predictor. A permutation encoder is used to compress the permutation vector required to correctly recover the water molecule ordering—see Algorithm 12.

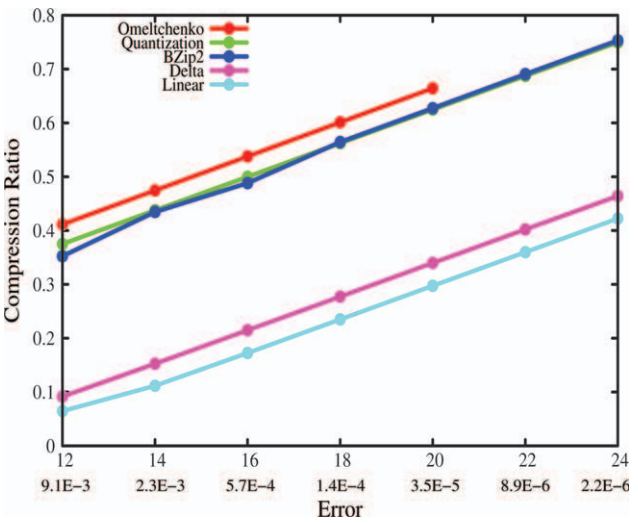

Figure 2. Changes in compression ratio as a function of permitted approximation error in Angstroms for the protein-normal dataset at a 2-fs sampling rate. Omeltchenko, Quantization, and BZip2 compression exhibit roughly comparable performance, whereas Delta and Linear compression perform significantly better.

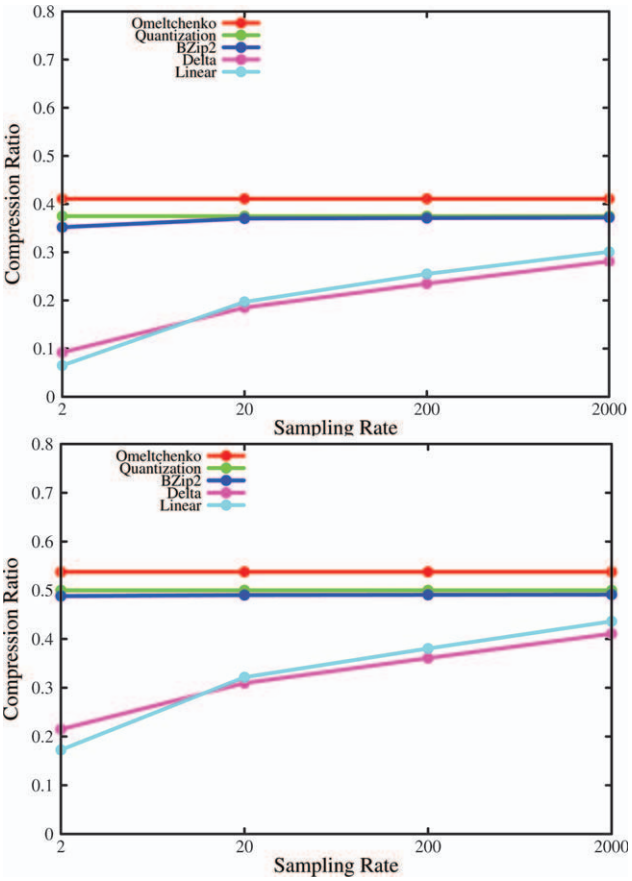

Figure 3. Changes in the compression ratio as a function of sampling, at 2, 20, 200, and 2000 fs sampling intervals. Omeltchenko, Quantization, and BZip2 compression do not rely on interframe coherence, whereas Delta and Linear compression do and degrade as the sampling interval increases. The dataset is protein. Top: approximation error of 0.0091 Å (12-bit quantization); Bottom: approximation error of 0.00057 Å (16-bit quantization). [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

Algorithm 12 Decompression

```
decompress(residual_decoder, tree_decoder, permutation_decoder):
  q = queue()
  q.push((NULL, constant_predictor))
  while q not empty: do
    (parent, predictor) = q.pop()
    prediction = predict(parent, predictor)
    O_position = residual_decoder.decode() + prediction
    H1_position = residual_decoder.decode() + O_position
    H2_position = residual_decoder.decode() + O_position
    child_preds = list()
    while true do
      pred = tree_decoder.decode_int()
      if pred == sentinal then
        break
      else
        child_preds.push(pred)
      end if
    end while
    index = permutation_decoder.decode()
    for pred in child_preds do
      q.push((index, pred))
    end for
    water_molecules[index].oxygen = O_position
    water_molecules[index].hydrogen1 = H1_position
    water_molecules[index].hydrogen2 = H2_position
  end while
```

Decompression. Decompression reverses the compression algorithm and regenerates a quantized frame—Algorithm 12. This process recovers the tree structure, atom positions, and molecule ordering.

Permutation encoding Unfortunately, the water compressor does not guarantee that the trajectory file atom ordering will be preserved across frames. This is a consequence of the tree construction process that uses spatial proximity to associate molecules, and thus determine tree traversal. These associations can change from frame to frame.

To deal with this issue, a permutation can be generated which describes how to permute the original atom indices to get the correct associations in the new frame. This is expensive, however: each atom needs an index in this permutation vector, and each frame (after the first) requires such a vector. To compress this permutation vector, we apply the same adaptive arithmetic coder used elsewhere in the compression pipeline. If the order does not need to be preserved, this step can be removed from the compression pipeline.

Evaluation

The evaluation of our compressors seeks to elucidate the impact of several factors that may affect the efficiency of the codecs. We examine the impact of the following factors:

Data quantization (*Fig*. 2). Quantizing the atom positional data is vital to achieving useful compression ratios. We present compression results for several quantization levels, all of which yield an error much lower than 10^{-3} Å. Most reported

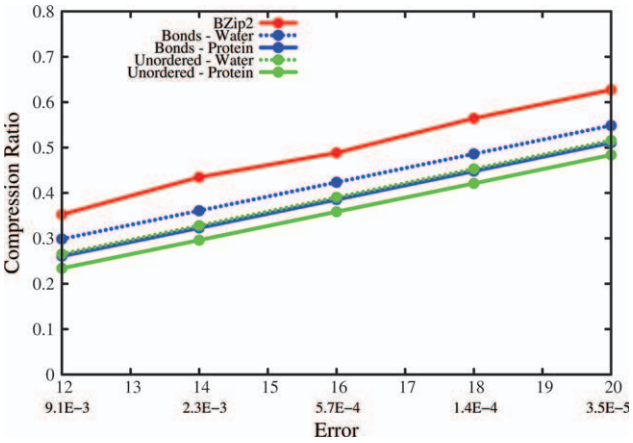

Figure 4. Changes in the compression ratio as a function of solvent ratio for protein-normal (solvent ratio = 0.275) and pure water (solvent ratio = 1.0) datasets at a 2-fs sampling rate and 0.00057 Å error. BZip2 compression is not affected by the solvent ratio, while, counter-intuitively, Bonds compression, which exploits the structure of water, degrades as solvent ratio increases. The 'unordered' graphs show that if one discards atom ordering information, compression improves marginally. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

Table 1. Information about the datasets used for experiments.

Dataset	Molecule	No. atoms	Size (GB)	Dim (Å)	Time step (fs)	# Frames	% Solvent	Temp. (K)
Protein-normal	Ubiquitin	7053	19.73	40.7 × 42.4 ×46.3	2	250,005	82.5	310
Protein-warm	Ubiquitin	7053	19.73	40.7 × 42.4 ×46.3	2	250,005	82.5	500
Pure water	Water	5943	16.63	40 × 40 × 40	2	250,100	100	300
Carb-normal	Maltose	10,353	2.91	3 × 55 × 55	1	25,100	94.3	300
Carb-isolated	Maltose	45	1.44	3 × 55 × 55	1	2,500,100	0	300
DNA	DNA	22,339	6.44	68.5 × 43.9 × 80.9	1	25,800	99.6	300

Columns are (from left): mnemonic used in figures, type of the primary molecule, number of atoms, size of the uncompressed simulation (in GB), simulation box dimensions (in Å), simulation time step (in fs), number of frames in the simulation, percentage of solvent present, and temperature (in K).

schemes choose a single quantization level, which may be inappropriate and do not investigate this further.

Data sampling (*Fig*. 3). High fidelity simulations are often subsampled to gain some insight into broad scale characteristics. This subsampling step affects frame coherence and requires careful scrutiny. We run tests at multiple subsampling intervals to assess the impact of this operation on the compression ratio.

Water (*Fig*. 4). we wish to determine if explicitly modeling the local intraframe properties of water can lead to a gain in compression performance.

Simulation temperature. "hot" systems typically have different frame to frame characteristics, and it is harder to predict the future positions of points without running an expensive simulation, which is not practical.

We implemented two generic point compressors: the kd-tree point compressor[6] and the spanning-tree scheme of Merry et al.[7] We included results for both BZip2 and straight quantization, that is, compression by discarding precision bits for each coordinate. We have also included our implementation of the MD compression scheme proposed by Omeltchenko,[8] as this is one of the few published MD-specific compression schemes. This implementation was based on the details provided in the article, and optimized where possible.

Simulation Details

To test our compression algorithms, six trajectory files were generated, with simulation systems as listed in Table 1. In all cases, the MD simulations were performed with the NAMD package.[16] Solvate molecules were represented with the CHARMM protein[17] and carbohydrate[18] parameter sets. Initial velocities for the atoms were selected at random from a Boltzmann distribution at 300 K. All simulations were performed in the canonical ensemble (constant *nVT*). The equations of motion were integrated using a Leap-Frog Verlet integrator with a step size of 1 fs. The SHAKE algorithm was used to fix the length of bonds involving hydrogen atoms and the water molecule geometry throughout each simulation. Nonbonded interactions were truncated using a switching function applied on a neutral group basis between 10.0 and 12.0.

Results and Discussion

We implemented two generic point compressors: the KD-tree point compressor[6] and the spanning-tree scheme of Merry

Table 2. Computation time and compression ratios tabulated for six representative datasets (by row) against four compression schemes (by column) at a quantization level of 12 and 16 bits (each table).

		Omeltchenko			BZip2			Delta			Linear		
Dataset	Err (Å)	cratio	Ctime	dtime	cratio	ctime	dtime	cratio	ctime	dtime	cratio	ctime	dtime
12 bit quantization													
Protein-normal	0.0091	0.411	38.6	52.0	0.352	78.1	62.2	0.092	28.7	39.0	0.065	26.9	34.5
Protein-warm	0.0091	0.410	40.7	58.4	0.358	71.4	55.1	0.099	29.3	38.2	0.066	27.8	35.6
Pure water	0.0084	0.413	39.1	55.1	0.359	71.7	49.1	0.094	29.1	39.6	0.065	26.7	35.2
Carb-normal	0.0095	0.407	32.0	53.1	0.303	55.9	42.2	0.056	23.9	38.9	0.050	23.8	38.2
Carb-isolated	0.0095	0.486	64.8	118.8	0.068	70.2	64.7	0.079	73.8	64.5	0.084	74.8	64.7
DNA	0.014	0.408	32.3	54.2	0.288	56.8	44.3	0.050	25.9	40.9	0.048	23.4	39.0
16 bit quantization													
Protein-normal	0.00057	0.538	47.4	65.7	0.488	94.9	70.3	0.215	43.5	62.3	0.173	41.3	61.4
Protein-warm	0.00057	0.537	48.4	66.2	0.489	96.7	72.7	0.223	45.1	61.5	0.174	39.2	57.2
Pure water	0.00052	0.539	49.0	67.5	0.492	93.6	65.7	0.218	43.6	60.1	0.174	38.9	55.2
Carb-normal	0.00059	0.533	40.6	60.5	0.483	79.6	56.7	0.173	36.1	52.8	0.118	31.6	47.3
Carb-isolated	0.00059	0.616	52.9	102.3	0.248	69.4	73.2	0.148	76.5	69.7	0.125	75.5	66.9
DNA	0.00087	0.534	40.1	65.2	0.478	78.7	62.6	0.166	38.2	56.8	0.109	30.3	49.1

The sampling rate is 2 fs. Maximum error (in Å) at each quantization level is provided in the second column. For each compression scheme, the columns are (from left): the compression ratio, time to compress per GB (in seconds), and time to decompress per GB (in seconds). Linear compression performs the best across all datasets in terms of both compression ratio and computation time. Experiments were executed on a single CPU of an Intel i7 3.4 GHz quad-core with 16 GB of RAM.

Table 3. Compression ratios tabulated for six representative datasets at 12 and 16 bit quantization.

		cratio			
Dataset	Err (Å)	Omeltchenko	BZip2	Delta	Linear
12 bit quantization					
Protein-normal	0.0091	0.411	0.371	0.235	0.255
Protein-warm	0.0091	0.410	0.371	0.252	0.269
Pure water	0.0084	0.413	0.373	0.241	0.259
Carb-normal	0.0095	0.407	0.369	0.214	0.232
Carb-isolated	0.0095	0.486	0.188	0.161	0.169
DNA	0.014	0.408	0.368	0.204	0.223
16 bit quantization					
Protein-normal	0.00057	0.538	0.491	0.361	0.381
Protein-warm	0.00057	0.537	0.491	0.378	0.395
Pure water	0.00052	0.539	0.494	0.366	0.385
Carb-normal	0.00059	0.533	0.489	0.342	0.362
Carb-isolated	0.00059	0.616	0.307	0.270	0.277
DNA	0.00087	0.534	0.489	0.334	0.357

The sampling rate is 200 fs. The maximum error (in Å) at each quantization level is provided in the second column. Delta compression edges out Linear compression at this sampling rate. Our compressors continue to perform best across all datasets. Experiments were executed on a single CPU of an Intel i7 3.4 GHz quad-core with 16 GB of RAM.

et al.[7] As neither of these schemes performs well compared to BZip2, we exclude these results. In Table 2, we list results for both BZip2 as well as our implementation of the MD compression scheme proposed by Omeltchenko,[8] as this is one of the few published MD-specific compression schemes. This implementation was based on the details provided in the article and optimized where possible.

Quantization level has the most significant impact on compressibility. For example, quantizing a file of 32-bit IEEE single precision numbers to 12-bit values compresses the file to 37.5% (12/32) of its input size. Applying further entropy coding will usually lead to further gains. Furthermore, each additional quantization bit utilized reduces the approximation error by a factor of 2. Note that quantization error is directly related to the size of the simulation box, as pointed out in section "Methods." Our simulation box sizes are stated in Table 1, and we list the associated quantization errors for all our results.

From Figure 2 and Table 2, it is clear that compression ratios decrease linearly with a decrease in quantization level. Furthermore, the relative compression performance of the various schemes does not change. This means that we can recommend linear or delta prediction as the best schemes across all quantization levels. Note that these results are for a sampling rate of 2 fs; we clarify the affect of sampling rate on compression performance later in this section. Linear compression shows a noticeable gain over delta compression when the simulation step size is small. A good trade-off between approximation error and compression ratio exists at the 16-bit quantization level. In this case, the maximum approximation error is 5.7×10^{-4} Å for our test cases, and our predictors achieve compression ratios of between 11 and 21%, as shown in Table 2.

At the 12-bit quantization level (9.1×10^{-3} Å maximum error), the compression ratio improves to around 5–8.5% for our examples, using linear prediction—Table 2. Observe that the other schemes are consistently worse, by between 25 and 30%. Although the PCA-based method[4] achieves around 5% compression at best, this is only possible with an enforced RMS error of 0.1–0.5 Å. Extrapolating from Figure 2, our linear predictor would achieve around 3% at 10-bit quantization (maximum error 0.037 Å for our examples).

Our 12-bits quantization has an associated maximum representational error of only 9.1×10^{-3} Å, which is well below the error inherent in many experimental methods. It is also worth noting that, if quantization levels of 10 bits are acceptable, then the compression ratios of our schemes will improve markedly across the board. By contrast, PCA-based methods cannot represent highly accurate nonlinear behavior, and their performance is likely to degrade significantly, if high precision is required.

Next, we turn our attention to the impact of simulation step size. Coordinates of the atoms in the system are typically produced in 1–2 fs time steps, but data "snapshots" may be recorded much less frequently than this. Indeed, order to limit the size of the trajectory files, data may be recorded less frequently than is actually desirable for subsequent analyses.

Our simulations were run at a time step of 1–2 fs. This yields extremely large data sets, which can be "down-sampled" for a first-pass analysis. To determine the effect of simulation step size, we subsampled the data by taking every nth frame, where $n = \{20, 200, 2000\}$. Results for the protein data set at 12-bit and 16-bit quantization are shown in Figure 3.

The linear and delta schemes perform best at all subsampling levels, with delta gaining a small advantage over linear prediction as the subsampling increases. This is expected, because subsampling destroys frame-to-frame coherence and predicting the new position as the old is likely to be as good, or better, than any other scheme. As the sampling interval increases, the compression ratios decrease monotonically at between 5 and 10% for each additional factor of 10 increase.

To further clarify the impact of simulation step size, we have shown compression results for all our data subsampled at 200 fs time steps—Table 3. Although the compression performance decreases with simulation step size, our schemes continue to outperform the benchmark methods.

Run-times are important when balancing fidelity against compression gains, because better compressors often require long run times. Some representative run times for our methods are given in Table 2. This table shows that, even for massive data sets, the times are not exorbitant, and that our predictors show improvement over the benchmark BZip2 application. The compression and decompression times are somewhat asymmetric. For our predictors (delta, linear, and spline), decompression generally takes longer than compression. Initially the difference ranges from 20 to 35% but diminishes as quantization level increases. Note that BZip2 takes longer to compress than to decompress.

A further aspect of our compression analysis concerns the possibility of exploiting the properties of water to bolster compression gains in water-rich simulations. The results of applying our water model to two data sets—"Protein-normal" and "pure water"—with varying proportions of solvent (water) are shown in Figure 4. We also present results for the benchmark BZip2 compressor.

It is immediately apparent that the water compressor, "Bonds," only performs marginally better than BZip2, which is largely unaffected by the constituent atoms comprising a data set (the two graphs for Bzip2 are essentially the same). Somewhat surprisingly, as the proportion of solvent increases, the performance of the Bonds compressor actually decreases. This can be attributed to the structuring of the first layer of solvent water molecules by the protein solute, which is present in protein system, but not in pure water. This results in the first layer of solvent waters being more structured and hence more easily predictable. The results demonstrate that in its current form, the Bonds compressor is not competitive with our interframe predictors, although it does still improve on bzip2. If one does not need the atom indexing to be preserved across frames, then compression performance increases by 3–5%.

The final aspect of the simulation, which might impact on compressibility, is the simulation temperature. We ran the experiments on the protein data set for both low (310 K) and high (500 K) temperatures to assess what impact this had on compression. This effects all schemes except bzip2, which does not rely on higher-order information. The affect is, however, rather minor, only becoming apparent when the original simulation is heavily down-sampled. In the case we evaluated, the reduction in compression ratio varies from 2.5% at 12-bit quantization to 4.8% at 20-bit quantization. It is debatable whether this latter case corresponds to a plausible scenario, however, as it is unlikely that great precision will be required in atom positions if only 1 in every 1000 frames is retained.

Conclusions

We have developed and tested a number of lossy, quantized compression algorithms designed specifically for MD trajectory files: interframe predictors that exploit temporal coherence between successive frames and an intraframe scheme that compresses each frame independently and exploits the fairly rigid geometry of water molecules in MD simulations. These compression techniques satisfy the identified requirements of high compression rates, low computational and memory overheads, streaming, configurability with regard to error bounds, and simplicity of implementation.

Interestingly, first-order (linear) predictors tend to, on average, beat higher-order polynomial prediction for small time step simulations. Linear predictors require only three simulation frames and therefore do not need a complex disk-based solution, even for very large simulations. The results for the intraframe water compressions schemes are less compelling, only showing a useful improvement if atom ordering is discarded.

We demonstrate clear improvement over previously developed MD compression schemes where the compression of high fidelity (small time step) simulations is required. In all cases, we outperform the generic BZip2 compressor, which is often used to achieve some level of compression for large data sets. We find the interframe encoders to be fast, space-efficient and well suited to the compression of massive data sets that cannot fit in memory. Where intervals between successive frames in the trajectory file are large, a simple delta predictor performs best. For high fidelity compression, our linear frame predictor gives the best results at very little computational cost: at moderate levels of quantization (max error $\approx$10 Å), we can compress a trajectory file to 5–8% of its original size. This accuracy is more than adequate for previews of data and for some analyses. However, if necessary, we do support close to full floating point precision, albeit with lower resulting compression ratios. Our scheme is thus configurable and supports different levels of compression, depending on the users' accuracy requirement.

The methods we propose scale to massive trajectory files, because they require at most three simulation frames to be memory resident at any given time. Thus, they can be used to facilitate the compression of trajectory files with small simulation time steps, as they are generated, producing a significantly smaller data buffer that can be streamed to a fast disk array with less impact on total simulation time.

There are several avenues for future work. To improve the performance of predictors, one could run a low fidelity MD simulation rather than using simple trajectory extrapolation. Of course, this would need to be a localized model to make it viable and would inflate compression times significantly. In addition, each trajectory can be predicted independently, making the compression problem eminently suitable for parallelization. There is, however, a space trade-off involved: processing trajectories completely independently would, naively, require maintaining a separate arithmetic coder state for each trajectory. Finally, one can investigate predictors that exploit the known spatial relationships of atoms within a larger molecule. Once a particular atom has been identified, the relative positions of its neighbors are also constrained and can be more efficiently predicted.

Acknowledgments

The authors thank John Stone of the University of Illinois at Urbana-Champaign for valuable early discussion and Nean Mathai of the University of Cape Town for providing the simulation data.

Keywords: Keywords: MD trajectory files · compression · arithmetic coding · interframe prediction

How to cite this article: P. Marais, J. Kenwood, K. Carruthers Smith, MM. Kuttel J. Gain, *J. Comput. Chem.* **2012,** *00,* 000–000. DOI: 10.1002/jcc.23050

[1] A. Drozdek. Elements of Data Compression, 1st ed.; Brooks/Cole Publishing Co.: Pacific Grove, CA, **2001**.
[2] S. Gumhold, Z. Kami, M. Isenburg, H.-P. Seidel, In ACM SIGGRAPH05 Sketches, **2005**.
[3] J. Seward, BZip2, **1996**.
[4] T. Meyer, C. Ferrer-Costa, A. Pérez, M. Rueda, A. Bidon-Chanal, F. J. Luque, C. Laughton, M. Orozco. *J. Chem. Theory Comput.* **2006**, *2,* 251.
[5] A. Kumar, Y.-C. Tu, In Proceedings of the Fourth SIGMOD PhD Workshop on Innovative Database Research, **2010**; ACM Press, New York, pp. 13–18.
[6] O. Devillers, P.-M. Gandoin, In Proceedings of the Conference on Visualization '00, **2000**; IEEE Computer Society Press, CA, pp. 319–326.

[7] B. Merry, P. Marais, J. Gain, In Proceedings of the 4th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa, **2006**; ACM Press, New York, pp. 15–20.
[8] A. Omeltchenko, T. J. Campbell, R. K. Kalia, X. Liu, A. Nakano, P. Vashishta, *Comput. Phys. Commun.* **2000**, *131,* 78.
[9] A. Moffat, R. M. Neal, I. H. Witten, *ACM Trans. Inf. Syst.* **1998**, *16,* 256.
[10] E. Bodden, M. Clasen, J. Kneis. Arithmetic coding revealed—a guided tour from theory to praxis, Technical Report 2007–5, Sable Research Group, McGill University, May **2007**.
[11] P. M. Fenwick, *Softw. Pract. Exper.* **1994**, *24,* 327–336.
[12] K. Maly, *Commun. ACM,* **1976**, *19,* 409–415.
[13] G. Farin, Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide, 4th ed.; Academic Press, San Diego, CA, **1997**.
[14] M. F Chaplin, *Biophys. Chem.* **2000**, *83,* 211.
[15] E. W. Dijkstra, *Numer. Math.* **1959**, *1,* 269.
[16] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, K. Schulten, *J. Comput. Chem.* **2005**, *26,* 1781.
[17] A. D. MacKerell, D. Bashford, M. Bellott, R. L. Dunbrack, J. D. Evanseck, M. J. Field, S. Fischer, J. Gao, H. Guo, S. Ha, D. Joseph-McCarthy, L. Kuchnir, K. Kuczera, F. T. K. Lau, C. Mattos, S. Michnick, T. Ngo, D. T. Nguyen, B. Prodhom, W. E. Reiher, B. Roux, M. Schlenkrich, J. C. Smith, R. Stote, J. Straub, M. Watanabe, J. Wiorkiewicz-Kuczera, D. Yin, M. Karplus, *J. Phys. Chem. B,* **1998**, *102,* 3586.
[18] E. Hatcher, O. Guvench, A. D. MacKerell, *J. Phys. Chem. B.,* **2009**, 113, 12466.

Received: 23 February 2012
Revised: 10 May 2012
Accepted: 2 June 2012
Published online on