Computer Physics Communications 131 (2000) 78–85

Computer Physics Communications

www.elsevier.nl/locate/cpc

Scalable I/O of large-scale molecular dynamics simulations: A data-compression algorithm

Andrey Omeltchenko, Timothy J. Campbell, Rajiv K. Kalia, Xinlian Liu, Aiichiro Nakano *, Priya Vashishta

Concurrent Computing Laboratory for Materials Simulations, Department of Computer Science, Department of Physics and Astronomy, Louisiana State University, Baton Rouge, LA 70803-4020, USA

Received 28 December 1999

Abstract

Disk space, input/output (I/O) speed, and data-transfer bandwidth present a major bottleneck in large-scale molecular dynamics simulations, which require storing positions and velocities of multimillion atoms. A data compression algorithm is designed for scalable I/O of molecular dynamics data. The algorithm uses octree indexing and sorts atoms accordingly on the resulting space-filling curve. By storing differences of successive atomic coordinates and using an adaptive, variable-length encoding to handle exceptional values, the I/O size is reduced by an order-of-magnitude with user-controlled error bound.

1. Introduction

Molecular dynamics (MD) simulations [1] have played a key role in our understanding of various phenomena in physics, chemistry, biology, and materials sciences [2]. In the MD approach, one obtains the phase-space trajectories of the system (positions and velocities of all atoms at all time) from the numerical solution of Newton's equations. Recent advances in scalable multiresolution algorithms [3] coupled with access to massively parallel computers have enabled very large MD simulations involving 10–100 million atoms [3–5].

These large-scale MD simulations present an enormous challenge from the standpoint of data management including input/output (I/O). The I/O problem is particularly serious, when computing and mass-storage resources are geographically distributed and data transfer is required through wide area networks [6]. A 100-million-atom MD simulation produces approximately 5 gigabytes (GB) of data per frame to store atom types, coordinates, and velocities. This amounts to 17 terabytes (TB) per day if the simulation runs for 35,000 steps [7] and data are saved after every 10 steps. Often additional atomic attributes such as atomic tensor components are stored, and make the I/O size significantly larger. Such massive data have made I/O a major bottleneck, and optimization of the I/O performance has become essential in the design of MD software.

A number of systems- and application-level approaches have been applied to the I/O problem [8]. At the application-software level, a promising approach uses data compression [9,10], optimized to the application's data and algorithmic structures. A data com-

* Corresponding author. E-mail: nakano@bit.csc.lsu.edu.

0010-4655/00/$ – see front matter
PII: S0010-4655(00)00083-7

pression scheme for MD configurations should satisfy the following requirements:

(1) *Compression with user-controlled error bound* – I/O size is reduced significantly (by an order-of-magnitude), while the user can prescribe the accuracy;
(2) *Outlier tolerance* – exceptional values are handled correctly;
(3) *Scalability* – both I/O size and computation for compression scale as $O(N)$ for an N-atom system;
(4) *Light weight* – minimal computation and memory overheads are involved;
(5) *Portability* – the source code and the data format are portable to different architectures;
(6) *Compatibility* – the algorithm can be incorporated into an existing MD code as a plug-in.

In this paper, the I/O problem is addressed using a scalable data-compression scheme. The scheme converts MD data to integers by dividing them by a user-specified error bound. An adaptive, variable-length encoding scheme is used to store these integers with minimal number of bits, while handling outliers. The algorithm also uses octree indexing of three-dimensional coordinates [11–13]. Atoms are sorted on the resulting fractal-like, space-filling curve. By storing differences between successive atomic coordinates [14,15], the I/O requirement with a given error tolerance level reduces from $O(N \log N)$ to $O(N)$. The compression algorithm is implemented in the C language, and is portable. It has been incorporated in existing MD programs written in FORTRAN. The next section contains the description of the compression algorithm. Numerical results given in Section 3 exhibit an order-of-magnitude reduction of I/O size. Finally Section 4 includes discussions.

2. Method

Our compression algorithm for MD configurations (atomic positions, velocities, and other attributes such as atomic stress components for N atoms) consists of the following major steps:

(1) Quantize all double-precision data by dividing them by a user-specified error bound (see Section 2.1);
(2) Compute the octree index, R_i, of atomic positions for all atoms, $i = 1, \ldots, N$ (Section 2.2);
(3) Sort the atoms in ascending order of R_i (Section 2.3);
(4) Store differentiated $\Delta R_i = R_i - R_{i-1}$, velocities, and other data using an adaptive, variable-length encoding (Section 2.4).

These algorithmic steps are described in Sections 2.1–2.4, and some implementation issues are discussed in Section 2.5.

2.1. Data quantization and accuracy requirements

In MD simulations, Newton's equations of motion for a set of N atoms are integrated by discretizing time with an interval, Δt, and applying a finite-difference integrator [1]. At each time step, the state of the system is completely specified by an MD configuration – $3N$ atomic positions and $3N$ velocities. As the system evolves in time, a sequence of MD configurations (i.e. an MD trajectory) is generated. Typical runs range from 10^4 to 10^6 steps, but usually it is sufficient to output the trajectory data after every few time steps for post-processing and visualization.

In addition to atomic positions and velocities, an MD configuration usually consists of tags to specify the type of each atom and other information. In the MD code considered in this paper, the resulting storage is 56 bytes for each atom, i: atomic position – $\vec{r}_i = (x_i, y_i, z_i)$, 3*sizeof(double) = 24 bytes; atomic velocity – $\vec{v}_i = (v_{ix}, v_{iy}, v_{iz})$, 3*sizeof(double) = 24 bytes; atom id, 1*sizeof(double) = 8 bytes. The fractional accuracy for a double precision number is $\sim 10^{-15}$. This level of accuracy is often necessary for numerical computation but not for I/O as is discussed below. Therefore compression can be achieved by cutting the precision, i.e. by adopting a lossy compression [9,10].

Distribution of atomic positions is characterized by a typical nearest-neighbor distance, a. In materials simulations, separation between atoms cannot become much smaller than a due to strong steric repulsion between atoms at small distances [16]. Atomic positions, $\vec{r}_i$, are constrained within the MD box of size $(L_x \times L_y \times L_z)$, i.e. $0 < x_i < L_x$, $0 < y_i < L_y$, $0 < z_i < L_z$. Thus $\vec{r}_i$ assume larger values as the system size increases (they are not scalable). This is in contrast to velocities, $\vec{v}_i$, which in equilibrium follow the Maxwell distribution. In general, there is no explicit size dependence in velocity distribution.

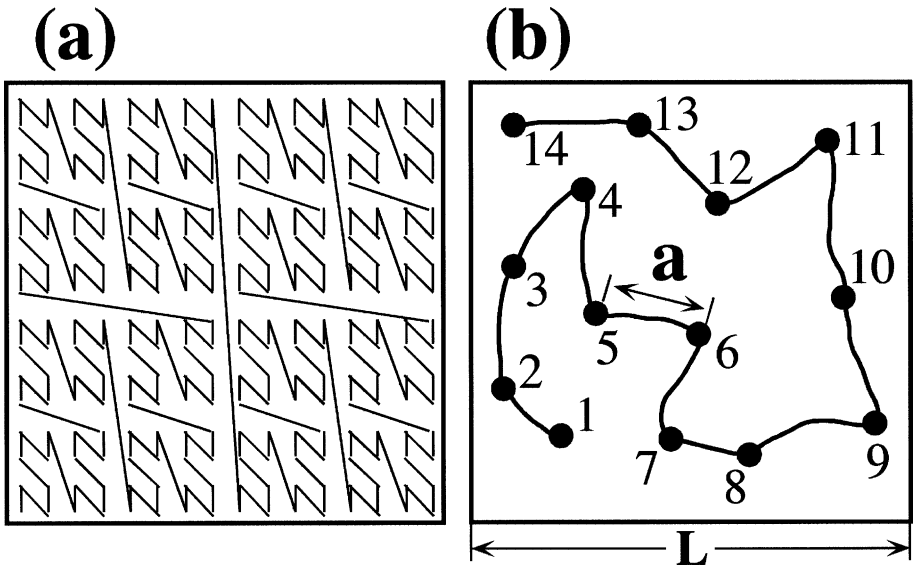

Fig. 1. (a) A space-filling curve based on octree indexing maps the three-dimensional space into a sequential list, while preserving spatial proximity of consecutive list elements. (The panel shows a 2D example.) (b) Atoms are sorted along the space-filling curve and only relative positions of successive atoms are stored.

To efficiently store an MD configuration, one has to decide to what extent the original double precision data be reproducible. For example, the precision needs not exceed that of the numerical integration for storing a configuration as a start-point of another run. A reasonable estimate for the tolerance on positions and velocities is obtained by first setting an error tolerance for the energy on the order of the total energy fluctuation resulting from the integration algorithm. A conservative estimate for the energy tolerance is given by $\delta E \sim 10^{-4}$ eV, which corresponds to temperature tolerance of $\sim$1 K. The resulting tolerance values for positions and velocities are $\delta r \sim 10^{-4}$ Å and $\delta v \sim 10^{-4}$ Å/fs, respectively.

Given the accuracy requirements, it is possible to divide the double precision data by an appropriate tolerance and switch to an integer representation: $\vec{r}_i \leftarrow \mathrm{nint}(\vec{r}_i/\delta r)$, and $\vec{v}_i \leftarrow \mathrm{nint}(\vec{v}_i/\delta v)$, where "nint" stands for the nearest integer.

2.2. Space-filling curve

The number of bits for representing a position, $\vec{r}_i$, is approximately

$$\log_2(L_x/\delta r) + \log_2(L_y/\delta r) + \log_2(L_z/\delta r) = \log_2(\Omega/\delta\Omega), \quad (1)$$

where $\Omega = L_x L_y L_z$ is the volume of the MD box and $\delta\Omega = \delta r^3$. Consider a path through all the atoms which only connects neighbor atoms and assign an index to each atom according to its position on the path, as illustrated in Fig. 1(b). One can then store the relative positions, $\Delta\vec{r}_i = \vec{r}_i - \vec{r}_{i-1}$, rather than the absolute values. Assuming that the atoms are distributed uniformly with a nearest-neighbor distance a, the average number of bits to store $\Delta\vec{r}_i$ is roughly

$$3\log_2(a/\delta r) = \log_2\left[(\Omega/\delta\Omega)(a^3/\delta\Omega)\right] \sim \log_2(\Omega/\delta\Omega) - \log_2 N. \quad (2)$$

Thus, the required number of bits per atom is reduced by $\log_2 N$ and is independent of the system size. Note that this reduction is achieved by abandoning the original order of atoms (i.e. the atomic array index), which in most cases is not necessary to preserve [17]. (The information associated with a given order is $\log_2 N$ per atom.)

Appropriate ordering of atoms may be accomplished by arranging them according to their position

on a space-filling curve [11–13]. A space-filling curve is a mapping of a one-dimensional array to three-dimensional grid points, which preserves the spatial proximity of successive array elements [13]. Fig. 1(a) shows a space-filling curve (the so-called Z-curves) in two dimensions.

Points on a Z curve are identified using an octree index, R_i, constructed by interleaving bits of x_i, y_i, and z_i. The number of bits to represent the x, y, or z coordinate is

$$l_x = \lceil \log_2(L_x/\delta r) \rceil, \quad l_y = \lceil \log_2(L_y/\delta r) \rceil,$$

or

$$l_z = \lceil \log_2(L_z/\delta r) \rceil,$$

respectively. If the same number of bits, $l = \max(l_x, l_y, l_z)$, is used to represent all coordinates, the octree index R_i is easily computed by interleaving the bits of the position components, as illustrated in Fig. 2(a). In the example, $l = 7$ bits and the resulting octree index is $3l = 21$ bits.

The above procedure is not suitable when one of the components is represented by a much smaller number of bits than the others (e.g., for a simulation of a thin plate). The above scheme requires $l_R = 3\max(l_x, l_y, l_z)$ bits, while $l_R = l_x + l_y + l_z$ bits are sufficient to encode each $\vec{r}_i$. We modify the encoding procedure to interleave only "useful" bits. The generalization is straightforward, as shown in the example in Fig. 2(b). For, $l_x = 5$, $l_y = 7$, and $l_z = 3$, the resulting octree index is $5 + 7 + 3 = 15$ bits. The computation of the octree index is implemented using bitwise "`and`", "`or`", and "`shift`" operators.

(a)

x_i =	1	1	0	1	0	1	1
y_i =	0	0	1	1	1	1	0
z_i =	1	0	1	0	0	0	1
R_i =	101	001	110	011	010	011	101

(b)

x_i =			1	1	0	1	1
y_i =	1	0	1	1	1	1	0
z_i =					0	1	0
R_i =	1	0	11	11	010	111	001

Fig. 2. (a) A 21-bit octree index, R_i, obtained by interleaving three 7-bit Cartesian coordinates, x_i, y_i, and z_i. (b) A generalized octree index for bit lengths $l_x = 5$, $l_y = 7$, and $l_z = 3$ consists of $5 + 7 + 3 = 15$ bits.

2.3. Sorting

Atoms are ordered along the Z curve by sorting them in ascending order of R_i. It is well known that the best comparison-based sorting algorithm requires $O(N \log N)$ operations [18]. Radix-based sorting of integers scales as $O(Nl)$, where l is the bit-number [18]. Computation time in radix sorting, however, involves a large pre-factor proportional to l. The sorting procedure can be further optimized for MD configurations, since atoms cannot concentrate above a certain density and the distribution of atoms is uniform within sufficiently small volumes. Octree indices are thus uniformly distributed in most of its range, $[R_{\max}, R_{\min}]$.

The sorting algorithm used in this paper consists of two stages. The first stage sorts R_i, into N bins of equal size, $(R_{\max} - R_{\min})/N$, in $O(N)$ operations [19]. If the distribution of R_i is uniform, each bin contains $n \sim 1$ integers. In the second stage, n integers are sorted within each bin. For efficiency, the cases of $n \leqslant 3$ are programmed explicitly, while $n > 3$ is handled with the heapsort algorithm [18]. The resulting algorithm is $O(N)$ for a uniform distribution and $O(N \log N)$ in the worst case.

To compress a large MD configuration, it is advantageous to separate the system into smaller subsystems. This is accomplished by dividing the MD box into several boxes of limited size, D. This procedure effectively reduces the bit-number of the integer atomic positions, $\vec{r}_i$, since the common high-order bits may be stored in the header, while only $\lceil \log_2(D/\delta r) \rceil$ bits remain to be stored.

2.4. Adaptive, variable-length encoding

The final task is to store integer representations of atomic attributes – differentiated $\Delta R_i = R_i - R_{i-1}$, velocities, and other data – with reduced number of bits. This requires encoding an array of unsigned integers $(x_1, x_2, \ldots, x_N)$ with the corresponding bit-numbers $(l_1, l_2, \ldots, l_N)$ following an unknown distribution.

To take advantage of the fact that most l_i fall into a limited range with existence of a small number of x_i with arbitrary large values ("outliers"), we use a variable-length encoding algorithm (see Appendix A). This algorithm involves two parameters: l and Δl. Initially, l bits are allocated to store each l_i. If $l_i > l$, then additional bits are allocated incrementally in units of Δl until all the bits in l_i are stored. The resulting sequence of bits can be decoded using status bits to decide how many bits to read next. The status bit 1 means that there are more bits to be read, while the status bit 0 terminates further allocation.

Examples of such variable-length encoding are given below for $l = 3$, $\Delta l = 2$, where the status bits are preceded with (ˆ): $l_i = 10 \to$ ˆ0 010; $l_i = 1100 \to$ ˆ1 001ˆ0 10; $l_i = 1010111 \to$ ˆ1 111ˆ1 01ˆ0 01. (Note that the bits are written starting from the lowest bit.)

In order to minimize the storage size, the variable-length encoding scheme is made adaptive by varying l and Δl according to the data being encoded. The minimization procedure is facilitated by defining the following variables: L^+ – number of extra bits (status and padding) resulting from l being too small; L^- – number of extra bits (status and padding) resulting from l being too large; ΔL^+ – number of extra bits (status and padding) resulting from Δl being too small; ΔL^- – number of extra bits (status and padding) resulting from Δl being too large. The following examples demonstrate how these variables are computed:

(i) $0 \to$ ˆ0 010 – l being too long results in one extra "0" padding bit, so that $L^- = 1$, $L^+ = \Delta L^+ = \Delta L^- = 0$;

(ii) $1100 \to$ ˆ1 001ˆ0 10 – l being too short results in one extra status bit, and Δl being too short leads to an extra padding bit, so that $L^+ = 1$, $\Delta L^+ = 1$, $L^- = \Delta L^- = 0$;

(iii) $1010111 \to$ ˆ1 111ˆ1 01ˆ0 01 – l being too short results in two extra status bits, and there are no padding bits, so that $L^+ = 2$, $L^- = \Delta L^+ = \Delta L^- = 0$.

The adaptive encoding scheme works as follows: As the data is being encoded, the difference between L^+ and L^- is accumulated as

$$L = \sum_i \left(L_i^+ - L_i^-\right),$$

and similarly

$$\Delta L = \sum_i \left(\Delta L_i^+ - \Delta L_i^-\right).$$

Whenever L exceeds a certain tolerance ($L > L_{\max}$) the encoding parameter l is incremented by one and the value of L is reset to zero. Similarly, l is decremented when $L < -L_{\max}$. The parameter Δl is adjusted in the same fashion, i.e. incremented or decremented when $\Delta L > \Delta L_{\max}$ or $\Delta L < -\Delta L_{\max}$, respectively. We choose $L_{\max} = \Delta L_{\max} = 32$, but the algorithm's performance is not sensitive to these parameters.

Though this heuristic algorithm involves little computation, it provides reasonable compression (see numerical results in Section 3). If all l_i are identical, only one additional bit per integer is required. Otherwise, the overhead is usually two to three bits per integer, depending on the data.

2.5. Implementation

The compression algorithm is implemented in the C language. An interface to an MD code, which is written in FORTRAN, is provided by a limited set of functions callable from a FORTRAN program. These routines enable the user to specify the types of data to be written along with corresponding accuracy tolerances, to format the MD data into the data structures used by the compression code, and to read/write the data to a file in the compressed format.

The user is allowed to select the data fields to be stored for each atom. The only required data are the atomic positions. The data to be stored may be either double precision (e.g., velocities, forces, charges), or integer (various tags attached to atoms). The user is responsible for choosing the accuracy requirements for each data type either experimentally or using estimates similar to those given in Section 2.1. The code is portable to computers that use 32-bit integers. The data format is also made portable by converting all integers to the network byte order [20] before writing them to a file.

To ensure that the code is efficient and portable, a simple bit-level I/O interface has been developed using bitwise operators in C. Routines are provided to open a bit-level stream for reading from or writing to a string or a stream. Once a bit-stream is open, the

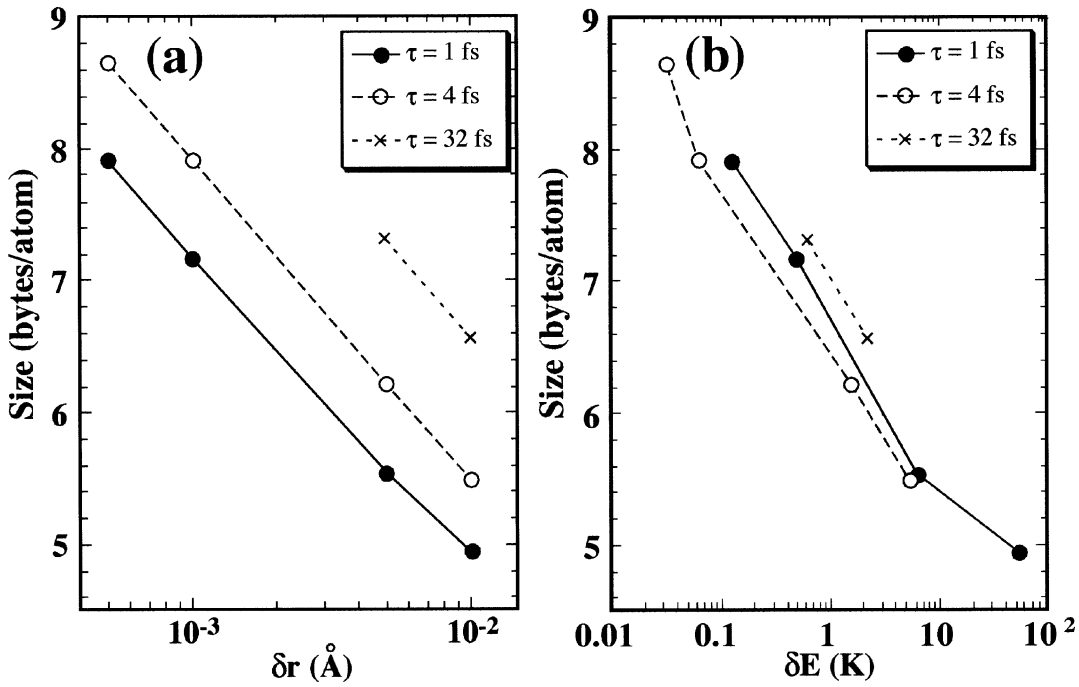

Fig. 3. Size (per atom) of compressed MD configurations: (a) effect of the tolerance parameters δr and τ, (b) relationship between the storage size and the error in total energy introduced by compression.

user can `put` or `get` an arbitrary number (up to 32) of lowest bits in an integer. The resulting storage is not aligned on byte boundaries. The `put` and `get` functions are implemented as preprocessor macros to improve efficiency.

3. Numerical results

The compression algorithm has been tested on a Digital AlphaServer 2100 for several MD configurations.

First, a diamond grain-boundary configuration containing 46,440 atoms at a temperature of 300 K is considered. In the original MD code, the storage was 56 bytes/atom which include positions and velocities in double precision and an 8-byte tag used for multiple purposes. To compress this data, one has to specify the position tolerance, δr, and the time parameter, τ. (The velocity tolerance is computed as $\delta r/\tau$.) Fig. 3(a) shows the amount of compressed storage per atom as a function of the position tolerance, δr, for three different values of τ. Significant reduction in storage is achieved by increasing δr. However, the information loss due to compression results in an error in the total energy, as is shown in Fig. 3(b). Assuming that an energy deviation of 1 K is acceptable, the minimum storage size is achieved with $\delta r = 5 \times 10^{-3}$ Å and $\tau = 4$ fs. (These values are used for further tests.) Fig. 3(a) demonstrates that the present compression scheme reduces the required storage size by nearly an order-of-magnitude, from 56 to 6.22 bytes/atom.

Scalability of the algorithm, is tested by measuring the compressed size for a sequence of system sizes ranging from 27,396 to 2,248,704 atoms, see Table 1. The resulting number of bytes per atom shows little dependence on the number of atoms, i.e. the scheme is scalable.

To test the applicability of the algorithm, we first consider a highly irregular configuration from a diamond-impact simulation [21]. The system consists of a 36,000-atom diamond cluster hitting a million-atom diamond film. The temperature in the system is also non-uniform: the initial temperature of the film is 300 K, while the temperature near the point of the impact is higher by more than an order of magnitude.

Table 1
Scaling of the compressed storage size with the number of atoms

Number of atoms	4392	38,136	281,088	2,248,704
Size (bytes/atoms)	6.238	6.206	6.200	6.200

Table 2
Compressed storage size as a function of temperature for amorphous and liquid carbon configurations

Temperature (K)	300	500	1000	2000	5000 (liquid)
Size (bytes/atoms)	6.185	6.318	6.515	6.557	6.915

Table 3
Error in the total energy introduced by the compression algorithm for diamond cluster system

Temperature (K)	500	1000	2000	3000
Energy change (K/atoms)	1.16	1.06	0.78	1.28

The algorithm is applicable in this situation as well, and results in 6.24 bytes/atom of storage.

Other types of MD configurations are also tested (the storage per atom is given in parenthesis): a two-dimensional graphite sheet with a propagating crack (6.21 bytes/atom) [22] and a million-atom Si_3N_4-coated Si mesa on Si substrate (6.56 bytes/atom) [23]. The results for disordered amorphous and liquid carbon systems at various temperatures are listed in Table 2. The storage size increases only slightly with temperature.

To decide whether the position tolerance should be adjusted with temperature, the compression error in the total energy has been computed for a range of temperatures in a small diamond-cluster system using the same tolerance parameters ($\delta r = 5 \times 10^{-3}$, $\tau =$ 4 fs). Table 3 shows that the error in the total energy is nearly independent of temperature. Thus the user will not have to adjust the compression parameters for different temperatures.

Finally, it is found that the computation time used for compression is negligible compared with typical simulation time.

4. Discussion

Based on a few simple ideas, including integer representation with controlled accuracy and octree ordering of atoms, a compression algorithm is designed to reduce the I/O size of MD simulations. An adaptive, variable-length encoding scheme is used to make the scheme tolerant to outliers. A flexible interface to MD codes is provided, and significant improvement in the I/O performance is achieved for actual MD data. The compression scheme can also be used for in-core compressed storage to minimize the use of RAM, since the algorithm is not compute-intensive. The in-core compressed storage would allow to extend accessible system sizes when the main memory size is a limitation.

Acknowledgments

This work is supported by NSF (Grant No. ASC-9701504 and DMR-9711903), ARO (Grant No. DAAH04-96-1-0393), NASA (Grant No. NCC 2-5320 and NAG2-1318), DOE (Grant No. DE-FG02-96ER45570), AFOSR (Grant No. F 49620-99-1-0250 and F 49620-98-1-0086), and USC-LSU MURI (Grant No. F 49620-95-1-0452). Simulations were performed at the Concurrent Computing Laboratory for Materials Simulations at Louisiana State University and on parallel computers at the DoD Major Shared Resource Centers under a DoD Challenge Applications Award.

Appendix A. Variable-length encoding algorithm

Input: an unsigned integer, x_i $(0 \leqslant x_i < 2^{32} - 1)$
Output: a sequence of bits

Step 1. Compute $l_i = \begin{cases} \lceil \log_2(x_i) \rceil, & x_i > 0; \\ 0, & x_i = 0. \end{cases}$

Step 2. If $l_i \leqslant l$ then
output one "0" bit ("status bit")
to indicate that x_i fits into l bits
output x_i using l bits
(i.e. l_i useful bits $+(l - l_i)$ extra "0" bits)
encoding is now complete
else
output one "1" status bit to indicate that
the number does not fit into l bits
output l lowest bits of x_i ($(l_i - l)$ bits
still remain to be written)

Step 3. Repeat until all the bits of x_i are written
if no more than Δl bits remain then
output one "0" status bit
output remaining bits plus "0"
padding to a total of Δl bits;
else
output one "1" status bit
output Δl of the remaining bits

References

[1] M.P. Allen, D.J. Tildesley, Computer Simulation of Liquids (Oxford Univ. Press, Oxford, 1987).

[2] A. Pechenik, R.K. Kalia, P. Vashishta, Computer-Aided Design of High-Temperature Materials (Oxford Univ. Press, Oxford, 1999).

[3] A. Nakano, R.K. Kalia, P. Vashishta, Comput. Sci. Eng. 1 (5) (1999) 39;
P. Vashishta, R.K. Kalia, A. Nakano, ibid. 1 (5) (1999) 56.

[4] F.F. Abraham, IEEE Comput. Sci. Engrg. 4 (2) (1997) 66.

[5] S.J. Zhou, D.M. Beazley, P.S. Lomdahl, B.L. Holian, Phys. Rev. Lett. 78 (1997) 479.

[6] I. Foster, C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure (Morgan Kaufmann, San Francisco, 1999).

[7] A 100-million-atom MD simulation on silica material runs at a speed of 35,000 steps per day on 1,024 Cray T3E processors.

[8] D.E. Womble, D.S. Greenberg, Parallel Comput. 23 (1997) 403, and the other articles in the same special issue on parallel I/O.

[9] G. Held, Data and Image Compression, 4th edn. (John Wiley & Sons, New York, 1996).

[10] D. Salomon, Data Compression, The Complete Reference (Springer, New York, 1997).

[11] J.P. Singh, C. Holt, T. Totsuka, A. Gupta, J. Hennessy, J. Parallel Distrib. Comput. 27 (1995) 118.

[12] M.S. Warren, J.K. Salmon, Comput. Phys. Commun. 87 (1995) 266.

[13] J.R. Pilkington, S.B. Baden, IEEE Trans. Parallel Distrib. Sys. 7 (1996) 288.

[14] B. Gottlieb, S.E. Hagereth, P.G.H. Denot, H.S. Rabinowitz, Proc. Nat'l. Comput. Conf. 44 (1975) 453.

[15] K. Sayood, K. Robinson, IEEE Trans. Signal Processing 40 (1992) 236.

[16] P. Vashishta, R.K. Kalia, A. Nakano, W. Li, I. Ebbsjö, in: Amorphous Insulators and Semiconductors, M.F. Thorpe, M.I. Mitkova (Eds.) (Kluwer, Dordrecht, 1996) p. 151.

[17] The order of atoms is necessary to preserve in some MD simulations. In macromolecular simulations, for example, the order of atoms often defines the topology of molecules. See B.R. Brooks et al., J. Comput. Chem. 4 (1983) 187;
W.D. Cornell et al., J. Amer. Chem. Soc. 117 (1995) 5179.

[18] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms (MIT Press, Cambridge, 1990).

[19] The size of the bin is adjusted to the closest power of 2, so that integer divisions may be replaced by more efficient shift operations.

[20] W.R. Stevens, Unix Network Programming, Vol. 1, 2nd edn. (Prentice-Hall, Upper Saddle River, 1998).

[21] A. Nakano, M.E. Bachlechner, T.J. Campbell, R.K. Kalia, A. Omeltchenko, K. Tsuruta, P. Vashishta, S. Ogata, I. Ebbsjö, A. Madhukar, IEEE Comput. Sci. Engrg. 5 (4) (1998) 68.

[22] A. Omeltchenko, J. Yu, R.K. Kalia, P. Vashishta, Phys. Rev. Lett. 78 (1997) 2148.

[23] A. Omeltchenko, M.E. Bachlechner, A. Nakano, R.K. Kalia, P. Vashishta, I. Ebbsjö, A. Madhukar, P. Messina, Phys. Rev. Lett. 84 (2000) 318;
M.E. Bachlechner, A. Omeltchenko, A. Nakano, R.K. Kalia, P. Vashishta, I. Ebbsjö, A. Madhukar, Phys. Rev. Lett. 84 (2000) 322.