SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2017; **00**:1–19
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Scalable processing and autocovariance computation of big functional data

Nieves R. Brisaboa[1], Ricardo Cao[2], José R. Parama[1,*]and Fernando Silva-Coira[1]

[1] *Universidade da Coruña, LBD group, Computer Science Department, Facultade de Informática, CITIC, Campus de Elviña, 15071 A Coruña, Spain*
[2] *Universidade da Coruña, MODES group, Mathematics Departament, Facultade de Informática, CITIC, Campus de Elviña, 15071 A Coruña, Spain*

SUMMARY

This paper presents two main contributions. The first is a compact representation of huge sets of functional data or trajectories of continuous time stochastic processes, which allows keeping the data always compressed, even during the processing in main memory. It is oriented to facilitate the efficient computation of the sample autocovariance function without a previous decompression of the data set, by using only partial local decoding. The second contribution is a new memory efficient algorithm to compute the sample autocovariance function.
The combination of the compact representation and the new memory efficient algorithm obtained in our experiments the following benefits. The compressed data occupy in disk 75% of the space needed by the original data. The computation of the autocovariance function used up to 13 times less main memory, and run 65% faster than the classical method implemented, for example, in the R package.

Received . . .

KEY WORDS: Big data; compact representation; autocovariance function; efficient computation; functional data; R package

1. INTRODUCTION

In the last decade, we are attending to an exceptional growing demand for large-scale data analysis, which is linked to the new field called Big Data. The need to process huge collections of data poses several challenges. On one hand, statistics and artificial intelligence communities continue to develop new methods and techniques to analyze data. On the other hand, computer scientists have to adapt analytical algorithms to data sets with *data volume too large, data rate too fast, and data too heterogeneous*, the so-called *Volume, Velocity, and Variability*. In this work, we deal with the first issue: data volume too large.

Researchers or professionals working in Big Data need mastering many different techniques and skills. To facilitate their work, several packages appeared, mainly SAS (see `http://www.sas.com/`), Matlab (see `http://www.mathworks.com/products/matlab`), and R (see `http://www.r-project.org`). These packages are very useful, but they have scalability problems [1]. For example, the Installation and Administration manual of R recommends loading into main memory data sets that occupy only 10-20 % of the available RAM and warns that if the data set exceeds 50% of the available RAM, the system will be unusable due to operation overhead, even the simplest ones. The solution to these problems is in most cases the use of parallel processing

[*] Correspondence to: jose.parama@udc.es

[2, 1, 3, 4]. Parallel processing is a straightforward solution, probably due to the existence of a good set of available tools. However, while putting most of the efforts in this strategy, one is missing chances to improve the scalability by means of other techniques. The use of more evolved data structures and algorithms is losing the role that they had in the past when the hardware technology was more limited.

One of the alternatives is the *in-memory data management*, that advocate to take advantage of the higher bandwidth and lower latency of the upper levels of the memory hierarchy [5, 6, 7]. A parallel research line called *compact data structures* [8] has the same objective. Perhaps in-memory data management is more focused in traditional databases, whereas compact data structures have a broader scope, including many different data types and data structures. In any case, the target is to fit, efficiently query, and manipulate much larger data sets in main memory, and thus avoid costly disk accesses. To achieve this goal, compression techniques are used, but one must be careful when choosing the techniques. Although there are highly sophisticated compression methods, not all are valid for in-memory data management. For example, most compression methods require starting the decompression from the beginning, something unfeasible in our scenario. In addition the decompression procedure should be fast, otherwise the processing times would be excessive.

Compression of floating point numbers has been proven difficult, mainly because the data sets usually contain many distinct values and with few repetitions. These two features make sequences of floating point numbers poorly skewed and, as a consequence, the entropy of those sequences is high, making them virtually incompressible with statistical compressors. Therefore, general purpose compressors may not succeed over sequences of floating point numbers. Instead, there are compressors that take advantage of properties of the data domain. Thus, there are compressors specially designed for images, video, or sound [9, 10, 11, 12], for general scientific data [13, 14], or for more specific domains [15, 16].

Although our method can be used for trajectories of any continuous time stochastic process, in this work, we rely on the characteristics of Brownian motion to develop an in-memory compression technique especially suited for these data. Since the seminal work by Einstein [17], the Brownian motion has been extensively used to model the movements of particles subject to instantaneous imbalanced combined forces exerted by collisions. Brownian motion and related stochastic processes have been successfully used to model the movement of colloidal particles or the trajectory of pollen grains suspended in water. Over the past forty years, starting with the papers by [18], the Brownian motion and related processes have been used to model option pricing and plenty of financial time series (see, for instance, [19]).

Our method is designed to efficiently compute empirical moments from a sample of observed trajectories of the stochastic process. One example of these moments is the empirical autocovariance function, which is a very important tool for functional principal component analysis. It can be used for dimension reduction, as in the Karhunen-Loève decomposition.

Our method, called *Compact representation of Brownian Motion* (CBM) compresses sequences of 32-bit floating point numbers representing Brownian motion trajectories up to around 75% of the original size. We show that we can compute the sample autocovariance function keeping the data compressed in main memory, using only partial local decoding of the data when needed. In addition, we present a modification of the typical algorithm to compute the sample autocovariance function, which allows a much better main memory usage, and that can be used for trajectories of any continuous time stochastic process.

We compare our C++ implementation, which receives as input CBM compressed data, against: the R implementation (in fact a C program) and our own C implementation, both operating on plain data. We show that, if we use the classical method to compute the sample autocovariance function, the CBM consumes up to 3.7 times less memory than the R package, and in the order of a C program. What is more surprising is that the computation of the sample autocovariance function is around 65% faster than the C program run over the plain data, thanks to the use of the compression process to precompute values and a better usage of the memory hierarchy.

The new memory efficient algorithm to compute the sample autocovariance function obtains big memory savings, up to 13 times less than the R package. Moreover, using CBM, this version is up to 25% faster than the classical version applied over plain data.

The outline of the paper is as follows. Section 2 presents the motivation for the use of compression in the context of empirical autocovariance computation for Brownian motion trajectories. Section 3 shows some related work in the compression field. Section 4 presents CBM. Section 5 describes the algorithm to compute the sample autocovariance function that saves main memory. Section 6 shows the results of our empirical evaluation, and Section 7 presents our conclusions and directions of future work.

2. BROWNIAN MOTION AND AUTOCOVARIANCE ESTIMATION

In real life, the Brownian motion can be used to model plenty of phenomena. It can be observed in microscopic particles that, when floating in a fluid, exhibit continuous but very jittery and erratic motion, since they are continuously bombarded by the fluid molecules. This natural phenomenon was formalized by Norbert Wiener in a rigorous mathematical way, as a stochastic process with continuous time.

The Brownian motion is a notion of central importance in probability theory, and it is used as a building block for a number of related random processes that are of great importance in a variety of applications in many fields, in pure Mathematics and in Applied Mathematics. Economics is one of the main applications of Brownian motion. It is used, for example, to predict the prices of financial products. In Medicine, it has been used in image analysis. Brownian motion has many applications in Engineering. For example, it can be used to model noise in electronics and instrument error. In Physics, for example, it is used to model the movement of little particles in a fluid or a gas, like in the aerosol transport phenomena.

2.1. *Brownian trajectories*

A Brownian trajectory (or curve) is just an observation of a Brownian motion stochastic process. In practice, this can be one of the components of a 3D motion, for example, the height of the particle, $X(t)$. These values make up a trajectory, and thus a trajectory is a function that, for every time instant, t, gives a real number. In practice, time is discretized and a trajectory is also discretized as a sequence of floating point numbers. Figure 1 shows an example of several Brownian trajectories.

2.2. *Autocovariance function estimation*

Brownian trajectories are randomly observed functions, so statistical analysis of them can be included in the field of functional data analysis. This is a very active research topic in modern statistics that focuses on analyzing complex and high dimensional data structures [20, 21, 22]. Classical important problems in this field are dimension reduction and supervised classification. These can be addressed using functional principal component analysis and functional data discriminant analysis (see [20, 23] among many other). To carry out these techniques, estimation of autocovariance functions or autocorrelation functions is needed. These are the extension of covariance matrices or correlation matrices to the context of functional data.

In this work, as an example of a statistical process that operates on functional data, we consider the autocovariance function estimation for Brownian trajectories. For a collection of trajectories $X_1, X_2, \ldots, X_n$, the value each one has at time $t \in T$, is represented by $X_1(t), X_2(t), \ldots, X_n(t)$, and the autocovariance function is estimated using Equation (1):

$$\hat{C}\left(s,t\right) = \frac{1}{n}\sum_{i=1}^{n}\left(X_i\left(s\right) - \overline{X}\left(s\right)\right)\left(X_i\left(t\right) - \overline{X}\left(t\right)\right), T, \tag{1}$$

for every $s, t \in T$, where $\overline{X}\left(s\right) = \frac{1}{n}\sum_{i=1}^{n} X_i\left(s\right)$. It is clear that direct computation of (1), based on a large number of trajectories, n, for a large number, m, of instants $t \in T$ is a time-consuming

DOI: 10.1002/spe

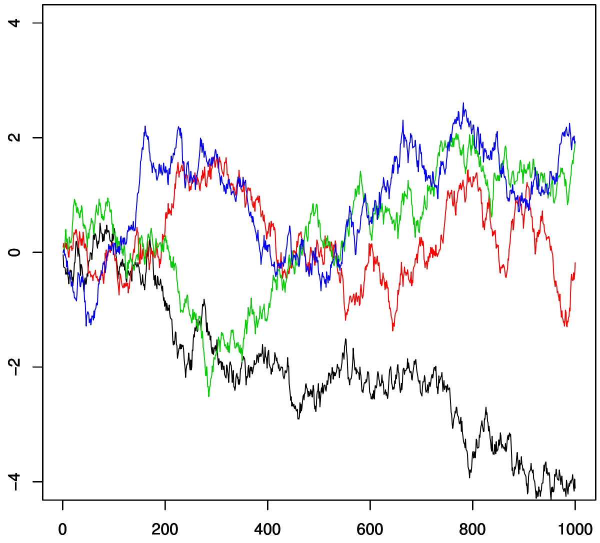


Figure 1. Four trajectories (curves) of a Brownian motion.

process and it requires plenty of memory and disk. This is an important problem in Big Data analysis.

3. RELATED WORK

3.1. Compressing Integer Numbers

Let $S = (s_1, s_2, \ldots, s_n)$ be a sequence of symbols over an alphabet Σ. A *code* is an injective function $\mathcal{C} : \Sigma \rightarrow \{0,1\}^*$, that assigns a distinct sequence of bits $\mathcal{C}(s_i)$ (codeword) to each symbol $s_i \in \Sigma$. A way to compress S is to order the symbols of Σ by their frequency in S, and assign shorter codewords to the more frequent symbols. This strategy is called *statistical encoding*, Huffman coding [24] is the best code that is univocally decodable.

If Σ is formed by integers, Huffman can be used, but if the size of Σ is large, since Huffman has to explicitly store the function $\mathcal{C}$, that space may be prohibitive. In this case, *fixed or static codes* can be used. These codes do not use the probabilities, instead, each integer is always mapped to the same codeword, that is, $\mathcal{C}$ does not depend on the exact input sequence S and therefore there is no need to store it.

Still, the main idea is the same, compression is achieved by assigning shorter codewords to the more frequent integers, and it is assumed that these numbers are the smallest integers. Therefore better compression can be obtained if the original sequence is preprocessed with relative or differencing encoding. Each integer, except the first one, is replaced by its difference with the previous one. However, the differences can be negative, this poses a problem since most codes for integers only work with positive numbers. The first solution is to store all the integers in absolute value and add 1 bit per integer to indicate whether the original integer was positive or negative. This additional bit can be avoided by using the ZigZag encoding, which maps signed integers to unsigned integers. The -1 is encoded as 1, 1 as 2, -2 as 3, and so on. The problem with this approach is that numbers with a large magnitude will have a codeword with an even greater magnitude.

Examples of fixed codes are the unary code, Elias codes (γ-codes and δ-codes) [25], or Golomb codes [26]. From the input sequence, the Golomb encoder chooses one parameter m. The codeword assigned to a source symbol s_i is composed of two parts. The first one is $q = \lfloor s_i/m \rfloor$ encoded in unary. The second part is $r = s_i - qm$ encoded in minimal binary: being $c = \lceil \log m \rceil$, the first $2^c - m$ values of r are encoded in binary using $c - 1$ bits, and the rest are encoded in binary using c bits. The Rice codes [27] are a special case of Golomb codes, in which the parameter m is chosen to be a power of two. This choice makes their computation faster, and thus Rice codes are extensively used.

The codes shown so far produce codewords of arbitrary bit lengths. This causes bit manipulations that slow down the encoding and decoding processes. To avoid this, there is a family of codes that produce codewords formed by one or more chunks of b bits, usually of 8 bits. The first example is Vbyte [28]. The $\lfloor \log s_i \rfloor + 1$ bits required to represent s_i in binary are split into blocks of $b - 1$ bits. The chunk holding the most significant bits of s_i is completed with a bit set to 0 in the highest position, whereas the rest are completed with a bit set to 1 in the highest position. Therefore, using chunks of 8 bits, the first chunk of a codeword is always a number between 0 and 127, and the rest are between 128 and 255. Therefore, we can split the byte values into two types, the *beginners* (values between 0 and 127) and the continuers (values between 128 and 255), the beginners signal the begin of a codeword. (s,c)-Dense Code (SCDC) [29] is similar to Vbyte, it also has two types of chunks, but instead of beginners, it uses *stoppers*. A codeword is formed by one stopper, and zero, one or more continuers. The stopper chunk signals the end of a codeword, and thus, that the next chunk corresponds to the next codeword. However, instead of using 128 values for each set, SCDC decides what is the best distribution of the byte values between stoppers and continuers for a given input sequence, in order to obtain the best compression.

Codes based on chunks are faster, but they pay a price in space. A different family of codes tries to join the good space consumption of the bit-based codes and the fast encoding and decoding of the byte-aligned codes. Instead of encoding and decoding each source symbol, these codes treat short sequences of numbers and read whole computer words from the input.

PforDelta [30] encodes a fixed number of integers at a time (typically 128), using for all of them the number of bits needed for the largest one. A fraction of the largest numbers (usually 10%) is encoded separately, and the other 90% is used to calculate how many bits are needed per number.

The codes shown so far do not provide direct access to positions, that is, we cannot directly access the codeword representing the i^{th} symbol in the original sequence without decompressing from the beginning, because they use variable length codewords. The classical solution to this problem is to use absolute pointers to sampled elements, that is, to each h^{th}-element of the sequence. These pointers obviously suppose an overhead. However, there are techniques that avoid the use of pointers. These techniques use a conventional code along with an additional structure to allow direct access. Examples can be based on Elias-Fano codes [31, 32], on Interpolative coding [33], or on Vbyte [34]. The latter approach is denoted as Directly Addressable Codes (DACs). Instead of storing all the chunks of each codeword before the chunks of the next codeword, DACs store the last chunk (the least significant) of all codewords sequentially. For those codewords that have more than one chunk, that is, those with the mark bit set to one, the second least significant chunks are stored separately and so on. DACs can use different chunk sizes for first, second,... chunks in order to adjust them to obtain the best possible compression. However, decoding DACs has to pay the price of recovering the different chunks of a codeword from the different independent sequences of chunks.

Various techniques make use of the wavelet tree [35], which is a structure that can store the variable length codewords of a sequence, allowing to retrieve the i^{th} codeword without decompressing the previous codewords. It has been used with Huffman [35], Vbyte [36], Elias and Rice codes [37], or Fibonacci codes [38].

3.2. *Compressing Floating Point Numbers*

General-purpose compressors may not perform well over floating point data, thus compression methods specifically designed to compress those numbers were developed following two main strategies. The first one is based on allowing some loss of precision [9, 10, 16]. The second one uses a *predictor* that, before compressing a symbol, obtains a prediction of its value based on previous values, and then stores the difference between the prediction and the actual value [13]. Space saving is obtained because such a difference is a smaller value than the original one and, in addition, it is usually compressed with some sort of encoder. Following this strategy, the well-known ALS compression method [11] of MPEG-4 combines two predictors and Golomb-Rice or Block Gilbert Moore coding. Lindstrom and Isenburg [12] presented a lossless compression method that uses a predictor, called Lorenzo, and encodes the difference with a two level compression scheme. The method proposed by [39] selects the best predictor from an available set, based on the values immediately compressed before the current value. In [15], a regression line computed from the last compressed values is used to predict the next value. The works in [14, 40, 41] use a forecast system based on jump address predictors for CPUs.

In [42], it is presented several alternatives for compressing and indexing sequences of floating point numbers. This work uses data structures designed to index and compress text, adapted to be used on the most significant part of the numbers, whereas the remainder part of the number is stored in plain form.

Concerning Brownian motion values and compression, [43] is the only related work. In this paper, compression methods were used, but the target was to predict future values of stock shares, and not space saving.

4. CBM

There are good compression methods designed for floating point numbers, but they are not valid for in-memory processing since they require to decompress from the beginning, and in some cases, they obtain slow compression and/or decompression times.

CBM is based on a very simple compression method, the differencing encoding. However, this technique cannot be directly applied. Observe that if we subtract two 32-bits floating point numbers, we obtain a new floating point number and therefore we still need 32 bits to represent it. To avoid this problem, we translate the floating point numbers into integers. As we will see later, we only deal with positive numbers, then we simply cast the floating point numbers to the integer that has the exact same 32-bit binary representation.

Next, we have to reduce the size of those 32-bit numbers. Using the typical prediction strategy to compress floating point numbers is a challenge since Brownian motion values are used precisely to model the randomness. Nevertheless, the processed values have an interesting characteristic, they come from a Brownian motion, and thus two consecutive values cannot differ too much. This fits quite nicely with the differencing encoding method, however, we do not use it directly. The differences of CBM are not with respect to the previous number, they are the difference with respect to the previous number plus the average of the differences between each pair of consecutive numbers of its trajectory. In other words, we use a prediction strategy, where the prediction computed to encode a given number is the previous number plus the average of the differences between each pair of consecutive numbers of the considered trajectory. That is, for each trajectory X_i, we compute the average of the differences between consecutive values at times $t - \delta$ and t for all $t \in T$ ($\textit{avgDiff}(X_i)$). Then, to compress the value of that trajectory at time $t \in T$, denoted by $X_i(t)$, we make a prediction $P_i(t)$ from the previous value $X_i(t-\delta)$ as: $P_i(t) = X_i(t-\delta) + \textit{avgDiff}(X_i)$, so we encode $X_i(t)$ as the prediction minus the actual value to encode ($P_i(t) - X_i(t)$).

CBM considers all numbers in absolute value since the difference between two numbers of different sign yields larger differences. Observe, for example, that the difference between 1 and -2 is 3, whereas the difference between 1 and 2 is only 1.

Therefore, the first step of CBM translates the original numbers into positive values. For this, we have to options:

- To use a bitmap (*bitMapS*) to mark the positions of negative numbers.
- To use ZigZag encoding.

Figure 2 shows an example using bitmaps.

DOI: 10.1002/spe

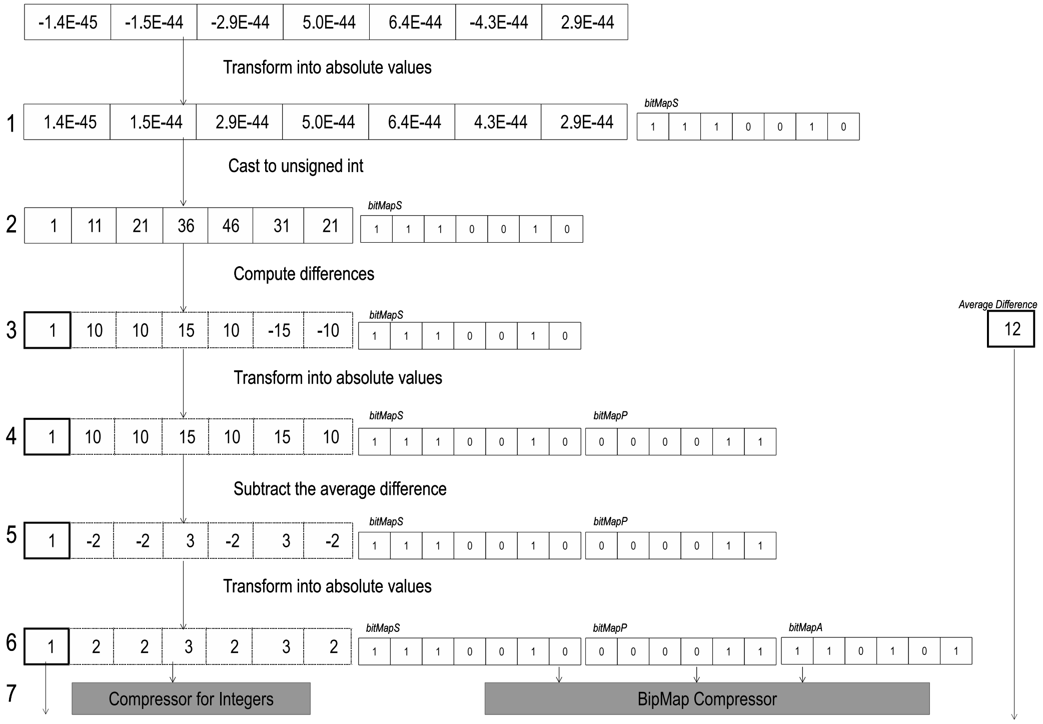


Figure 2. Compression process of a trajectory.

The second step takes the 32-bit number representing each floating point number and casts it into an unsigned integer without any change in the bit number. Step 3 computes the differences between consecutive numbers and obtains the average difference of the trajectory, which in our example is 12. Obviously, observe that the first value of the sequence must be kept in plain.

Step 4 transforms the differences into positive values again. Therefore, we can use either a bitmap (*bitMapP*) or a ZigZag encoding.

Step 5 shows the result of subtracting the average difference from each difference. Converting the resulting values to absolute values requires the addition of another bitmap (shown in step 6), or the use of a ZigZag encoding again. Finally, the sequence of differences is compacted with a compressor for integers, the bitmaps (if they exist) are compacted with a bitmap compressor [44], and the first value of each trajectory and the average difference is stored in plain.

Algorithm 1 Compression.

```
function COMPRESSION($t$ trajectories, $\#Points$)
    for each trajectory $t_i$ do
        $First[t_i] = t_i[1]$                                ▷ Store first value of each trajectory
        for $p \leftarrow 2, \#Points$ do
            if $t_i[p] < 0$ then $bitMapS[t_i, p-1] \leftarrow 1$       ▷ Mark the positions of negative values
            $diffs[t_i, p-1] \leftarrow |t_i[p]| - |t_i[p-1]|$                  ▷ Compute differences
            if $diffs[t_i, p-1] < 0$ then
                $bitMapP[t_i, p-1] \leftarrow 1$                  ▷ Mark the positions of negative values
                $diffs[t_i, p-1] \leftarrow |diffs[t_i, p-1]|$                   ▷ Change of sign
        $avgDiff \leftarrow avg(diffs[t_i])$      ▷ Compute the average difference of the current trajectory
        for $p \leftarrow 1, \#Points - 1$ do
            $diffs[t_i, p] \leftarrow diffs[t_i, p] - avgDiff$                ▷ Subtract the average difference
            if $diffs[t_i, p] < 0$ then
                $bitMapA[t_i, p] \leftarrow 1$                    ▷ Mark the positions of negative values
                $diffs[t_i, p] \leftarrow |diffs[t_i, p]|$                       ▷ Change of sign
    Compression of bitMapS, bitMapP, and bitMapA
    Compression of diffs with a compressor for integers.
```

Algorithm 1 shows the pseudocode of the compression algorithm, where the bitmaps can be avoided by using ZigZag encoding. Since the compression method is based on differences, it is obvious that the decompression must start at a position with a number in plain form. In our method, we only store in plain form the first number of each trajectory, but we could store numbers in plain form at regular intervals, in order to be able to start the decompression at those points. This feature allows us to decompress portions of the input data set. In our case, we can decompress trajectories individually. This allows saving main memory during the computation of any algorithm over the compressed sequence since we can decompress only the trajectories needed at a given step of the algorithm. For this, it is important to have a fast decompression algorithm, otherwise, the computation times could be harmed.

Algorithm 2 shows the decompression algorithm. This process starts by taking the first number of a trajectory in plain. In line 5, we obtain the average difference of that trajectory, which was stored in plain for each trajectory during the compression procedure. Next, for each number in the compressed file, we perform the reverse process of that shown in the compression procedure. For decompressing a given number, we read the difference corresponding to that number and we add (or subtract) the average difference of the trajectory. Then that value is added (or subtracted) to the previous number. Finally, if the number was originally a negative value, then the sign is changed.

Algorithm 2 Decompression

```
 1: function DECOMPRESSION($\#trajectories$, $\#Points$)
 2:     for $t \leftarrow 1, \#trajectories$ do
 3:         $Values[t][1] \leftarrow First[t]$                     ▷ Get first value of each trajectory
 4:         $lastProcNumber \leftarrow First[t]$
 5:         $avgDiff \leftarrow AvgDiffs[t]$                       ▷ Get the average difference of the current trajectory
 6:         for $p \leftarrow 2, \#Points$ do
 7:             $number \leftarrow diffs[t, p-1]$                  ▷ Get the difference between points
 8:             if $bitMapA[t][p-1]$ then                         ▷ Check if is added or subtracted to average difference
 9:                 $number \leftarrow avgDiff - number$
10:             else
11:                 $number \leftarrow avgDiff + number$
12:             if $bitMapP[t][p-1]$ then                         ▷ Check if is subtracted or added to the previous number
13:                 $number \leftarrow lastProcNumber + number$
14:             else
15:                 $number \leftarrow lastProcNumber - number$
16:             $lastProcNumber \leftarrow number$                 ▷ The number is saved to calculate the next
17:             if $bitMapS[t][p-1]$ then                         ▷ Check if real number was negative
18:                 $number \leftarrow -number$
19:             $Values[t][p] \leftarrow number$
20:     return $Values$
```

5. MEMORY EFFICIENT COMPUTATION OF THE SAMPLE AUTOCOVARIANCE FUNCTION

To compute the sample autocovariance function in (1) at two time instants, s and t, the values of all trajectories in these two time instants are needed. In addition, the classical implementation of that equation forces to use each trajectory many times. In this way, the classical algorithms, implemented in R for example, maintain the whole data set in memory. However, the equation can be implemented using each trajectory only once, accumulating in each entry of the sample autocovariance function the contribution to the sum due to the considered trajectory, as shown in Algorithm 3.

However, the algorithm requires the average value of all trajectories at all time instants (avg_t). To obtain those values, for each time instant, we would require loading the value of all trajectories in that time instant. This process should be performed in advance, which yields to worse the running times. Instead, we compute this value during the compression process, and we add it to the compressed sequence, indeed it is required by the decompression procedure. In this way, the autocovariance computing process can read it from the compressed file.

DOI: 10.1002/spe

Algorithm 3 Autocovariance computation trajectory by trajectory

```
function COVARIANCE COMPUTATION($t$ trajectories, $\#Points$)
    for each trajectory $t_i$ do
        for $p \leftarrow 1, \#Points$ do                    ▷ Traverses all the time instants
            for $q \leftarrow p, \#Points$ do                ▷ Traverses the time instants from $p$ to the end
                $cov[p, q] \leftarrow cov[p, q] + ((t_i[p] - avg_p) * (t_i[q] - avg_q))$
                $cov[q, p] \leftarrow cov[p, q]$
    for $p \leftarrow 1, \#Points$ do
        for $q \leftarrow 1, \#Points$ do
            $cov[p, q] \leftarrow cov[p, q]/\#trajectories$
```

CBM uses a compact data structure strategy since it enriches the compact representation to improve the manipulation of the data. It takes advantage of the compression process to perform pre-calculations that will be useful during further processing of the data set.

Note that this algorithm is not linked to CBM or the Brownian motion, as it can be used for trajectories of any continuous time stochastic process.

6. EXPERIMENTAL STUDY

6.1. Data set analysis

Table I shows the details of the data sets used in this study. Several thousands ($n = 10000, 20000, 30000$) of Brownian motion trajectories were simulated in $m = 1000$, 2000, 10000, 15000, 20000, 30000 time instants in $T = [0, 10]$. The Brownian motion considered has zero mean and covariance function $c(s, t) = \min\{s, t\}$. It has been simulated by using independent normally distributed increments for contiguous time instants. The data sets can be downloaded from `http://sarela.dc.fi.udc.es/datasets`.

The first column of Table I shows the number of trajectories (n) and the number of time instants (m), with the form $n \times m$. The second column shows its size in MBs.

In order to give an idea of how hard is to compress those data sets, we use Shannon's information theory [45] to measure the amount of information in those data sets. We used the *zero-order empirical entropy* (in bits/number). Given a sequence $S[1, n]$ over an alphabet $\Sigma = [1 \ldots \sigma]$, the zero-order empirical entropy is $H_0(S) = \sum_{i=1 \ldots \sigma} n_i/n \log(n/n_i)$, where n_i is the number of occurrences of the i^{th} symbol of Σ in S.

To compute the entropy, we regarded the source file as a sequence of 1-byte, 2-byte, and 4-byte integers. The latter case considers the original numbers, but regarded as integers. For each case, we provide the empirical entropy in bits per number and the value of $\log(|\Sigma|)$, where $|\Sigma|$ denotes the size of the alphabet, that is, the list of distinct values found in the data set. $\lfloor \log(|\Sigma|) \rfloor + 1$ gives the minimum number of bits required to represent each number using binary codes of the same length, which is adequate for uniform distributions.

When the original data set is processed considering integers of 1 byte, the empirical entropy is around 7.38 bits per number, whereas $\log(|\Sigma|)$ is exactly 8, since in all datasets, the 256 possible 1-byte values are present. When the data set is regarded as a sequence of 2-byte integers, the entropy is around 14.22 bits per number and $\log(|\Sigma|)$ is 16. Finally, in the case of 4 byte integers the values of the empirical entropy are between 22.75-26.50 bits per integer. In this case, $\log(|\Sigma|)$ is not 32, since not all possible 32-bit values are present in the datasets.

Observe that in the case of 1-byte integers and 4-byte integers, the empirical entropy is very close to $\log(|\Sigma|)$, whereas in the case of 2-byte integers, there is a little gap. This is probably due to the nature of Brownian trajectories and the internal format of floating point numbers. In a Brownian trajectory, the differences between close numbers are small, thus the most significant 16-bits of the 32-bit floating point numbers will vary less since that part includes the sign (1 bit), the exponent (8 bits) and the most significant part of the mantissa (7 bits). Therefore, the first 16-bit numbers will have a more skewed distribution and this is precisely where CBM obtains compression, like all compressors designed for floating point data.

Anyhow, it seems that the chances to compress are low, since the simple binary representation of the numbers is close to the amount of information they carry, that is, the original sequence has an almost flat distribution and a large amount of distinct numbers, and this is precisely the worst scenario to achieve compression. This is not surprising, since values simulated from a Brownian motion are inherently random.

		1-byte integers		2-byte integers		4-byte integers	
Data set size	Size (MBs)	Entropy (bits/ 1b-int)	$\log(\|\Sigma\|)$	Entropy (bits/ 2b-int)	$\log(\|\Sigma\|)$	Entropy (bits/ 4b-int)	$\log(\|\Sigma\|)$
10000x1000	38.15	7.38	8	14.22	16	23.14	23.17
20000x2000	152.59	7.39	8	14.22	16	24.83	24.94
10000x10000	381.47	7.39	8	14.21	16	25.66	25.86
10000x15000	572.20	7.38	8	14.22	16	25.93	26.18
20000x20000	1525.88	7.38	8	14.22	16	26.34	26.71
30000x30000	3433.23	7.39	8	14.22	16	26.50	27.00

Table I. Data set sizes and entropy.

	1-byte integers		2-byte integers		4-byte integers	
Data set size	Entropy (bits/ 1b-int)	$\log(\|\Sigma\|)$	Entropy (bits/ 2b-int)	$\log(\|\Sigma\|)$	Entropy (bits/ 4b-int)	$\log(\|\Sigma\|)$
10000x1000	6.60	8	12.07	16	21.53	21.86
20000x2000	6.54	8	11.89	16	21.56	22.39
10000x10000	6.37	8	11.44	16	20.79	22.47
10000x15000	6.33	8	11.32	16	20.59	22.65
20000x20000	6.29	8	11.24	16	20.46	23.30
30000x30000	6.24	8	11.13	16	20.25	23.75

Table II. Entropy of the files of differences.

Recall that our strategy can be divided in two main steps: first, we preprocess the original data set in order to obtain a sequence of differences and second, we compress such sequence using compressor for integers. The sequence of differences (shown in the Step 6 in Figure 2) is analyzed in Table II. We can see a decrease in the entropy, and thus the integer compressor will be more successful in this new preprocessed file. As an example, if we apply the SCDC compressor over the original sequence, the output is even bigger than the original file, around 125%, whereas if we apply it over the sequence of differences, the compression ratio is around 85% in large files, including the auxiliary bitmaps. These auxiliary bitmaps are the price that CBM has to pay, in part, for that decrease of entropy, which in the case of the SCDC version represents $11 - 11.5\%$ of the compressed data set, and between $11.5 - 13.34\%$ in the case of the DAC version.

Another interesting effect of the preprocessing is that, when the original dataset is considered as a sequence of 4-byte integers, the entropy grows as the size of data set increases. However, in the sequence of differences, the entropy decreases as the size of data set increases, and this will be reflected in the compression ratio, as it will be shown below.

6.2. Setup

Our test machine has an Intel(R) Core(tm) i7-3820@3.60GHz CPU (4 cores/8 siblings) and 64GB of DDR3 RAM. It runs Ubuntu Linux 12.04 (Kernel 3.2.0- 121-generic). The hard disk was a Seagate ST3000DM001.

As compressor for integers, we used the following techniques: DAC, SCDC, PforDelta, and Kulekci [37] using Rice and Elias encoding. We tested other compressors for integers, but they yield worse values.

By default, we used the three bitmaps: $bitMapA$, $bitMapP$, and $bitMapS$. For DAC and Kulekci with Rice encoding, we also provide the values substituting $bitMapP$ and $bitMapS$ by ZigZag encoding. We tested this approach for the rest of techniques, but the results were worse than the full bitmap versions or they did not run.

We only substituted the $bitMapP$ and $bitMapS$, since if we use ZigZag encoding in the step corresponding to the $bitMapA$, the numbers will have a larger magnitude, and thus, since compressors for integers are designed to compress small integers, most of them did not work, or if they run, the results were poor.

6.3. *Compression performance*

Table III shows the compression ratio, that is, the size of the compressed data set as a percentage of its original size. We compare CBM, against: Gnu *gzip*[†], a Ziv-Lempel-based compressor, the *p7zip*[‡] compressor, which is a LZMA compressor with a dictionary of up to 4 Gigabytes, and *fpzip*[§], a compressor specially designed to compress floating point numbers. In the case of gzip, we used the default level of compression.

fpzip obtains the best results, between 15% and 21% better than CBM, except in the largest data set, where fpzip did not run. The next level is p7zip, yet CBM is very close, in the largest data set, the Kulekci-Elias version is on a par, and the PforDelta version is at most 8% worse. However, neither fpzip nor p7zip allow partial decompression, essential to directly use the compressed data in main memory.

With respect to the CBM versions, the PforDelta and DAC-ZZ are the most homogeneous, although in the largest data set the best one is the Kulekci-Elias version. The SCDC and Kulekci-Rice versions are in general worse.

				CBM						
				DAC			Kulekci			Pfor
Data set size	gzip	p7zip	fpzip	bitmap	ZZ	SCDC	Rice-b	Rice-ZZ	Elias	Delta
10000 × 1000	92.61	80.81	72.42	85.13	84.73	89.78	95.85	99.13	132.88	83.20
20000 × 2000	92.30	79.05	70.82	81.41	81.05	87.58	92.15	91.22	105.33	80.84
10000 × 10000	90.35	75.01	66.85	76.23	75.91	85.63	89.16	84.08	79.20	76.00
10000 × 15000	89.66	73.98	65.79	75.32	75.00	85.43	88.88	83.39	76.30	75.16
20000 × 20000	89.15	73.21	64.96	74.61	74.29	85.26	88.76	83.03	74.59	74.59
30000 × 30000	88.47	72.12	n/a	73.76	73.44	85.07	88.61	82.66	72.67	73.93

Table III. Compression ratio.

Table IV shows the performance in compression time. Again the best method is fpzip, except in the largest data set, where it did not run, and then gzip is the fastest. CBM pays the price of performing a compression process per trajectory. If CBM would compress the data of all trajectories in a unique run of the integer compressor, and thus producing a unique compressed data set, it would be even faster than fpzip. However, in order to be able to load into memory the data of only one compressed trajectory, the trajectories have to be compressed isolatedly.

For our purpose of using compressed data in main memory, the key feature is the decompression performance, a slow decompression process will harm the processing. Table V shows the decompression times. CBM-PforDelta is the fastest technique, between 2.2 and 3.9 times faster than gzip, which compresses between 9 and 15 percentage points less. fpzip, which obtains the best compression, is around 5 times slower than CBM PforDelta.

[†] `http://www.gzip.org/`
[‡] `http://p7zip.sourceforge.net/`
[§] `http://computation.llnl.gov/projects/floating-point-compression-zfp-fpzip`

DOI: 10.1002/spe

				CBM						
				DAC			Kulekci			Pfor
Data set size	gzip	p7zip	fpzip	bitmap	ZZ	SCDC	Rice-b	Rice-ZZ	Elias	Delta
10000 × 1000	1.59	6.28	0.74	1208.92	4848.92	14.21	5.30	5.31	6.30	11.46
20000 × 2000	6.31	25.81	3.29	2721.44	10514.35	39.04	14.84	14.93	18.75	44.46
10000 × 10000	17.17	66.07	8.39	1578.49	6191.75	73.53	24.42	24.84	33.46	103.31
10000 × 15000	26.13	98.36	12.83	1596.06	6232.87	102.14	34.99	34.83	47.69	152.22
20000 × 20000	68.38	365.18	37.71	3246.89	13229.90	256.13	91.02	91.31	123.80	401.15
30000 × 30000	155.19	915.98	n/a	4968.80	21806.09	518.64	198.20	197.19	271.69	889.26

Table IV. Compression time (secs.).

				CBM						
				DAC			Kulekci			Pfor
Data set size	gzip	p7zip	fpzip	bitmap	ZZ	SCDC	Rice-b	Rice-ZZ	Elias	Delta
10000 × 1000	0.45	2.02	0.72	0.26	0.24	0.19	0.94	1.59	8.42	0.14
20000 × 2000	2.14	7.97	2.80	1.08	0.89	0.69	4.24	6.14	32.39	0.60
10000 × 10000	5.19	19.36	6.89	2.52	2.07	1.61	8.20	10.49	75.07	1.45
10000 × 15000	7.78	28.74	9.98	3.74	3.14	2.40	10.68	13.62	109.90	2,16
20000 × 20000	13.21	81.34	28.13	10.69	8.14	6.42	28.18	34,81	289.06	5.98
30000 × 30000	50.36	187.55	n/a	22.06	18.24	14.68	59.55	71.64	633.63	13.01

Table V. Decompression time (secs.).

6.4. Memory consumption during the computation of the sample autocovariance function

Table VI shows the maximum virtual memory[¶] consumption during the computation of the sample autocovariance function using the classical algorithm (see Section 2.2). With this approach, the whole input data set is stored in main memory.

We compare the R implementation, using plain data (binary representation of numbers), our own C implementation, and our implementations using CBM compressed data as input.

We also performed another experiment, which supposes that the data set is stored on disk compressed with a classical compressor. Therefore a previous full decompression is needed in order to obtain the uncompressed version, which is then processed with the normal C program. In this case, we give the highest value of memory consumption between the decompression process and the computation of the sample autocovariance function. These values correspond to the columns labeled as “C+gzip”, “C+p7zip”, and “C+fpzip”.

The C and CBM implementations are basically the same C program. Both programs maintain the whole data set in main memory, but the CBM version keeps the data set in compressed form and only decompresses an individual trajectory when the algorithm requires those data in a given step.

The C implementation is on a par with CBM, there are two reasons for this. First, in both cases the output is kept in main memory uncompressed, and this is the biggest component of the memory consumption. Observe in Table VI that when processing the data set of size 30000 × 30000, both alternatives consume around 10 GBs. The input data set requires around 3.3 GBs uncompressed and 2.5 GBs compressed. The output is stored in doubles (in order to be fair with R), and then it occupies 6.6 GBs, the biggest part. The second factor is that the advantage of the CBM in the input size (0.8 GBs) is compensated by the fact that, at a given step of the algorithm, the CBM version has to decompress a treated trajectory, and therefore, that trajectory is stored twice in main memory. In addition, some auxiliary data structures are needed to perform the decompression.

Considering the experiment where the data are compressed with a classical compressor, the decompression process of gzip and p7zip has a smaller memory footprint than the computation of the sample autocovariance function. However, fpzip, which is the best compressor in disk space,

[¶] This includes the complete space of addresses of the process.

consumes a large amount of memory, between 76% and 3.7 times more than CBM, and indeed it did not run with the largest file.

Finally, the R implementation is the worst one. It consumes between 87% and 3.71 times more memory than CBM.

The memory consumption of the memory efficient version, which uses the Algorithm 3, is shown in Table VII. Except in the small files, where the C implementation consumes less space than CBM, in the larger data sets, they are on a par again.

In the case of "C+gzip" and "C+p7zip", even using the memory efficient algorithm to compute the sample autocovariance function, the decompression process consumes less main memory, except with p7zip in small files. Obviously, now the gap between "C+fpzip" and CBM is even bigger: CBM consumes between 87% and 5% of "C+fpzip".

						CBM						
			C +			DAC			Kulekci			Pfor
Data set size	R	C	gzip	p7zip	fpzip	bitmap	ZZ	SCDC	Rice-b	Rice-ZZ	Elias	Delta
10000 × 1000	223	50	50	50	254	60	60	60	60	60	60	60
20000 × 2000	648	187	187	187	982	197	197	197	197	197	197	197
10000 × 10000	2545	1149	1149	1149	3204	1159	1159	1159	1159	1159	1159	1158
10000 × 15000	4060	2293	2293	2293	3777	2303	2303	2303	2303	2303	2303	2303
20000 × 20000	9096	4582	4582	4582	12782	4592	4592	4592	4592	4592	4592	4592
30000 × 30000	19358	10305	10305	10305	n/a	10315	10314	10315	10315	10315	10314	10315

Table VI. Memory consumption (in MBs) of the classical algorithm.

					CBM						
			C +		DAC			Kulekci			Pfor
Data set size	C	gzip	p7zip	fpzip	bitmap	ZZ	SCDC	Rice-b	Rice-ZZ	Elias	Delta
10000 × 1000	12	12	37	254	25	23	25	25	23	25	25
20000 × 2000	35	35	37	982	59	49	60	59	49	59	59
10000 × 10000	767	767	767	3204	814	789	814	814	789	814	814
10000 × 15000	1721	1721	1721	3777	1787	1749	1787	1787	1749	1787	1787
20000 × 20000	3130	3130	3130	12782	3216	3116	3216	3216	3116	3216	3216
30000 × 30000	6919	6919	6919	n/a	7219	6994	7219	7219	6994	7219	7219

Table VII. Memory consumption (in MBs) of the memory efficient algorithm.

6.5. Time to Compute the sample autocovariance function

Table VIII shows the time required to compute the sample autocovariance function. In the experiments "C+gzip", "C+p7zip", and "C+fpzip", the values are resulting from adding the decompression time and the time of the computation of the sample autocovariance function using the C program over the uncompressed data.

CBM did not improve the values of the C implementation in memory consumption, however in this experiment, CBM clearly beats the C implementation. There are two reasons for this. First, the computation of the covariance function requires the average value of all trajectories at all time instants (see Section 2.2). In the case of CBM, those values are computed during compression and stored in the compressed file. However, the C program has to calculate those values before running the main loop implementing Equation (1). The second factor is the memory hierarchy. While the output has a big impact in the memory footprint, that space is not critical for the running times. However, the input data is read repeatedly, and making the input data available to the processor as quickly as possible has a big impact in the running times. A shorter input has more chances of being stored in higher levels of the memory hierarchy.

With the largest dataset, the C implementation of the classical algorithm consumes up to 10 GBs of virtual memory, while the maximum resident memory[‖], is 6.8 GBs. Therefore the 3.2 GBs of difference must be stored on disk in the swap area, moreover, that data are interchanged between disk and memory during the computation, implying an important slowdown. Even in the smallest data set, the virtual memory peak was 50 MBs, while the resident memory peak was 44 MBs. These values are similar for the CBM version, but the number of interchanges between memory and disk could have an important impact. The same applies for the interchanges between memory and the different levels of processor cache, where the number of reads that are successfully solved in low level caches has a big impact.

CBM is between 50% and 66% faster than the C implementation and between 13% and 40% faster than R. Observe that, R is faster than the C program. The reason is probably again the memory hierarchy: R uses almost twice as much main memory space as the C program.

If we consider that a previous decompression is needed before running the C program, the improvements are obviously better, CBM is between 67% and 2 times faster.

This experiment shows that the decompression required by CBM is really fast and therefore the computation time is not harmed.

						CBM						
			C +			DAC			Kulekci			Pfor
Data set size	R	C	gzip	p7zip	fpzip	bitmap	ZZ	SCDC	Rice-b	Rice-ZZ	Elias	Delta
10000 × 1000	5	6	7	8	7	4	4	4	5	5	12	4
20000 × 2000	42	50	52	58	53	31	31	30	34	36	63	30
10000 × 10000	515	612	618	632	619	372	375	372	379	383	447	371
10000 × 15000	1155	1374	1382	1403	1384	834	845	834	844	854	942	834
20000 × 20000	4134	4863	4876	4944	4891	2945	2965	2948	2982	2994	3240	2947
30000 × 30000	14006	16488	16538	16676	n/a	9952	10233	9945	9984	10022	10847	9938

Table VIII. Computation time (secs.) for the sample autocovariance function with the classical algorithm.

Table IX shows the times when using the memory efficient version of the algorithm. Now, the gap between CBM and the C implementation is shorter, they are almost on a par, and only in the largest files, CBM is around 3.5% faster. In a given step of the algorithm, only one trajectory is loaded into memory, therefore, the C implementation has more chances to fit that trajectory in the upper levels of the memory hierarchy.

Again, resident memory usage supports this explanation. In the largest data set, the peak consumption of virtual memory was 6,919 MBs, while the maximum resident memory consumption was 6,915 MBs. This implies that interchanges between the swap area and memory are almost inexistent, and then the memory efficient algorithm using CBM is even faster than the C implementation of the classical algorithm, which intuition says that should be faster since it has the complete data set uncompressed in main memory.

If we consider the experiment where the data were compressed with a classical compressor, then CBM is up to 13 % faster.

In Figure 3, we show the trade-off between memory consumption and computation time using the largest data set with the two algorithms to compute the sample autocovariance function (classical and memory efficient ones). We only provide the values of some of the versions of CBM to avoid cluttering the figure. In addition, the values of "C+fpzip" are not present since fpzip did not run in this data set.

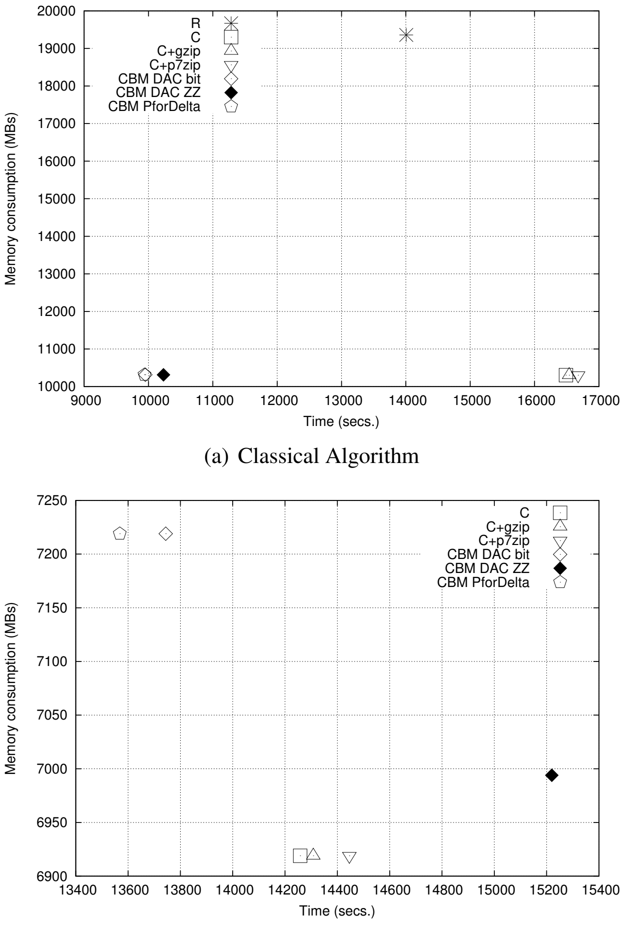
With the classical algorithm, we can see that clearly CBM has the best balance, as it has almost the same memory consumption as the C implementation, but better running times. Observe that R has by far a worse memory consumption.

[‖] The space of RAM used by the process.

DOI: 10.1002/spe

					CBM						
		C +			DAC			Kulekci			Pfor
Data set size	C	gzip	p7zip	fpzip	bitmap	ZZ	SCDC	Rice-b	Rice-ZZ	Elias	Delta
10000 × 1000	3	4	5	4	6	4	4	5	5	12	4
20000 × 2000	42	45	50	45	44	44	45	47	49	75	43
10000 × 10000	487	493	507	494	490	520	507	519	518	583	488
10000 × 15000	1131	1139	1160	1141	1109	1139	1141	1158	1155	1267	1105
20000 × 20000	4088	4101	4169	4116	4015	4028	4085	4092	3982	4373	3936
30000 × 30000	14258	14308	14446	n/a	13744	15220	13838	14301	13587	15853	13568

Table IX. Computation time (secs.) for the sample autocovariance function with the memory efficient algorithm.



(a) Classical Algorithm

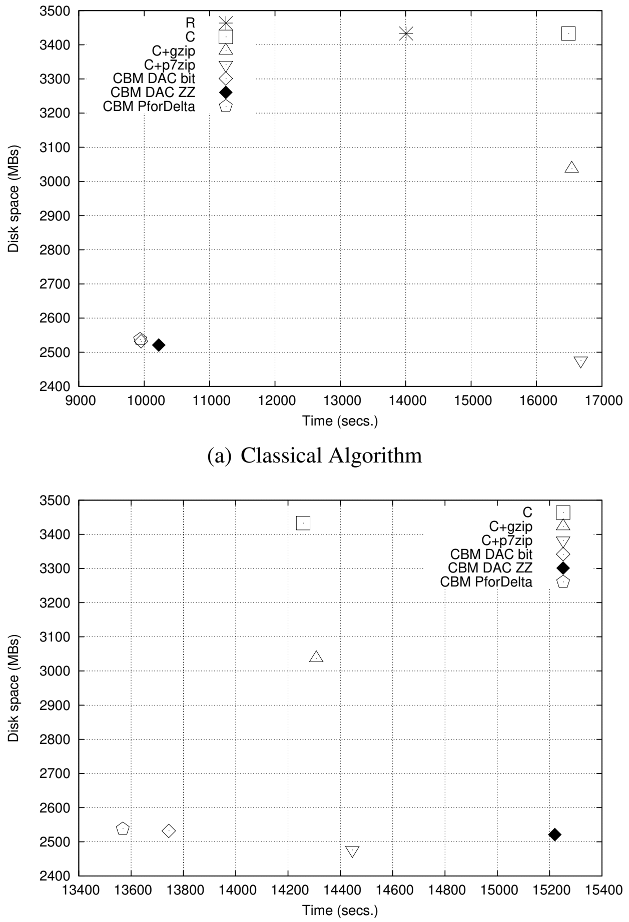
(b) Memory Efficient Algorithm

Figure 3. Trade off memory consumption/computation time with the dataset 30000 × 30000.

In the memory efficient version plot, R is not present. The differences are shorter, having the C implementation a slight advantage in memory consumption, and the CBM-DAC-bit and CBM-PforDelta versions a slight advantage in time.

Figure 4 shows the trade-off between disk space and the time needed to compute the sample autocovariance function. With the classical algorithm, CBM is again the best alternative by far in both disk space and time. “C+p7zip” occupies a bit less space in disk, but requires much more time to compute the sample autocovariance function. When using the memory efficient algorithm, CBM PforDelta shows the best balance.

DOI: 10.1002/spe



(a) Classical Algorithm

(b) Memory Efficient Algorithm

Figure 4. Trade-off disk space/computation time with the data set of size 30000 × 30000.

Figure 5 shows the final picture, by comparing R and the classical C implementation against our proposal: the use of CBM and a new memory efficient algorithm to compute the sample autocovariance function. With respect to memory consumption, the efficient versions has the lowest memory consumption. The memory efficient C implementation consumes 2.8 times less memory than R. In the case of the C program and CBM, the use of the memory efficient algorithm implies a reduction around 48% in the memory footprint with respect to the classical implementations.

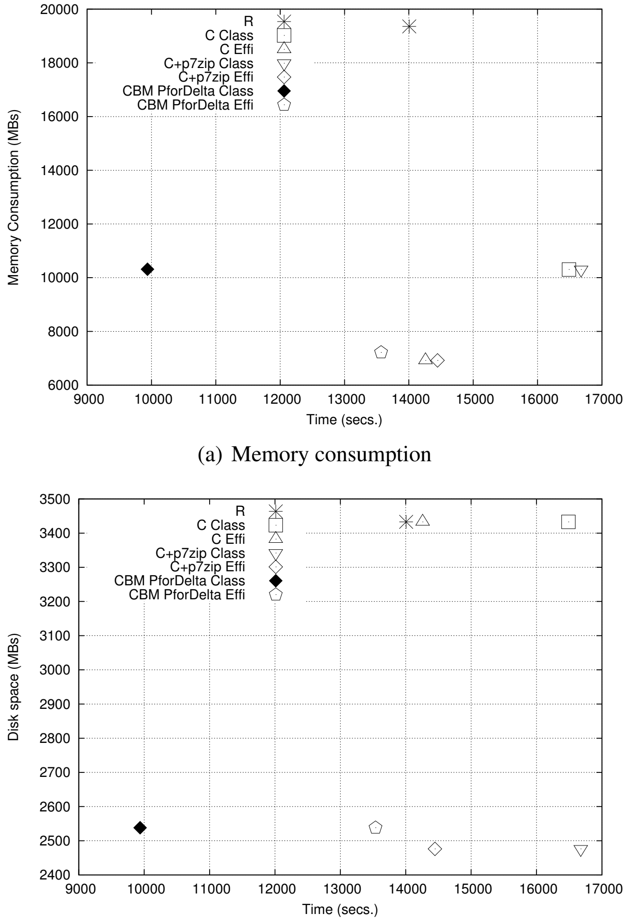
In disk space, the best option is p7zip, but its fastest version (that uses the memory efficient algorithm) is 45% slower than CBM-PforDelta with the classical algorithm, which is the version that shows a best balance in this plot.

Finally, the classical implementation of CBM-PforDelta is 40% faster than R and 65% than the classical C implementation.

7. CONCLUSIONS AND FUTURE WORK

In this work, we have shown that parallel processing is not the only way to improve the scalability of a large-scale analysis system. It is an easy way since there many hardware and software tools available. We still have the opportunity to squeeze more data structures and algorithms, something common in the early times of computer science, where the hardware and software resources were much more limited. We apply a compact data structures strategy to improve the scalability of the

DOI: 10.1002/spe



(a) Memory consumption

(b) Disk space

Figure 5. Overall the data set of size 30000 × 30000.

analysis of Brownian motion trajectories, although our solutions can be applied to any continuous time stochastic process. However, our techniques do not prohibit the use of parallelization, on the contrary, they are even more suitable, since each node can process one trajectory independently from the others and data interchanges consume less bandwidth.

CBM is a strategy to represent Brownian trajectories in a compressed way using around 75% of the original space. The novelty is that this saving is also applicable to main memory space, since we can keep the data set compressed all the time, decoding an isolated trajectory when needed and keeping the rest of the representation in a compressed form. Moreover, one isolated trajectory can be extracted from the compressed file in disk, loaded into main memory, and decompressed. In this way, the empirical autocovariance function is computed using up to 13 times less main memory space than when using the traditional method on plain data.

The new approach does not only save space in disk and main memory, but it also obtains reductions in running times. We use two strategies for this. First, the average value of all trajectories in all time instants, which is needed in the computation of the autocovariance function, is computed during the compression and stored in the compressed file. Second, the savings in main memory allow a better usage of the memory hierarchy. Therefore, our approach is up to 65% faster than running a classical C program. Moreover, we have shown that our memory efficient version of the algorithm is even faster than the classical setup, that is, storing the complete input data set in main memory

In addition, thanks to the possibility of decompressing parts of CBM compressed data, we can apply the memory efficient algorithm without a previous decompression. If we have to decompress the input dataset before the application of a C program, that process is up to 67% slower than computing the sample autocovariance function using directly CBM compressed data.

As future work, we plan to improve the compression ratio of the method, to test the method with other continuous time stochastic processes (other than the Brownian motion) and to include other interesting statistics to be computed from the sample of trajectories.

ACKNOWLEDGEMENTS

This work was supported by Ministerio de Economía y Competitividad (PGE and FEDER) under Grants [TIN2016-78011-C4-1-R; MTM2014-52876-R; TIN2013-46238-C4-3-R], Centro para el desarrollo Tecnológico e Industrial MINECO [IDI-20141259; ITC-20151247;ITC-20151305;ITC-20161074]; Xunta de Galicia (co-founded with FEDER) under Grupos de Referencia Competitiva Grants ED431C-2016-015; Xunta de Galicia-Consellería de Cultura, Educación e Ordenación Universitaria (co-founded with FEDER) under Redes Grants R2014/041, ED341D R2016/045; Xunta de Galicia-Consellería de Cultura, Educación e Ordenación Universitaria (co-founded with FEDER) under Centro Singular de Investigación de Galicia Grant ED431G/01.

REFERENCES

1. Kane M, Emerson J, Weston S. Scalable strategies for computing with massive data. *Journal of Statistical Software* 2013; **55**(1):1–19.
2. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Communications of the ACM* Jan 2008; **51**(1):107–113.
3. Dudoladov S, Xu C, Schelter S, Katsifodimos A, Ewen S, Tzoumas K, Markl V. Optimistic recovery for iterative dataflows in action. *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, 2015; 1439–1443.
4. Schelter S, Ewen S, Tzoumas K, Markl V. "all roads lead to rome": Optimistic recovery for distributed iterative data processing. *Proceedings of the ACM International Conference on Information & Knowledge Management (CIKM '13)*, 2013; 1919–1928.
5. DeWitt DJ, Katz RH, Olken F, Shapiro LD, Stonebraker MR, Wood DA. Implementation techniques for main memory database systems. *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*, 1984; 1–8.
6. Plattner H. A common database approach for oltp and olap using an in-memory column database. *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*, 2009; 1–2.
7. Plattner H, Zeier A. *In-Memory Data Management: Technology and Applications*. SpringerLink : Bücher, Springer Berlin Heidelberg, 2012.
8. Navarro G. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016.
9. Wallace GK. The JPEG still picture compression standard. *Communications of the ACM* Apr 1991; **34**:31–44.
10. Mitchell JL, Pennebaker WB, Frogg CE, Legall DJ. *MPEG Video Compression Standard*. Chapman and Hall, 1996.
11. Liebchen T, Reznik YA. MPEG-4 ALS: an emerging standard for lossless audio coding. *Proceedings of Data Compression Conference (DCC'04)*, 2004; 439–448.
12. Lindstrom P, Isenburg M. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics* 2006; **12**(5):1245–1250.
13. Engelson V, Fritzson D, Fritzson P. Lossless compression of high-volume numerical data from simulations. *Proceedings of Data Compression Conference (DCC'00)*, 2000; 574.
14. Ratanaworabhan P, Ke J, Burtscher M. Fast lossless compression of scientific floating-point data. *Proceedings of Data Compression Conference (DCC'06)*, 2006; 133–142.
15. You Y, Sung MY. Haptic data transmission based on the prediction and compression. *Proceedings of International Conference on Communications*, 2008; 1824–1828.
16. Muckell J, Hwang J, Patil V, Lawson CT, Ping F, Ravi SS. Squish: An online approach for gps trajectory compression. *Proceedings of International Conference on Computing for Geospatial Research & Applications*, 2011; 13:1–13:8.
17. Einstein A. On the theory of the Brownian movement. *Annalen der Physik* 1906; **19**(4):371–381.
18. Black F, Scholes M. The pricing of options and corporate liabilities. *The Journal of Political Economy* 1973; **81**(3):637–654.
19. Hulll J. *Options, Futures, and other Derivatives*. Pearson Prentice Hall, 2009.
20. Ramsay J, Silverman B. *Functional Data Analysis*. Springer, 2005.
21. Ferraty F, Vieu P. *NonParametric Functional Data Analysis: Theory and Practice*. Springer, 2006.
22. Horvath L, Kokoszka P. *Inference for Functional Data with Applications*. Springer, 2012.

DOI: 10.1002/spe

23. López-Granados F, Peña-Barragán, JM M Jurado-Expósito, Francisco-Fernández M, Cao R, Alonso-Betanzos A, Fontenla-Romero O. Multispectral classification of grass weeds and wheat (triticum durum) crop using linear and nonparametric functional discriminant analysis, and neural networks. *Weed Research* 2008; **48**:28–37.
24. Huffman DA. A method for the construction of minimum-redundancy codes. *Proceedings of the Institue of Radio Engineers Inc.* 1952; **40**(9):1098–1101.
25. Elias P. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory* 1975; **21**(2):194–203.
26. Golomb SW. Run-length encodings. *IEEE Trans. Inform. Theory* 1966; **IT-12**:399–401.
27. Rice RF. Some practical universal noiseless coding techniques. *Technical Report 79-22*, Jet Propulsion Laboratory, Pasadena, CA 1979.
28. Williams, Zobel. Compressing integers for fast file access. *The Computer Journal* 1999; **42**(3):193–201.
29. Brisaboa N, Fariña A, Navarro G, Paramá J. Lightweight natural language text compression. *Information Retrieval* 2007; **10**(1):1–33.
30. Zukowski M, Héman S, Nes N, Boncz PA. Super-scalar RAM-CPU cache compression. *Proceedings of the International Conference on Data Engineering, (ICDE'06)*, IEEE Computer Society, 2006; 59–71.
31. Munro JI. Tables. *Proceedings Conference Foundations of Software Technology and Theoretical Computer Science,*, 1996; 37–42.
32. Okanohara D, Sadakane K. Practical entropy-compressed rank/select dictionary. *Proceedings of the Workshop on Algorithm Engineering and Experiments, (ALENEX'07)*, 2007.
33. Teuhola J. Interpolative coding of integer sequences supporting log-time random access. *Information Processing Management* 2011; **47**(5):742–761.
34. Brisaboa NR, Ladra S, Navarro G. Dacs: Bringing direct access to variable-length codes. *Information Processing and Management* 2013; **49**(1):392–404.
35. Grossi R, Gupta A, Vitter JS. High-order entropy-compressed text indexes. *Proceedings Symposium on Discrete Algorithms (SODA 03)*, 2003; 841–850.
36. Brisaboa NR, Faria A, Ladra S, Navarro G. Reorganizing compressed text. *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'08)*, Singapore, 2008; 139–146.
37. Külekci MO. Enhanced variable-length codes: Improved compression with efficient random access. *Proceedings Data Compression Conference, (DCC'14), Snowbird, UT, USA, 26-28 March, 2014*, 2014; 362–371.
38. Klein ST, Shapira D. Random access to fibonacci encoded files. *Discrete Applied Mathematics* 2016; **212**:115 – 128. Stringology Algorithms.
39. Fout N, Ma K. An adaptive prediction-based approach to lossless compression of floating-point volume data. *IEEE Transactions on Visualization and Computer Graphics* Dec 2012; **18**(12):2295–2304.
40. Burtscher M, Ratanaworabhan P. Fpc: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers* 2009; **58**(1):18–31.
41. Burtscher M, Ratanaworabhan P. gfpc: A self-tuning compression algorithm. *Proceedings of Data Compression Conference (DCC 10)*, 2010; 396–405.
42. Fariña A, Ordóñez A, Paramá JR. Indexing and self-indexing sequences of ieee 754 double precision numbers. *Information Processing Management* 2014; :857–875.
43. Azhar S, Badros G, Glodjo A, Kao M, Reif H. Data compression techniques for stock market prediction. *Proceedings Data Compression Conference (DCC'94)*, 1994.
44. González R, Grabowski S, Mäkinen V, Navarro G. Practical implementation of rank and select queries. *Poster Proceedings Volume of Workshop on Efficient and Experimental Algorithms (WEA'05)*, 2005; 27–38.
45. Shannon CE. A mathematical theory of communication. *Bell System Technical Journal* 1948; **27**:370–423, 623–656.