Scientific Research Publishing

Journal of Applied Mathematics and Physics, 2025, 13(12), 4247-4279
https://www.scirp.org/journal/jamp
ISSN Online: 2327-4379
ISSN Print: 2327-4352

Designing a Parallel Fractal Dimension Estimator

Athanasios I. Margaris

Department of Digital Systems, Gaiopolis Campus, University of Thessaly, Thessaly, Greece
Email: atmargaris@uth.gr

How to cite this paper: Margaris, A.I. (2025) Designing a Parallel Fractal Dimension Estimator. *Journal of Applied Mathematics and Physics*, **13**, 4247-4279.
https://doi.org/10.4236/jamp.2025.1312236

Received: November 3, 2025
Accepted: December 13, 2025
Published: December 16, 2025

Copyright © 2025 by author(s) and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/

Open Access

Abstract

This paper presents a theoretical framework for parallelizing the FD3 algorithm, which estimates the capacity, information, and correlation dimensions of chaotic time series using the box-counting method. We propose a hybrid parallel implementation leveraging MPI as well as OpenMP to achieve scalable performance on modern high-performance computing architectures. The serial FD3 algorithm is analyzed in detail, followed by a comprehensive parallelization strategy for its key phases, including data retrieval, trajectory reconstruction, sorting, and dimension estimation. Theoretical models based on Amdahl's and Gustafson's Laws predict significant speedup, supported by complexity analysis and communication cost models. While experimental validation is planned as future work, this study provides a robust foundation for parallel fractal dimension estimation, with potential applications in chaotic system analysis and computational physics. The motivation for this research is that, although the FD3 algorithm is relatively old and more advanced and efficient algorithms have since been developed, its structure and operation allow for parallelization, which, by providing significant acceleration, could render it competitive once again and suitable for efficient use.

Keywords

Fractal Dimensions, Lyapunov Exponent, Open MP, MPI, Henon Map

1. Introduction

One of the fundamental properties associated with a chaotic attractor, and in general, with a chaotic time series, is the attractor dimension, that allows the estimation of the area or volume occupied by the attractor in the phase space of a chaotic system. As it is well known from the study of such systems, unlike ordinary geometrical objects that are characterized by an integer dimension (for example, a

circle is a two dimensional object, a cube is a three dimensional one, and so on), the dimension of a strange chaotic attractor is a fractional number.

Generally speaking, there are many types of fractional dimensions that can be defined for a chaotic attractor, the most important ones are presented. An effective way to estimate the value of those dimensions, is to fill the d-dimensional phase space containing the attractor, with a number of hypercubes of side size ℓ, so as to cover the entire surface of the attractor. Let us denote by $M(\ell)$ the minimum number of hypercubes that give the minimum covering of the attractor's surface. In this case, three different types of dimensions can be defined (see [1]-[3]) and more specifically:

- *Capacity dimension*. The capacity dimension D, is given by the number of hypercubes of side size ℓ, required to cover the surface of the attractor. This number is defined as

$$M(\ell) = C\left(\frac{1}{\ell}\right)^{d} \tag{1}$$

where d is the embedding dimension of the phase space, and C is an appropriately chosen constant. To estimate the capacity dimension, the above equation has to be solved with respect to d, leading to the expression

$$d = \frac{\ln(M(\ell)) - \ln C}{\ln(1/\ell)} \tag{2}$$

and then, the limit of the above expression for the case $\ell \to 0$, has to be considered. Based on the above description, the capacity dimension is defined as

$$D = \lim_{l \to 0} \frac{\ln(M(l))}{\ln(1/l)} \tag{3}$$

According to the theory presented so far, it is not difficult to understand that the number of hypercubes $M(\ell)$ varies with the hypercube side size ℓ, according to the rule $M(\ell) \sim \ell^{-D}$. It should also be noted, that the capacity dimension is a purely geometric property, and therefore, it is independent of the frequency at which the trajectories of the system pass through each one of the cells defined in the phase space. Therefore, the contribution of two cells to the value of the capacity dimension is the same, regardless the value of the above frequency. As a geometric property, the capacity dimension is quite difficult to compute, a fact that is particularly true for embedding dimensions d of the reconstructed phase space.

- *Information dimension*. In contrast to the capacity dimension whose value is independent of the number of visits of the system trajectory to each cell of the phase space, the information dimension σ, depends directly on this number. More specifically, the information dimension is a measure of the amount of information that can retrieved each time the state $\boldsymbol{x}(t)$ of the system is recorded at some time t, with an accuracy equal to ℓ. This dimension is related to the entropy $S(\ell)$ of the system, which is given by the equation

$$S(\ell) = -\sum_{i=1}^{M(\ell)} p_i \ln(p_i) \tag{4}$$

where p_i is the probability to find the state point of the system $\boldsymbol{x}(t)$, within the i_{th} cell. In the most common cases, the entropy $S(\ell)$ is varied as a function in the form

$$S(\ell) \simeq \text{constant} - \sigma \ln(\ell) \tag{5}$$

and therefore, the information dimension can be defined as

$$\sigma = \lim_{l \to 0} \frac{-S(l)}{\log(l)} \tag{6}$$

- *Correlation dimension*. The correlation dimension ν is a measure of the correlation that exists between the points belonging to the attractor. Starting from a sequence of state space vectors $\boldsymbol{x}(t)$ that have emerged as the result of the reconstruction of a time series $z(t)$ in the appropriate embedding space, the number of pairs of points $(\boldsymbol{x}_i, \boldsymbol{x}_j)$, whose distance is less than the side size ℓ, can be easily conunted. This number is given by the correlation integral

$$C(l) = \lim_{N \to \infty} \frac{1}{N^2} \left\{ \text{numbers of pairs } (i, j) : \left| \boldsymbol{x}_i - \boldsymbol{x}_j < \ell \right| \right\} \tag{7}$$

For a reconstructed embedding space of dimension m, the correlation integral is given by the equation

$$C(l) = \int_0^l \left(\mathrm{d}^m r\right) C(r) \tag{8}$$

where

$$C(r) = \lim_{N \to \infty} \frac{1}{N^2} \sum_{i,j=1}^{N} \delta^m \left(\boldsymbol{x}_i - \boldsymbol{x}_j - \boldsymbol{r}\right) \tag{9}$$

is the correlation function and $\delta^m\left(\boldsymbol{x}_i - \boldsymbol{x}_j - \boldsymbol{r}\right)$ is the Heaviside function, namely, the step function. Based to this definition of the correlation function $C(r)$, the correlation dimension ν, is just the boundary

$$\nu = \lim_{r \to 0} \frac{\ln(C(r))}{\ln(r)} \tag{10}$$

It turns out, that for small values of the distance ℓ, the quantities $C(\ell)$ and ℓ^ν are related as $C(\ell) \sim \ell^\nu$. Therefore, the correlation dimension is a useful quantity that allows us to describe the behavior of the attractor at a local level. The main advantage of this quantity, is that its calculation is much easier compared to the calculation of the values of the other two types of dimensions. Moreover, it is important to mention, that this type of dimension is not associated with the whole attractor, but with a specific trajectory of the attractor. Therefore, if only just one of the attractor's trajectories has been recorded, it is possible to estimate the correlation dimension, even though this is the only available information from the dynamical system under consideration.

These three different types of dimensions are related to each other, and in cases

where the coverage of the attractor's surface by hypercubes of side length ℓ is uniform, they have almost the same value. In the general case, it turns out that the inequality $\nu \le \sigma \le D$ holds, and therefore, the correlation dimension is the lower bound of the information dimension, which in turn, is the lower bound of the capacity dimension. The proof of the above inequality, as described by Grassberger and Procaccia, is done in three steps and involves the proof of the three simple inequalities $\sigma \le D$, $\nu \le D$ and $\nu \le \sigma$ [2].

2. The Experimental Estimation of the Chaotic Attractors Dimensions

A computationally simple and easy to implement approach to calculate the fractal dimensions of a chaotic attractor directly from experimental time series, was devised by Liebovich and Toth [4] in 1992. The central idea of this method, is the observation that the fractal structures are characterized by the property of self-similarity. This means that the fractional dimension of an attractor, can be computed by comparing properties of the physical system, which have been computed at different scales of space. The key feature of this method, which is presented in the following pages, is that allows the estimation of the dimension of the chaotic attractor, based on an experimentally recorded time series and without knowing anything else about the nature and the characteristics of the considered dynamical system.

Liebovich and Toth, following the usual practice of counting the minimum number $N_B(\varepsilon)$ of d-dimensional hypercubes required to cover the surface of the attractor, defined the experimentally calculated capacity dimension d_B by the relation

$$d_B \cong \frac{\mathrm{d}\log\left(N_B(\varepsilon)\right)}{\mathrm{d}\log(1/\varepsilon)} \tag{11}$$

This dimension is known as the box counting dimension, since its calculation is based on counting the minimum number of hypercubes $N_B(\varepsilon)$ of side size ε, that are required to cover the surface of the attractor. In full accordance with the computation method of the capacity dimension, each hypercube defined in the d dimensional reconstructed space, contributes to the dimension $N_B(\varepsilon)$ of the attractor, if it contains at least one of the points defining the attractor. This procedure is applied for a set of discrete values of the side size ε which has the form $\varepsilon = 2^m$ $(0 \le m \le k)$, where $k \in \mathbb{N}$, is a parameter of the problem. The algorithm uses the hashing technique to encode all the points in the reconstructed space, in such a way that points that belong to the same hypercube, are labeled by the same code number. After the application of this process, the codes of the points created in this way are sorted using any of the well known sorting methods, such as quicksort or heapsort, and then, the number of code numbers that are different from each other, is counted. It is not difficult to see, that this number is none other than the number of hypercubes required to cover the surface of the

attractor, used in the estimation of the fractional dimension. According to Liebovitch and Toth (see [5]-[9]), even though there are many other algorithms for the calculation of the attractor dimension in this way, however, these techniques require the use of a prohibitively large number of points, leading thus to very high memory requirements, as well as computation cost.

In an algorithmic description, the method of Liebovich and Toth, is composed by the following steps:

- Starting from the experimentally recorded time series $\{x(t)\}$, the points of the reconstructed attractor $\boldsymbol{r}_i = \{x_1, x_2, \cdots, x_d\}$ are created, using appropriate values for the embedding dimension d and the time delay τ.
- The coordinates x_i $(i = 1, 2, \cdots, d)$ of each one of the reconstructed attractor points, are normalized via its projection in the interval $(0, 2^k - 1)$, where the integer k is a parameter of the problem. After the normalization of each coordinate, the algorithm considers only the integer part of it. The fact that these values are integer numbers in the interval $(0, 2^k - 1)$, allows their straightforward conversion to binary numbers with a length of k bits.
- The space occupied by the attractor, is covered with hypercubes of side size 2^m, in such a way, that each point of the attractor is assigned to one of these hypercubes. Then, a binary number M of k bits length is defined, whose value depends on the value of the parameter m. More specifically, the number M is configured such that the first $k - m$ bits have a value of 1, while, the last m bits have a value of 0. For example, if the parameters k and m have the values of 10 and 4 respectively, the value of the binary number M will be equal to 1,111,110,000.
- For each coordinate x_i of each trajectory point, a second binary number y_i of length k bits is defined as $y_i = x_i$ AND M, namely, as the logical conjunction between the corresponding bits of the binary numbers x_i and M. For example, if x_i is equal to 1,010,111,010 and M is equal to 1,111,110,000, then the y_i value according to the above definition, is equal to 1,010,110,000. This logical conjunction between the values of x_i and M, is analogous to the one performed between a 32-bit IP address and a subnet mask of equal length. As it is well known from computer networks, for the networks of class C that use 24 bits for the network part and 8 bits for the host part, the subnet mask has a value of 255.255.255.0 namely, a binary value with 1's for the network address part and 0's for the host address part. Therefore, the result of the logical conjunction between the IP address and the subnet mask, returns the network address of the computer, whose value is the same for all computers that belong to the same subnet. In the same way, in the algorithm of Liebovich and Toth, the operation $y_i = x_i$ AND M will remove the differences in point coordinates due to their different positions inside the same hypercube, returning thus in the variable y_i, the coordinate of the origin of the hypercube that contain these points. In other words, the value of y_i is the same for all the points that belong to the same hypercube.

- For each reconstructed point $\boldsymbol{r}_i$, its coordinates are transformed according to the above procedure, to form d binary numbers of length k bits each. Then, a new binary variable z_i is defined with a length of dk bits via the concatenation of the d binary numbers. This variable is treated as a string. To understand this process, let us consider a reconstructed three-dimensional attractor ($d=3$), as well as a point $\boldsymbol{r}=\{x_1,x_2,x_3\}$, whose coordinates after projection in the interval (0, 1023), corresponding to the value $k=10$, have the values $x_1=389$, $x_2=740$, and $x_3=67$ in the decimal system, namely, the values $x_1=0110000101$, $x_2=1011100100$ and $x_3=0001000011$, in the binary system, respectively, each one of them, has a length of $k=10$ bits. Using a hypercube side size of $\varepsilon=16$ corresponding to the value $m=4$, the binary variable M is equal to 1,111,110,000. In this case, the logical conjunction between the coordinates x_1, x_2 and x_3 of the point $\boldsymbol{r}$ and the binary variable M, will yield the values $y_1=0110000000$, $y_2=1011100000$ and $y_3=0010000000$. Therefore, the string z corresponding to the point $\boldsymbol{r}$ and formed via the concatenation of the values of y_1, y_2 and y_3, will have the value $z=011000000010111000000010000000$ with a length of size $dk=30$ bits.
- The above procedure is repeated for each one of the N points $\boldsymbol{r}_i$ of the reconstructed chaotic attractor, and the values of the corresponding variables z_i are calculated. Due to the particular way of forming these variables, it is not difficult to note, that if two attractor points belong to the same hypercube, they will have the same values for the variables z_i, otherwise, the value of z_i associated with them, will be different. Therefore, to estimate the attractor's dimension, the sequence of z_i values has to be sorted using one of the available sorting algorithms, such as quick-sort, and the number of the different values that belong in the sorted sequence has to be counted. The number estimated in this way, is identical to the number of d dimensional hypercubes, $N_B(\varepsilon)$, required to cover the surface of the attractor, and therefore, it is a measure of the attractor's dimension.

If this procedure is applied for a set of values of m, a diagram in the form $\left(\log N_B(\varepsilon), \log(1/\varepsilon)\right)$ can be constructed, allowing the estimation of the attractor's dimension as the slope of the least square line that fits to the data of this diagram. Note, that in this curve fitting procedure, not all the estimated data values are used, due to the finite number of reconstructed points, as well as to reduction in discriminative power associated with it. This is particularly true, for the values of $N_B(\varepsilon)$ corresponding to the parameter values $m=k$ and $m=k-1$. Another problem associated with the small values of m, is the reduction of the number of points of the phase space, assigned to hypercubes with a very small side size ε. In the extreme case, the quantity $N_B(\varepsilon)$ is subject to saturation. In other words, each hypercube of the phase space, contains a single point, meaning that $N_B(\varepsilon)=N$. To avoid all these complications, the values of m that used in this procedure, are only the ones for which the relation $N_B(\varepsilon)\le N/5$ holds.

The practice has shown that the application of the method of Liebovich and Toth, leads to better results compared the ones given by other methods, which in general, depend on the nature of the chaotic system, as well as the quality and the number of the recorded data. Other advantages of this method, include the relatively small memory requirements with a value of $\mathcal{O}(Nd)$, where N is the number of points and d is the embedding dimension of the embedding space, as well as the relatively short computation time, which is equal to $\mathcal{O}(N\log N)$.

3. The Algorithm of the FD3 Application

In 1992, Sarraille & DiFalco implemented a variation of the algorithm of Liebovich & Toth in a form of a computer program identified by the name FD3 [10]. This application written in C programming language, allows the estimation of the three basic types of fractal dimensions (namely, the capacity dimension, the correlation dimension, as well as the information dimension), from experimental data series [12]. The FD3 application that is called as an executable from the command line, gets as input in the form of a command line argument, a text file whose contents is not the original time series data, but instead, the reconstructed trajectory points organized in tabular format namely in rows and columns, with the number of columns of this data organization to give the value of the embedding dimension d of the reconstructed state space. This fact means that the use of the DF3 applications, requires the reconstruction of the system trajectory from an experimental time series using some other application, different than FD3 and the storage of the trajectory points to a tabulated text file that can be used by the FD3 application as the input file. According to Sarraille & DiFalco, the reliable estimation of the values for the three fractal dimensions using their method, requires the use of at least 2^{4d} time series points. This means that for an embedding dimension with a value of $d=3$, the number of time series points should be at least equal to 4096.

To carry out the estimation of the three fractal dimensions, Sarraille & DiFalco used a generalized dimension $D(q)$ defined as

$$D(q)=\frac{I(q,\varepsilon)}{log(\varepsilon)} \tag{12}$$

where

$$I(q,\varepsilon)=\frac{1}{1-q}\log\left(\sum_{m=1}^{N(\varepsilon)}p(m,\varepsilon)^{q}\right) \tag{13}$$

It is not difficult to note, that this generalized dimension is reduced to the capacity, information, and correlation dimension, for the values $q=0$, $q=1$ and $q=2$, respectively. It can be proven, that the estimation time of this computation is equal to $\mathcal{O}(N\log N)$ where N is the number of the trajectory points, in fully accordance with the algorithm of Liebovich & Toth. In the above expression, the quantity $p(i,\varepsilon)$ expresses the percentage of the attractor points that lie within the i_{th} hypercube of side size ε, while $N(\varepsilon)$ is the minimum number of hy-

percubes required to completely cover the space occupied by the attractor.

The value of k used in FD3 application is equal to $k = 32$, and therefore, the normalization of the coordinates of the trajectory points is performed via their projection in the interval $\left(0, 2^{32} - 1\right)$, namely, in the interval [0, 4294967295]. This k value leads to the counting of the number of cubes $N_B\left(\varepsilon\right)$ required to cover the space occupied by the attractor, as well as all of the values of other related quantities, for 32 different values of the side size $\varepsilon = 2^m$ $\left(0 \le m \le 32\right)$. More specifically, for each value of side size ε, the application estimates and stores in the computer memory for further processing, the following information:

- The current value of the parameter m, which is equal to the logarithm (base 2) of the side size ε, namely, $m = \log_2 \varepsilon$.
- The number $N\left(\varepsilon\right)$ of hypercubes of side size ε, required to cover the space occupied by the attractor, by applying the algorithm of Liebovich & Toth.
- The logarithm $\log\left(N\left(\varepsilon\right)\right)$ of the number $N\left(\varepsilon\right)$ defined above.
- The value of the sum

$$I\left(\varepsilon\right) = \sum_{i=1}^{N(\varepsilon)} z_i \log\left(z_i\right) \tag{14}$$

where z_i is the number of points assigned to the i_{th} hypercube of the phase space. This value is required for the computation of the information dimension σ.

- The value of the sum

$$F\left(\varepsilon\right) = \sum_{i=1}^{N(\varepsilon)} z_i^2 \tag{15}$$

where z_i is the parameter defined above. This value is required since its (negative) logarithm is used for the estimation of the correlation dimension.

After the estimation of the above quantities for each value of side size ε, the application computes the approximate values of the three fractal dimensions for each one of spatial scales, via the calculation of the values $\log\left(N\left(\varepsilon\right)\right) - \log\left(N\left(2\varepsilon\right)\right)$, $I\left(\varepsilon\right) - I\left(2\varepsilon\right)$ and $F\left(2\varepsilon\right) - F\left(\varepsilon\right)$. This calculation is performed for the whole range of values of the parameter m, namely for the interval [0, 31], followed by the application of the least squares method to identify the slopes of the three data fitting lines, representing the three fractional dimensions. It is interesting to note, that the least squares method does not use all the above values, but only the ones corresponding to the m values in the interval [22, 29]. This is due to the fact, that according to the theory described above, the values $0 < m < 22$ are associated with saturation effects, while the values $m > 29$ are characterized by very low discriminative power, leading thus to the computation of incorrect values for the three fractal dimensions.

4. The Implementation Details of the FD3 Application

According to the above description, the DF3 application takes as argument from the command line a tabulated text file containing data in the form

20		
0.42	1.42	−0.972644
0.1	1.48214	1.48214
−0.5864	−0.5864	0.898604
0.42	−0.972644	−0.972644
0.48214	1.48214	0.300718
−0.5864	0.898604	0.898604
−0.972644	−0.972644	1.57915
0.48214	0.300718	0.300718
0.898604	0.898604	−1.0035

(the data in this example, are associated with the Henon chaotic map and they represent trajectory points in a three-dimensional embedding space). The single value in the first line (the value 20 in this example), represents the number of trajectory points L, while, each one of the lines that follow contain the coordinates of those points in the reconstructed space, with the consecutive lines to contain the coordinates of consecutive trajectory points, one point per line. To determine the embedding dimension d, the application counts the number of data values of the first line (in this example there are three coordinate values, namely the value 1.42, 1.42 and −0.972644 leading to an embedding dimension with a value $d = 3$), namely, the number of columns in this tabular organization. After the identification of the values of these two parameters (namely, the number of points and the embedding dimension), the application proceeds to the retrieval of the coordinates of the trajectory points, reading these points, one after the other. Since each one of these values should be projected in the interval $\left[0, 2^k - 1\right]$, the application after the retrieval of each value from the data file, proceeds immediately to its normalization. The upper limit of the above interval, namely the value of $2^k - 1$, is stored in the maxdiam variable, which is initialized to the maximum value of an unsigned long int C data type. On the other hand, the k parameter is described by the source code variable numbits. The integer values resulting from this normalization, are huge (an illustrative such value is a value of 9,315,461,120,293,234,688 which corresponds to the binary number 1000000101000111001010101000100010010010000010100110100000000000 of 64 bits length) and they are stored in an array named data, a two-dimensional array of dimensions $(L+2)\times d$ of type unsigned long int (therefore, this array is declared as unsigned long int **data). This process that reads the point coordinates, normalizes them on the fly and stores them in the cells of the data array is demonstrated in Figure 1.

The estimation of the fractal dimensions via the application of the algorithm of Sarraille & DiFalco, requires the sorting of the normalized coordinates of the trajectory points and the sorting algorithm used in the FD3 application, is the

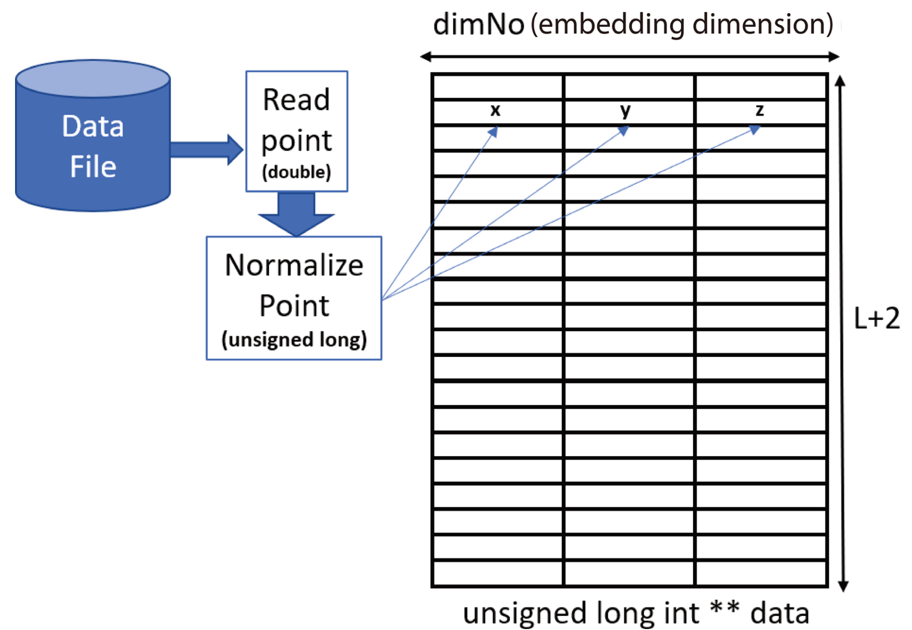


Figure 1. Reading and normalizing the points of the reconstructed trajectory in the FD3 application.

radix sort algorithm [11]. In the case of decimal numbers, the sorting of a set of four-digit numbers, requires four stages, that perform the sorting of the numbers with respect to the first, the second, the third and the fourth digit respectively. In fact, there are two variations of this sorting algorithm, depending on whether the digits to be sorted are traversed from left to right or from right to left. In each case, the fact that the decimal system is characterized by the use of ten digits, imposes the use by the algorithm of ten memory areas (or buckets) to store the numbers to be sorted in each of the above four phases. In the case of the FD3 application, because the data to be sorted are binary numbers of 64 bits in length, the process requires the use of only two such memory areas, which are implemented as queues named Q_0 and Q_1. Since this sorting procedure is performed one bit at a time, the radix sort operation is composed of 64 phases, each one of them, involves the sorting according to the value of a specific bit position, with the traversal of the bits to be performed from right to left, namely from the least significant bit to the most significant one. In FD3 application, each one of the 64 phases is performed by the rad_pass function, which takes as an argument an integer coord that contains the value to be sorted, as well as a second integer bitpos, that defines the position of the bit within the number, used for the sorting during each phase. The radixsort function, calls repeatedly the rad_pass function 64 times, one time at the beginning of the procedure to sort the numbers by the least significant bit (corresponding to the value *bitpos* = 0) and then, another 63 times via a single (one-dimensional loop) whose counter varies from 1 to *bitpos*. In each iteration of this loop, the process of the sorting is performed with respect to the currently used bit of all the coordinates of each of the L trajectory points, and for this reason, the rad_pass function in each iteration of the above loop, is called a total of d times

(where d is the embedding dimension), via an inner loop of d iterations, in order to sort all the d coordinates of the reconstructed points. To speedup the process, each time a point is read from the data file and normalized in the way described above, it is automatically placed in one of the two queues, according to the value of the least significant bit of the last coordinate of the reconstructed point located in column $d-1$ of the two-dimensional data array. In particular, if this bit has a value of 0, then the point is placed in queue Q_0, while if this bit has a value of 1, then the element is placed in queue Q_1.

Each one of the two queues Q_0 and Q_1 required by the radix sort algorithm, is fully identified by the memory addresses of the first and the last element of the queue, described by the variables Qfront and Qrear, respectively, or equivalently, by the QHeader data structure composed by these variables, and therefore, defined as

```
typedef struct QHeader { long int Qfront, Qrear; } QHeader;
```

To save space and increase the performance, these queues are not implemented in a physical way, namely, as separate data structures, with the coordinate values to be physically added to those queues, but rather in a virtual or logical way, via arrays of indices, as it is shown in **Figure 2**. In this figure, the first element placed in the queue and identified by the Qfront pointer, is located in cell number 6 of the two-dimensional data array (remember, that in C programming language, the index of the first element of an array has a value of zero), while the next element of the queue (remember that the queue is a FIFO structure, with Qfront being the element added first and Qrear being the element added last), is the one identified by the contents of cell next_after [6]. The contents of this cell is a value of 11, meaning that the second element added to the queue and considered as the next element of Qfront, is the one stored in the cell number 11 of the data array.

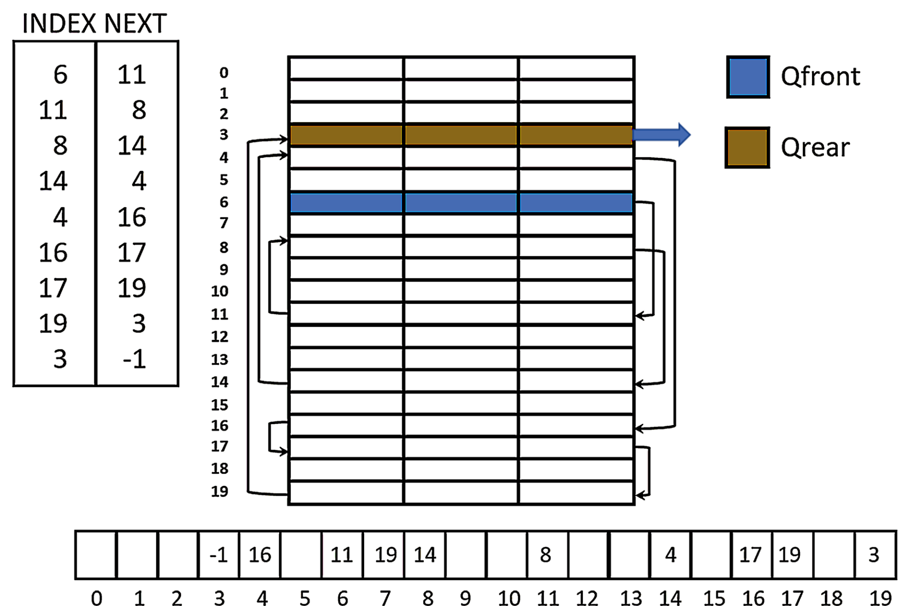


Figure 2. Use of queues in the FD3 application.

Continuing in the same way, the third element of the queue is stored in the cell number 8 of the data array. Noting from the figure that next_after [11] = 8, the third element of the queue is in cell number 14 of the data array, since next_after [8] = 14, and so on. To determine the last element of the queue,the cell of the next_after array that contains the value −1, has to be identified. This cell is the cell number 3 of the next_after array, since next_after [3] = −1. Therefore, the last element of the queue identified by the Qrear index, is cell number 3 of the data array. In this figure, only the half of the elements of the next_after array are filled in, while the remaining half of the array elements are assumed to contain the corresponding information for the second queue that is maintained and used in exactly the same way.

The insertion and the removal of points in these two queues, is performed by the functions

```
void en_Q (QHeader *p_QHeader, long int pointer)
```

and

```
void de_Q (QHeader *p_QHeader, long int *p_pointer)
```

respectively, while, the movement of a point from the first to the second queue and visa versa during the execution of the rad_pass function, is performed by the transf_Q function, that takes as arguments the queues involved in this process. Since these queues are FIFO structures, this function first calls the de_Q to remove the head of the source queue and then the function en_Q, to place this item to the tail of the target queue. Note, also, that in FD3 application, the queue management is supported by a marker array parameter, which is used in the way described below. There are two such markers, namely, one marker for each queue, and because the information regarding these fields, is also stored in the next_after array, this array does not have a length of L, but instead a length of $L+2$. Because these fields can be located in any cell of the next_after array, the application uses another array, named marker, which also has a length of $L+2$. According to the adapted notation, if an element of this array has a zero value, then the corresponding element in the next_after array contains a reference to a data value, otherwise (namely, when a value of 1 is used), this element refers to a marker. Therefore, the marker array, will have L elements with a zero value, as well as two elements with a value equal to unity, identifying the two markers associated with the two queues. When these queues are created at application startup, the first element added to each one of them, is a marker field. Then, during the file opening procedure and the retrieval of the data values, each queue is populated with data, in the manner described above. When the radix sort procedure is started, and the rad_pass function is called for the first time, the two markers are located at the heads of their respective queues. To start the process, the two markers are moved from the head to the tail of the associated queue. During the execution of rad_pass function, the data points are transferred in the appropriate way, from the first to the second queue, and visa versa, according to the value of the currently examined

bit. Since the points are placed always at the tail of queues, the two markers move progressively towards the head of their associated queue. When these markers appear at the head of the queue, this means that all the data points have been processed, and the procedure is considered completed. Therefore, the reappearance of the two markers at the head of their queues is the terminating condition of the process. This procedure is complex enough, but it is the only way to run the algorithm, since, even though the number of trajectory points retrieved from the data file, is known in advance, this is not true for the numbers of points that belong to each queue (remember, that during the file read operation, the data values are inserted in the two queues according to the value of the least significant bit of a binary number, and therefore the initial length of the two queues is strongly dependent of the data values read from file).

Based on the above description, the algorithm of the rad_pass function is as follows:

1) The marker of each queue is moved from its head back to its tail by calling the functions de_Q and then en_Q (the same result can be achieved by using the transf_Q function, if the same queue is used for its first as well as its second argument).

2) For each queue $Q[i]$ $(i=1,2)$, and as long as the marker[Q[i].Qfront] cell has a value of zero (meaning that the head of the queue $Q[i]$ contains a data value and not a marker), perform the following actions:

- Check the value of the bit in the current bitpos position of the data value located in the data[coord][Q[i].Qfront] cell.
- If the value of this bit is equal to 1, the $Q[i]\rightarrow Q[1]$ transfer is performed.
- Otherwise (namely, if the value of this bit is equal to 0), the transfer $Q[i]\rightarrow Q[0]$ is performed.

On the other hand, the radixsort function that performs the actual sorting of points, the only thing that has to do, is to call repeatedly the rad_pass function for each bit position, for each coordinate of each point, and for each dimension. Assuming that the trajectory points read from the data file, have coordinates x, y, z (the generalization to more dimensions is straightforward), the coordinates of the k_{th} point are retrieved from the k_{th} line of the two-dimensional data array, with the coordinate x being (after normalization) at the position data[0][k], the coordinate y being at the position data[1][k] and the coordinate z being at the position data[2][k]. Remembering that during the data retrieval from the file, the application (as a preprocessing step) places the data in the appropriate queue, based on the value of the least significant bit (located at bitpos = 0) of the last coordinate (in our example, the z coordinate) of each point, the radixsort function does not repeat this process again, and therefore, when is calls the rad_pass for the position bitpos = 0, it performs this process only for the other two coordinates x and y of each point (specifically for this bitpos value). On the other hand, for all the remaining bitpos values and for each coordinate of each point, the radixsort function performs the sorting in the appropriate queue, for all coor-

dinates x, y and z of each point. The result of this process is a complete sorting of all the trajectory points in such a way, that all the points in which the most significant bit of the x coordinate is equal to 0 being sorted in queue Q[0], while, all the points in which the most significant bit of the x coordinate is equal to 1 being sorted in queue Q[1]. In the last step, the radixsort function removes the two markers from the queues, since they are no longer needed, so that the sweep function which is called next, to sweep in the appropriate way the two sorted queues, and collect the data required for the estimation of the fractional dimensions.

Based on the above description, the algorithm of the radixsort function is as follows:

1) For each coordinate i $(0 \le i \le d-2)$ where d is the embedding dimension, call rad_pass(i,0).

2) For each bit location B $(1 \le B \le \text{numbits})$ and for each coordinate i $(0 \le i \le d-1)$, call rad_pass(i,B).

3) At the end of the process, remove the markers from the tails of the queues.

In the next step of the process, and after the sorting of the queues Q[0] and Q[1], the sweep function is called to scan the two sorted queues and collect the required information, to calculate the necessary parameter values for the estimation of fractal dimensions. At each stage of this iterative process, and for each queue, the function compares two consecutive queue points to get a positive or a negative answer to the question whether these two points belong to the same hypercube or not, for each one of the numbits different hypercube sizes. This process starts from the bit in the position numbits-1 that corresponds to the maximum hypercube size. Note, that even though the two points may belong to the same hypercube for a specific size ε, however, this may not be the case, for some other such size. This situation is illustrated in **Figure 3**, depicting the Henon's attractor in a two-dimensional space, with the hypercubes being simple squares. From this figure it is clear that the points A and B belong to the same hypercube (the hypercube 3) for a size of $\varepsilon = 8$, but to different hypercubes (the hypercubes 1 and 2), for a size of $\varepsilon = 4$. The comparison described above, is carried out separately for each coordinate and it is based to the application of the logical conjunction between the corresponding coordinates of the points under consideration. If the embedding dimension is equal to d and therefore, each one of the two points P_1 and P_2 is described by d coordinates of the form $(x_1, x_2, \cdots, x_n)$ and $(y_1, y_2, \cdots, y_N)$ respectively, it is not difficult to see, that the above procedure requires the application of d logical conjunctions of the form x_i AND y_i, whose outcomes are stored in the corresponding cells of an one-dimensional array named diff_test of d elements, allocated during the initialization phase and after the retrieval of the embedding dimension value from the data file. This test is performed via a doubly nested loop, with the outer and the inner loop to allow the specification of the examined coordinate (in the ordering z, y, x),

as well as the examined bit position (from the value of numbits-1 to the zero value), respectively. If the bit position under consideration has a value of ℓ, this procedure examines whether or not the current consecutive examined points belong to the same hypercube with a side size of $\varepsilon = 2^{\ell}$.

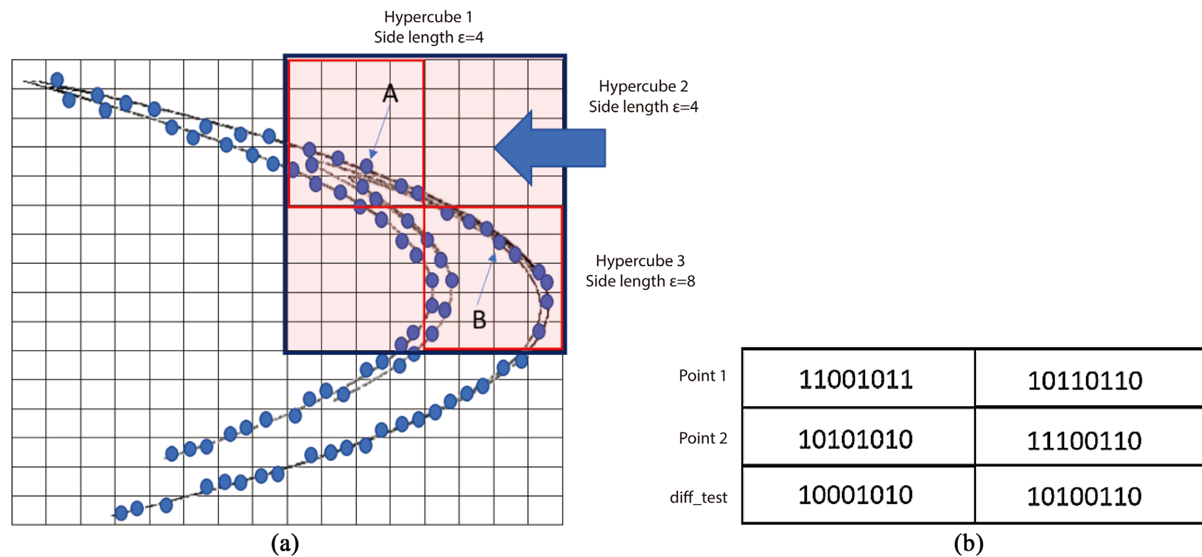


Figure 3. (a) Although two trajectory points may belong to the same hypercube of a certain size, they may belong to different hypercubes of different sizes; (b) In order to examine whether or not two such points belong to the same hypercube or not, the logical conjunction operation is used.

The estimation of the fractal dimensions is based on the values of the parameters estimated by the sweep function, whose calculation, in turn, requires the proper initialization of the following arrays, each one of which has a length of numbits + 1:

- boxCountS: the number of hypercubes required to cover the space occupied by the attractor.
- pointCount: the number of points that belong to these hypercubes.
- sumSqrFreq: the sum of the squares of the relative frequencies of the points in all hypercubes containing points.
- informationS: a sum used to calculate the information dimension.

The scanning process described above, is an iterative process applied to each sorted queue separately, first to the queue $Q[0]$ and then to the queue $Q[1]$. During the initialization phase, all the cells of the boxCount and pointCount arrays, are set to 1 while, all the cells of the sumSqrFreq and informationS arrays are set to a zero value. The two consecutive points compared at each iteration step to determine whether they belong to the same hypercube or not, for the currently examined hypercube side size, are determined by the variables previous and current. At the initialization stage, the previous variable identifies the first queue element ($Q[0]$ or $Q[1]$) that contains data, while the value of current variable, is set to the first element of the current queue under consideration. At the beginning of the next iteration, the new value of previous variable is the current value of

current variable, while the new value of current variable, specifies the next item identified by the next_after[current] cell value. The scanning process starts from the greatest hypercube side size, associated with the value numbits-1, and with the examined hypercube side sizes to decrease continually, until the minimum side size equal to unity is reached. For each examined dimension and for each examined bit position in the doubly nested loop, the function examines whether the points identified by the previous and current variables belong to the same hypercube or not. In the second case, which is the simplest one, there is nothing to be done, and the algorithm proceeds to examine the next dimension. On the other hand, in the first case, the algorithm is based on the observation that since the current point does not belong to the same hypercube, as happens with the previous point, this is something that will also holds for the smallers side sizes and therefore, a new hypercube must be created, in which this point should be placed. On the other hand, for bigger side sizes, the current point will belong to all hypercubes with a size larger the currently examined side size.

The implementation of the above procedure is based on the use of two one dimensional loops, that are executed one after the other. The first loop allows the processing of the hypercubes with a side size less than or equal to the current one (this means that the countpos counter of this loop, is changed from bitpos to 0 with a variation step equal to the unity). In each iteration of this loop, the value of the relative frequency is estimated via the expression

$$\text{freq} = \frac{\text{pointCount}[\text{countpos}]}{(\text{double})\text{dataLines}} \tag{16}$$

where dataLines is the number of lines in the data file. At this stage of the procedure, this parameter represents the number of points in the data file, but in a later stage this value is adjusted in the appropriate way. The value of the freq variable is raised to the power of 2, in order to be used in the calculation of the correlation dimension, and then it is added to the current value of the sumSqrFreq[pointcount] array element, that expresses the contribution of the hypercube currently examined, to the final value of this parameter. Furthermore, the value of the informationS[pointcount] array element is increased by the value of the ratio $\text{freq} \times \left[\log(\text{freq})/\log(2.0)\right]$, in order to be used in the calculation of the information dimension. Finally, the current value of the boxCountS parameters is increased by one, since a new hypercube has been encountered, while the value of the pointcount[countpos] is set to unity, since the only point we know so far that belongs to the new hypercube, is the point currently examined. After the execution of the above loop, a second loop starts to execute, to process the hypercubes with a side size greater than the currently examined size, up to the maximum size, meaning that the countpos counter of this loop, gets all the values from bitpos + 1 to numbits, with a variation step equal to the unity. If we take into account that as long as a trajectory point is assigned to a hypercube for some side size, then, it will be assigned in all hypercubes with bigger side sizes containing that hypercube, all we have to do, is to increase by one, the values of all the cells in the pointcount

array for the values bitpos+1 to numbits of the countpos counter.

In the last step of the sweep function, the contribution of the remaining data as well as the of last point of the reconstructed trajectory is added to the estimated quantities, and a normalization process is applied by dividing the values stored in the above arrays by the quantity boxCountS[0]. This value is equal to the number of hypercubes with side size of $\varepsilon = 1$ (namely, with the smallest side size) that contain reconstructed points. As it is pointed out by Saraille and Di Falco, for all the practical purposes, the value of this count is equal to the number of discrete points stored in the data file, since these hypercubes are extremely small compared to the ones with the maximum size. The points are scaled by the application in such a way, that the smallest difference that can be analyzed by the algorithm, to be equal to unity. This means that a hypercube with side size of $\varepsilon = 1$ contains only a single point of the input data set, since the application is unable to distinguish between points with a smaller difference.

The last two functions called to complete the process are the findMark function that returns the value of the largest marker whose number of points is greater than boxCountF[0]/cutoff_factor (the value of the cutoff_factor used in this estimation, is equal to $2^d + 1$ where d is the embedding dimension), as well as the function fitLSqrLine that implements the well known least squares data fitting procedure. This function is called from the GetDims function three times in the appropriate way, and returns the slope and the intersect of the associated least square line, with the slopes of the three lines estimated in this way, to be equal to the values of the three fractional dimensions. The use of the findMark function is justified by the fact, that according to the description of the DF3 application, the m values that are smaller than the one returned by this function, are not used in the calculations due to the appearance of saturation effects that may lead to incorrect values for the fractional dimensions. On the other hand, the fitLSqrLine function is declared as

void fitLSqrLine (long first, long last, double * X, double * Y, double * slope, double * intercept)

and estimates the slope as well as the intercept of the least squares lines described above, using the coordinates of the X and Y arrays from the start position to the end position. In all cases the X array is always the logEps array, containing the values of the parameter m in the interval [1, 31], while, the Y array, is the negLogBoxCountF for the case of the capacity dimension, the informationF array for the case of the correlation dimension, and the logSumSqrFreq array, for the case of the correlation dimension. The fitLSqrLine function is called three times by the GetDims function, and upon return, the slope argument of the fitLSqrLine function, contains the value of the appropriate dimension, with the values of the three fractional dimensions to be returned by the last three arguments of the GetDims function. The procedure is complete.

5. Parallelizing the FD3 Application

The parallelization of the serial FD3 application can be easily performed in an

efficient as well as a parametric way, by modifying some features that characterize the original serial program. For example, even though in the original application, the input data file is a trajectory file containing the coordinates of the points of the reconstructed trajectory (a point per line), in the parallel application, a time series file (that contains a single data value per line), can be used instead, as an input file. This modification allows the reconstruction of the trajectory by the application itself and not by some other application that created the initial trajectory file, allowing thus the experimentation with different embedding dimensions. In all practical cases, the most common values for the embedding dimension, are the values $d = 1$, $d = 2$ and $d = 3$.

Furthermore, in the serial application implemented 30 years ago and more specifically in 1994, the very limited RAM size of the order of 4 MBytes that characterized the computers of that time, imposed the management of the available memory in an efficient and ingenious way, leading thus the to complex pointer-based data structures, that make the code very complicated and difficult to understand. More specifically, the conversion of the data values to binary numbers of 64 bit length was performed on the fly during the retrieval of the point coordinates from the input data file, followed by the assignment of these values to the queues Q_0 and Q_1 (implemented virtually via pointers), as a preparation or a pre-processing step for the application of the radix sort algorithm. However, in the new parallel version, there are not such memory limitations, since the main memory of the modern computers has a size of the order of GByte. Furthermore, the complicated pointer-based implementation of the parallel application is very difficult (but of course, not impossible) to be supported by the traditional parallel programming models, such as the ones associated with MPI [12] and OpenMP [13]. Based on all these remarks, it is reasonable (after the reconstruction of the system trajectory) from the time series read from the input file, to store the required data in more tables. More specifically, the coordinates of the reconstructed points can be stored in a two dimensional array of doubles named data (this is the extra array that does not used in the serial implementation), while their binary equivalents are stored to a two-dimensional arrays of unsigned longs named coords (this is the array data of the serial implementation—see **Figure 1**). Of course, if someone wants to follow the initial serial approach, this can be done very easily. In other words, after the retrieval of each point coordinate from the trajectory file, or during the reconstruction of the trajectory from the time series file, it can be converted on the fly to binary format and stored in the coords array, while the data array is not needed anymore.

In a more detailed description, and keeping in mind that the serial FD3 application is composed of a lot of stages such that

- data retrieval from the input data file.
- normalization of data in the interval [0, *maxdiam* − 1].
- assignment of the data in the queues Q_0 and Q_1 according to the value of the least significant bit.

- sorting of the binary point coordinates via the radix sort algorithm.
- scanning of the ordered data point and collection of the required statistics.
- estimation of the three fractal dimensions via the least square fitting approach.

Let us describe now how these procedures can be performed more efficiently in a parallel way.

5.1. Parallelization of the Data Retrieval from the Input File

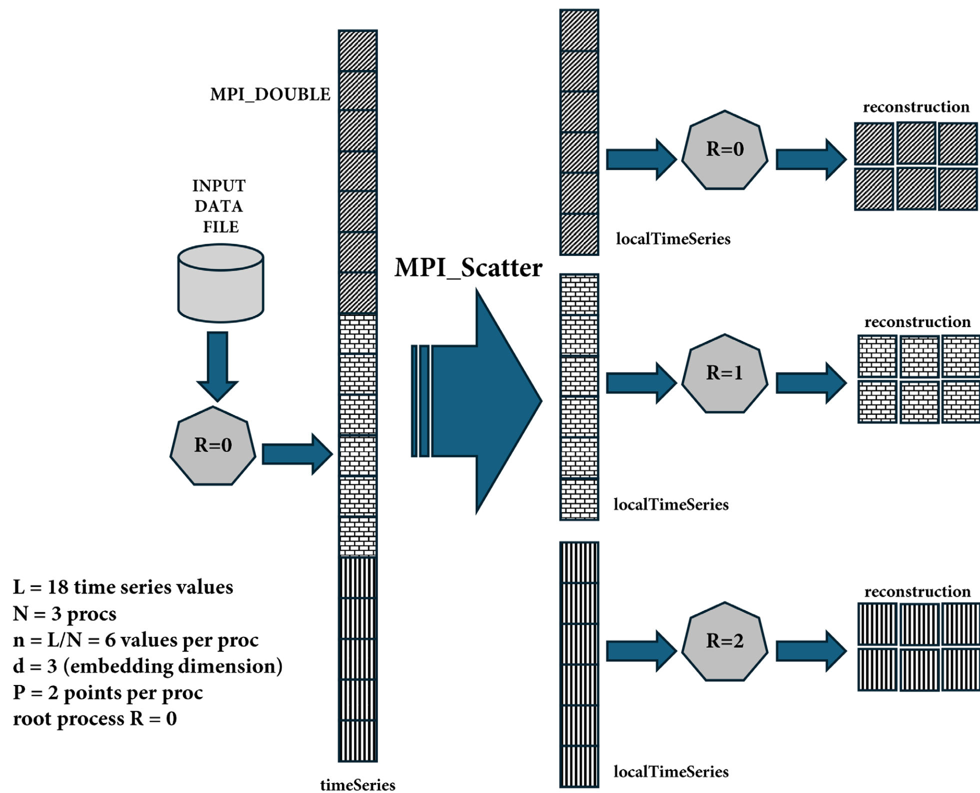
The most straightforward way to retrieve the data from the input file, is to use only one process to read the data values from the file, and then, to distribute these values to the other processes of the application. In the next step, each process will use its own data to reconstruct its own part of the trajectory and perform the necessary operations on this part, in a local fashion. In the MPI library, this procedure can be done very easily using the MPI_Scatter function to distribute to each one of the N processes of the application, a number of time series values equal to $n = L/N$ where L is the number of the trajectory points. The MPI_Scatter function is used in the form

```
MPI_Scatter (timeSeries, n, MPI_DOUBLE, localTimeSeries, n, MPI_DOUBLE,
root, MPI_COMM_WORLD);
```

In the above code, the root argument determines the root process of this collective operation, while the arrays timeSeries and localTimeSeries are supposed to have been allocated and initialized appropriately. This procedure is depicted graphically in **Figure 4**.

A more efficient implementation exploits a unique feature of the MPI library that allows the processes of the parallel application to open collectively the same data file and read concurrently its contents, with each process to retrieve different and non-overlapping data sections, as it is determined by a properly defined file view. Of course, since the MPI I/O functions work only with binary files, the text time series files should be converted to binary format, via a simple conversion utility implemented specifically for this reason. In the scenario shown in **Figure 5**, the time series of the input file contains 500 data values that are read concurrently by a group of $n = 5$ process, according to the pattern shown in the figure. In a more detailed description, each process reads 100 data values, and more specifically, the process $R = 0$ reads the first 100 values, the process $R = 1$ reads the next 100 values and so on. The value disp = 0 used in the figure, means that the data values start from the beginning of the file, and therefore, in this case there are no headers and related stuff in the beginning of the file that have to be ignored, a condition that generally holds. In our implementation, the first value stored in the file is an unsigned long value specifying the number of the time series points. This value should be skipped and therefore the disp argument should be initialized to a value of sizeof (unsigned long).

The most important of the MPI code that implements this procedure is shown below. In the first step, all the process call the MPI_File_open function to collectively open the input file that can be accessed via the fileHandle argument of type



MPI_Scatter (timeSeries, n, MPI_DOUBLE, localTimeSeries, n, MPI_DOUBLE, root, MPI_COMM_WORLD);

Figure 4. In this approach, the root MPI process reads the data from the input file and scatters them to the processes of the parallel application each one of them applies the reconstruction procedure to its own times series data.

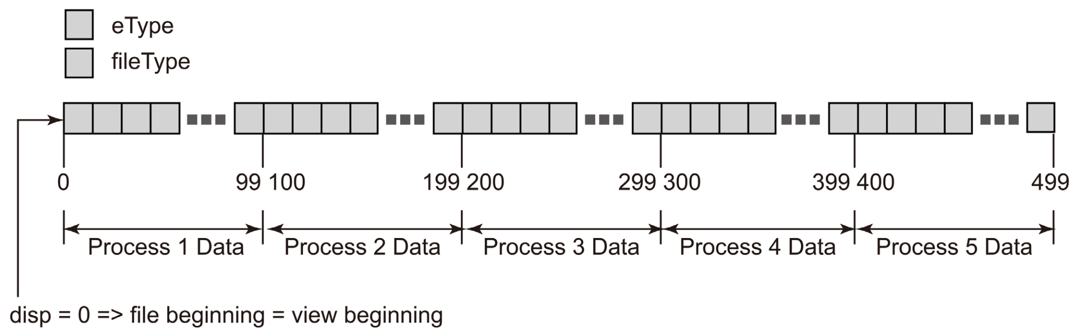


Figure 5. In this approach the available MPI processes open collectively the data file and read concurrently the appropriate portion of the data values.

MPI_File. In the second step, the processes call the MPI_File_set_view to define the file view using for the elementary data type (eType), as well as the file type (fType), the type MPI_DOUBLE. According to the above description the value of the disp argument is equal to sizeof (unsigned long). Finally, in the third step, the processes call the MPI_File_read_at to read the appropriate portion of the input data, starting from an offset value equal to rank*n, where rank is the process rank retrieved via MPI_Comm_rank and $n = L/N$ is the number of data that have to be read from the file and stored to the localTimeSeries array of the same length.

After the call of the MPI_File_close function that closes the input data file, each MPI process can perform the reconstruction procedure to its own portion of the input data, as in the previous case. For the detailed description of the arguments of each one of the above functions, the reader is referred to the rich MPI documentation.

```
MPI_File_open (MPI_COMM_WORLD, fileName, MPI_MODE_RDWR,
MPI_INFO_NULL, &fileHandle);
MPI_File_set_view (fileHandle, disp, eType, fType, "native", MPI_INFO_NULL);
MPI_File_read_at (fileHandle, offset, local-TimeSeries, n, MPI_INT, &status);
MPI_File_close (&fileHandle);
```

On the other hand, if the parallel application is based on the use of OpenMP, the parallel file access is not supported and therefore, the data retrieval operation will be performed by only one OpenMP thread. In this case the distribution of the data of the timeSeries array to the threads of the application that can be created via the directive #pragma omp parallel is based of the thread rank that can be retrieved via the appropriate function, named omp_get_thread_num. Note, that in this case the timeSeries array is a shared variable, and therefore the time delay of the message passing operations that characterize the MPI implementation is not an issue anymore.

5.2. Parallelization of the Reconstruction and the Binary Conversion Procedures

The parallelization of the reconstruction procedure is performed in the way described above by the set of the MPI processes, each one of them in the next step can perform the conversion of the data values to 64—bit binary numbers working with its own data values. Alternatively, this conversion can be performed on the fly, to the values stored in the localTimeSeries, after the MPI_File_read_at function, while the reconstruction can be applied to these binary values. Generally speaking, the procedures of the reconstruction and the conversion can be performed in any order, leading to the same result.

In the case of the OpenMP implementation, the reconstruction and the binary conversion is performed in the same way by each OpenMP thread using its own data portion that can be defined via the use of the function *omp_get_thread_num*. In an alternative implementation, these procedures can be performed only by the master thread using either both the arrays data and coords or only the array coord (if the conversion is performed on the fly). An alternative option is to ask from the master thread to perform the reconstruction and parallelize only the conversion process using a group of threads, each one of them can access the shared data array in order to create collectively the shared coords array.

5.3. Parallelization of the Data Sorting Operation

Even though in the original FD3 application, the sorting of the reconstructed trajectory points required by the algorithm, is based on the radix sort algorithm (this

algorithm works correctly when the data values to be sorted are described by the same number of digits, a requirement that is satisfied in this case since each value is an 64-bit binary number), however, if the algorithm modified in the correct way, this sorting can be done using anyone of the well known sorting algorithms, having thus the opportunity to evaluate and compare the performance of the application for all those cases.

In the current serial implementation, the nature of the numbers to be sorted (namely, binary numbers consisting of 0 s and 1 s), requires the use for the implementation of the radix sort of only two buckets, implemented in the form of pointer-based virtual queues. However, the fact that the modern computers generally have a lot of memory whose size is of the order of GByte, allows us to resort to a more easy implementation, and more specifically, to the use of a traditional queue data structure, implemented as an array or as a linked list. Since the number of binary data values whose LSB has a value of 0 or 1 can be easily counted in advance, the implementation of the array approach is a straightforward one, while the use of the marker mechanism of the serial application, is not required anymore. The two arrays that implement the two queues, can be allocated at run time via the malloc function. Regarding the insertion of the data values to these queues, it can be done in many ways. In the case of the MPI implementation, each MPI process can create and use its own queues (the queue Q_0 for the data values with an LSB equal to 0 and the queue Q_1 for the data values with an LSB equal to 1, , as it is shown in Figure 6). On the other hand, in the case of the OpenMP implementation, since these array based queues are shared data structures, we can use two OpenMP threads, in order to fill the queues Q_0 and Q_1 with the appropriate data values.

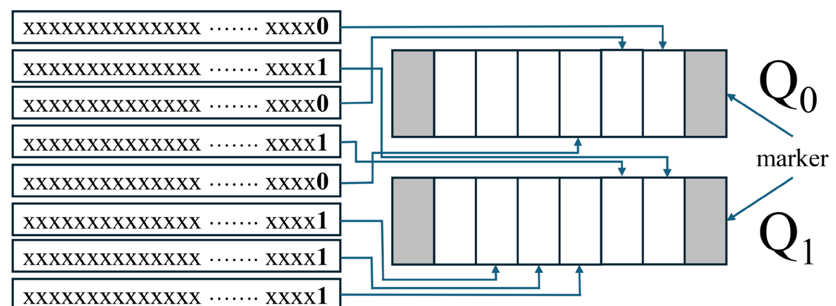


Figure 6. The assignment of the reconstructed trajectory points to the queues Q_0 and Q_1 according to the value of the least significant bit.

Returning to the radix sort algorithm applied to a set of binary keys each one having a length of 64 bits, it can be used in its serial form by each one of the MPI processes, even though this algorithm can also be parallelized. In a more detailed description, the serial radix sort algorithm is performed as follows:

- During each pass of the algorithm, scan m consecutive bits, starting from the least significant digit.
- Store the keys to be sorted into 2^m different buckets.

- After the completion of the above procedure, count how many keys each bucket has.
- Compute exclusive prefix sum for each bucket.
- Assign starting address according to prefix sums.
- Examine m bits to determine the target bucket and move the currently used key to that bucket.

On the other hand, in the parallel version of the radix sort, the process starts from the least significant bit, too, and then the algorithm moves to the next bit, until all data bits have been examined. More specifically, in any parallel sort implementation, there are multiple sections of regular sort algorithm that run on multiple processors in parallel. In this approach, a different sorting method can be used for each bit, such as counting sort or bucket sort. When each processor completes its work, the results are communicated between the processors, and this pattern maybe repeated multiple times until the whole sequence is completely sorted. The algorithm generally requires the same number of digits per data value. It is important to note, that this algorithm may not work in an efficient way with very large data values, or when the data to be sorted follow a skewed distribution. However, the algorithm is stable, meaning that it preserves the relative order of elements that have equal value.

In a more detailed description, the parallel radix sort implementation is similar to the serial one described above, but with a few exceptions. More specifically, if the algorithm is implemented using MPI, the keys to be sorted must be stored and moved across different processors, via message passing based inter-process communication. As a result, each processor can end up having varying number of keys to manage after each stage, a fact that requires additional work to be done for keeping track of these moves. The theoretical analysis of algorithm reveals that its performance depends of a lot of factors, such as the total number of keys to be sorted, the total number of available processors, or equivalently, the number of parallel sequences to be sorted by each individual processor, the time spent to sort the local sequence by each processor, as well as the time spent for processor communication. It can be proven that the complexity of the MPI based parallel radix sort algorithm is $\mathcal{O}(d(n+k))$ where d is the number of digits of each data value, k is the range of values for each digit, and n is the number of keys to be sorted. The execution of algorithm requires $\mathcal{O}(dp)$ messages, where p is the number of processors, as well as the use of $\mathcal{O}(n)$ data values. The memory requirements of this parallel algorithm to allow the communication required during each digit starting phase (this communication depends on the base m and the digit size) is $\mathcal{O}(n/p+k)$ per processor, since each processor has to store its local data as well as the auxiliary arrays for the counting or bucketing operation.

The parallel MPI based radix sort algorithm, is performed as follows (the following steps are executed in each pass of the algorithm, with the k_{th} pass to store the keys according to the value of the k_{th} bit):

- Split the initial problem set into p subsets or parts and assign each one of these subsets to one of the p available processors. Note, however, that in

general, the number of processors used in the procedure, may be different from the number of those subsets.

- Build a histogram by counting how many elements should go into each bucket, scanning m bits every pass (the paremeter m is called the base of the algorithm). This is a local operation performed by each processor using its own part of data. In pseudocode this is implemented as hist[subset][bucket]++.
- Move keys within each processor to the appropriate buckets (this is a local operation performed by each processor).
- Each processor waits until all processors finished the previous step, and then perform an one-to-all bucket permutation across processors to find prefix or partial sums over counts, revealing thus an initial index of each bucket for every part of data (this is a global operation as well as a sequential process).
- Send/receive keys between the processors and stores them in the appropriate position determined by hist[subset][bucket] (this is a global operation performed by all processors).

In pseudocode the entire pass of the parallel radix sort is performed as follows:

```
parallel_for part in 0..K-1
     for i in indexes(part)
         bucket = identifyBucket(buffer[i])
         hist[subset][bucket]++
base = 0
for bucket in 0..R-1
    for part in 0..K-1
        hist[subset][bucket] += base
        base = hist[subset][bucket]
parallel_for part in 0..K-1
     for i in indexes(part)
         bucket = identifyBucket(buffer[i])
         out[hist[subset][bucket]++] = buffer[i]
```

In the above pseudocode, the parameters K and R describe the number of cores and the number of buckets, respectively, since, in general, these numbers may be different each other. The data to be sorted are stored in the buffer array, while the bucket to which a data value is going to be assigned is identified via the identifyBucket function. Finally, the histogram created in the second stage is described by the hist two-dimensional table.

To implement an efficient algorithm, the partitioning of the individual sequences among processors must guarantee balanced workload in such a way that there are no idle processors. On the other hand, the communication needs to be performed as efficiently as possible, an objective that generally depends on the number of processors, the frequency of communication, the payload associated with the message passing operations, as well as the network latency, since the communication time is an important part of the total execution time. Regarding the scalability of parallel MPI based radix sort, it is not without limits, with the main

limiting factor to be the bandwidth and the latency of the network. This is due to the fact, that as we add more nodes to increase the degree of parallelization and the ability to handle larger problem sizes, there will be a point, above which, the network will suffer by over-saturation and message contention, due to the above limitations.

On the other hand, in the parallel OpenMP based radix sort implementation, the algorithm is performed in a similar way. However the main execution entity is not an MPI process but an OpenMP thread that uses the shared memory model. In this OpenMP based implementation, the array of data to be sorted is distributed to the available threads and each thread counts locally how many keys have to be assigned to each one of its local buckets. These local counts are then added together, to give the global count value for each bucket. Since the array of global counts for each bucket that gives the required histogram is a shared variable, a #pragma omp critical directive has to be used. After the completion of this procedure by all threads (a directive in the form #pragma omp barrier can be used to implement this synchronization), a single thread (#pragma omp single) is used to compute the offsets and then, each thread one after the other, uses the appropriate part of this array to estimate the prefix sum. In the last step, via a parallel for loop, each thread distributes its values in order the data to be moved to the appropriate bucket.

In the case of very large data sets, the MPI and OpenMP based approaches can be combined to increase the efficiency of the parallel applications. In other words, after the distribution of the time series data to the MPI processes and the procedures of the reconstruction and binary conversion that leads to the coords array, each MPI process can create the appropriate number of the OpenMP threads to sort its local coords data, In the last step, all these sorted coords local copies should be merged to the final sorted coords global array, via an appropriately defined global operation. This procedure is one of the most well-known MPI applications and can be implemented in many different ways.

5.4. Parallelization of the Sweep Function

According to the description of the sweep function given above, this function is composed by two loops. The first loop is a two-iteration loop with each iteration to process each one of the two sorted queues Q_0 and Q_1, while the second loop is a for loop with numbits + 1 iterations that performs the required post-processing of data. The parallelization of those loops can be performed either via MPI either via OpenMP. In the first case the application can use two MPI processes, one process for the processing of queue Q_0 and a second process for the processing of the queue Q_1, while in the second case this procedure is based on the use of two OpenMP threads. In the MPI approach each process will make its own contribution in the values of the cells of the arrays boxCountS, sumSqrFreq, pointCount, and informationS and then, via a message passing collective operation, these values will be added each other to estimate the final values used in the

least square fitting. Even though each MPI process executes loop structures that could be parallelized by creating the appropriate number of threads, this parallelization is not possible using OpenMP, since these loops are do... while loops whose total number of iterations can not be estimated in advance, as it is required by the OpenMP specifications. On the other hand, in the OpenMP approach, the addition of the partial sums to estimate the final values can be performed in the usual way, via the #pragma omp critical directive. Regarding the second loop of numbits + 1 iterations it can be parallelized in the same way and since its number of iterations is known in advance it can be easily parallelized in OpenMP, with the use of MPI to be characterized by a higher degree of complexity. It seems that the most effective way to parallelize the sweep function is to use OpenMP, either by itself or via a single MPI process, according to the decisions taken during the design of the application.

5.5. Parallelization of the Least Squares Fitting Operations

In the last stage of the serial FD3 algorithm, the function findMark is called to estimate the lower limit of the box size used in the least square fitting function fitLSqrLine and then, this function is called three times from the GetDims function, to estimate the capacity dimension (as the slope of the line for the arrays logEps and negLogBoxCountF), the information dimension (as the slope of the line for the arrays logEps and informationF) and the correlation dimension (as the slope of the line for the arrays logEps and logSumSqrFreqF). In this function, the logEps array contains all the available positive integers in the interval [1, numbits], even though, the fitLSqrLine does not use all these values but only the ones that belong in the interval [markF, numbits − 1], where markF is the value returned by the findMark function. Even though the three calls of the fitLSqrLine are independent each other and can run in parallel, the number of the loop iterations is small enough (with value equal to numbits – markF – 2) to justify the parallelization of the operations, at least via MPI, since the speedup of the process will be canceled by the message passing latency time (the same is true for the parallelization of the findMark function). However the parallelization via OpenMP probably will result to a small speedup, and therefore, it can be used in the parallel implementation of the FD3 application.

5.6. Putting all the Pieces Together

After the description of the most common options and opportunities for parallelization associated with the serial FD3 application, let us now present one of the candidate models that can be adopted to parallelize this program. This model is presented in **Figure 7** and is based to the MPI library, even though in some stages we can increase the degree of parallelization using OpenMP threads created by the MPI processes.

In the first step, the N processes of the application open collectively the input time series data file and each one of them reads its own part of data. The process

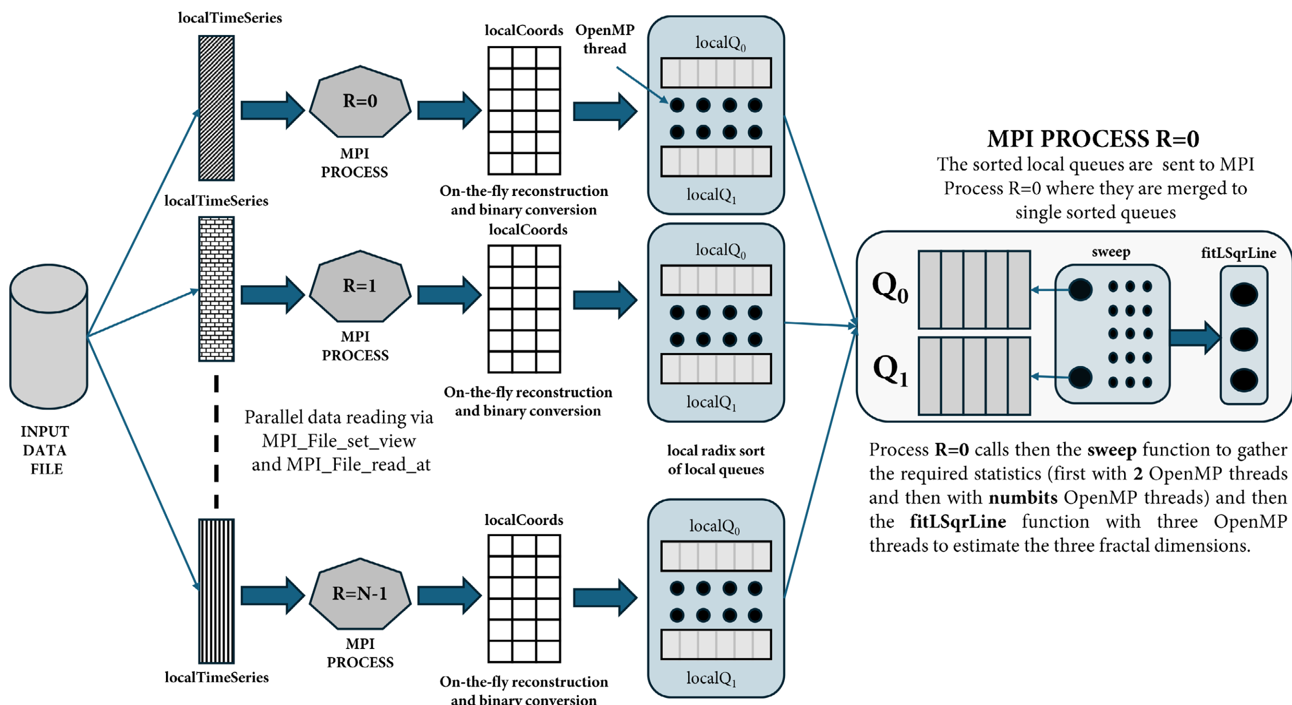


Figure 7. The proposed model for the parallel FD3 application.

number as well as the thread number together with the name of the time series data file, as well as the embedding dimension and the time delay are defined by the user from the command line during execution via a syntax in the form

mpirun -np <processes> <dataFile> <threads> <dimension> <tau>

To speedup the process, only the coords array is used, meaning that after the data retrieval operation and the initialization of the localTimeSeries array by each process, the reconstruction of the trajectory points and the conversion of their coordinates in binary numbers of 64-bit length, are performed by each process on the fly, leading thus to the initialization of the localCoords array that contains the binary coordinates of the trajectory points associated with each process. Then each process performs a local parallel radix sort on these data by creating the appropriate number of OpenMP threads to increase the degree of parallelism and in the next step all these sorted local queues are sent to the MPI process with a rank $R=0$ that merges them to the sorted queues Q_0 and Q_1. At this point, all the MPI processes are terminated, except the process $R=0$ that calls the sweep function and finally estimates the three fractal dimensions in the way described above. In other words, in the sweep function, two OpenMP threads are created to gather the required information from the sorted queues Q_0 and Q_1 and then a second group of numbits + 1 OpenMP threads, is also created, to parallelize the second loop of this function. Finally, in the GetDims function, the three independent calls of the fitLSqrLine can be executed in parallel by three OpenMP threads, to estimate the values of the capacity, information and correlation dimension. The procedure is completed.

6. Performance Evaluation of Parallel Application

Evaluating the performance of the parallel FD3 algorithm is complicated by a peculiarity that highlights the need for a theoretical performance model. Reliable speedup measurements require large datasets ($N \geq 10^6$) to ensure measurable serial execution times. However, excessively large datasets can introduce saturation effects ($N_B(\varepsilon) \approx N$), which degrade the accuracy of fractal dimension estimates, as noted by Sarraille and DiFalco. On the other hand, smaller datasets ($N \approx 10^5$) yield negligible serial execution times, casting doubt on the necessity of parallelization.

To address this, we propose: 1) using moderately sized datasets ($N = 10^5$ - 10^6) to balance computational effort and estimation reliability; 2) repeating executions to improve timing precision; 3) increasing the embedding dimension ($d = 4$ - 5) to raise serial computation time without requiring excessive N; 4) focusing speedup measurements on the most compute-intensive phases, such as radix sort and sweep; and 5) using synthetic datasets for performance evaluation while reserving smaller real-world datasets for accurate dimension estimation.

These strategies, guided by predictions from our theoretical model, enable robust speedup evaluation while preserving estimation fidelity—thus resolving the peculiarity without the need for extensive experimental validation.

6.1. Motivation for a Theoretical Performance Model

The peculiarity outlined above—namely, the conflict between the dataset size required for measurable serial execution times and the accuracy of fractal dimension estimates—motivates the construction of a theoretical model that predicts the performance of the parallel FD3 algorithm in the absence of empirical validation.

Rather than serving merely as a placeholder for missing benchmarks, the model offers a systematic framework for reasoning about performance under realistic architectural and algorithmic constraints. Specifically, it decomposes the total execution time $T_{\text{parallel}}(p,t)$ into analytically estimable components corresponding to each major computational phase:

- $T_{\text{IO}}(p)$: parallel data retrieval from disk,
- $T_{\text{recon}}(p)$: trajectory reconstruction and binary conversion,
- $T_{\text{radix}}(p,t)$: local radix sort and inter-process merging,
- $T_{\text{sweep}}(t)$: neighborhood counting over multiple scales,
- $T_{\text{lsqr}}(t)$: least-squares estimation of dimensions,
- $T_{\text{comm}}(p)$: total communication overhead.

Each term captures both computation and communication costs, parameterized by N (number of embedded points), d (embedding dimension), p (number of MPI processes), and t (threads per process). For example, the radix sort phase includes intra-node parallel sorting modeled as $T_{\text{radix,local}}(p,t) = (c_3 Nd \cdot 64)/(pt)$, and inter-process merging modeled as $T_{\text{comm,radix}}(p) = \alpha p + \beta (N/p)\log p$, where α and β denote message latency and inverse bandwidth, respectively.

Moreover, the model reflects realistic performance regimes:

- For small N, T_{serial} is negligible, making parallelization unjustified;
- For excessively large N, saturation effects distort $N_B(\varepsilon)$, reducing the accuracy of dimension estimation;
- For moderate N, both accurate dimension estimation and measurable speedup become attainable, justifying parallel deployment.

This analytical model, grounded in algorithmic complexity and qualitative agreement with past implementations, provides a robust tool for reasoning about performance, guiding implementation, and informing future empirical validation.

6.2. The Proposed Theoretical Performance Model

The proposed theoretical model for predicting the performance of the parallel FD3 algorithm, is presented below:

Serial Execution Time. The serial FD3 algorithm's execution time (T_{serial}) is dominated by:

- **Data Retrieval**: $\mathcal{O}(L)$, where L is the number of time series points.
- **Trajectory Reconstruction**: $\mathcal{O}(Nd)$, where $N \approx L - d\tau$, d is the embedding dimension, and τ is the time delay.
- **Binary Conversion**: $\mathcal{O}(Nd)$, converting d-dimensional points to 64-bit binary numbers.
- **Radix Sort**: $\mathcal{O}(Nd)$, sorting N points with d coordinates over 64 bits.
- **Sweep**: $\mathcal{O}(Nd + k)$, where $k = 32$ is the number of hypercube sizes.
- **Least-Squares Fitting**: $\mathcal{O}(k)$, negligible for large N.

Assuming $N \gg k$, the serial execution time is approximated as:

$$T_{\text{serial}} \approx c_1 L + c_2 Nd + c_3 Nd \cdot 64 \tag{17}$$

where c_1, c_2, c_3 are constants reflecting computational costs per operation.

Parallel Execution Time. The parallel execution time ($T_{\text{parallel}}(p,t)$) depends on p (MPI processes) and t (OpenMP threads per process). We model each phase:

- **Data Retrieval (Parallel I/O)**: Each process reads $\frac{L}{p}$ points:

$$T_{\text{IO}}(p) = \alpha_f + \frac{\beta_f L}{p} \tag{18}$$

This equation models the time for parallel data retrieval using MPI's collective I/O operations. Each of the p processes reads a subset of L/p points from the input file. The term α_f represents the fixed file system latency, which includes the time to open the file and initiate the read operation, independent of data size. The term $\beta_f L/p$ accounts for the data transfer time, where β_f is the inverse of the disk bandwidth per process (in seconds per data point), and L/p is the number of points read by each process. This assumes that the disk bandwidth scales with the number of processes up to the hardware's limit, and the total I/O time decreases as p increases, reflecting the parallel distribution of the workload.

- **Trajectory Reconstruction and Binary Conversion**: Each process handles $\frac{N}{p}$ points sequentially:

$$T_{\text{recon}}(p) = \frac{c_2 N d}{p} \tag{19}$$

The reconstruction and binary conversion phase involves processing N points, each with d coordinates, with a serial complexity of $\mathcal{O}(Nd)$. In the parallel implementation, each of the p MPI processes handles N/p points independently, performing both trajectory reconstruction (using embedding dimension d and time delay τ) and conversion of coordinates to 64-bit binary numbers. The constant c_2 encapsulates the computational cost per operation (e.g., arithmetic operations for reconstruction and bit manipulations for conversion). Dividing the total work by p reflects the even distribution of points across processes, assuming no inter-process communication during this phase, resulting in a linear reduction in execution time with increasing p.

- **Radix Sort**: Each process sorts $\frac{N}{p}$ points using t threads, followed by merging:

$$T_{\text{radix,local}}(p,t) = \frac{c_3 N d \cdot 64}{pt} \tag{20}$$

The local radix sort phase sorts N points, each with d 64-bit coordinates, with a serial complexity of $\mathcal{O}(Nd \cdot 64)$, where 64 is the number of bits processed per coordinate. In the parallel version, each of the p processes sorts N/p points, and within each process, t OpenMP threads further parallelize the work. The constant c_3 represents the cost per bit operation (e.g., comparisons and bucket assignments). The total work is divided by pt, reflecting the combined parallelism of p processes and t threads per process. This assumes perfect load balancing and negligible thread synchronization overhead within each process.

Communication for merging:

$$T_{\text{comm,radix}}(p) = \alpha p + \beta \frac{N}{p} \log p \tag{21}$$

After local sorting, the sorted sublists from each process are merged into a global sorted order using a binary tree merge strategy, incurring communication overhead. The term αp represents the total message latency, where α is the per-message latency (e.g., network setup time), and p messages are sent/received across processes in the merge tree. The term $\beta(N/p)\log p$ models the data transfer time, where β is the communication cost per byte (inverse bandwidth), N/p is the number of points per process, and $\log p$ arises from the number of levels in the binary tree merge (each level involves merging data between pairs of processes, halving the number of active processes per level). This logarithmic term reflects the hierarchical nature of the merge, increasing commu-

nication cost with more processes.

- **Sweep**: Two threads process queues Q_0 and Q_1 in the root process:

$$T_{\text{sweep}}(t) = \frac{c_4(Nd+k)}{t} + \alpha_t \log t,\ \ t \le 2 \tag{22}$$

The sweep phase scans the sorted queues Q_0 and Q_1 to compute statistics (e.g., number of hypercubes, point frequencies) across $k = 32$ hypercube sizes, with a serial complexity of $\mathcal{O}(Nd+k)$, where Nd accounts for processing N points with d coordinates, and k reflects the iterations over hypercube sizes. In the parallel implementation, this phase is executed by the root MPI process, which uses $t \le 2$ OpenMP threads to process the two queues independently (one thread per queue). The term $c_4(Nd+k)/t$ represents the computational work divided by the number of threads, where c_4 is the cost per operation (e.g., comparisons and array updates). The restriction $t \le 2$ arises because there are only two queues, limiting thread-level parallelism. The term $\alpha_t \log t$ accounts for thread synchronization overhead, where α_t is the synchronization latency (e.g., time for thread creation and barrier operations), and $\log t$ approximates the coordination cost, which grows logarithmically with the number of threads due to scheduling and memory access contention.

- **Least-Squares Fitting**: Three independent calls with $t \le 3$ threads:

$$T_{\text{lsqr}}(t) = \frac{c_5 k}{t} + \alpha_t \log t \tag{23}$$

The least-squares fitting phase involves three independent calls to compute the slopes of lines for the capacity, information, and correlation dimensions, each processing $k = 32$ hypercube sizes, with a serial complexity of $\mathcal{O}(k)$. In the parallel implementation, the root MPI process uses $t \le 3$ OpenMP threads, with each thread handling one of the three fitting operations. The term $c_5 k/t$ represents the computational work divided by the number of threads, where c_5 is the cost per operation (e.g., arithmetic operations for least-squares regression). The limit $t \le 3$ reflects the three independent tasks, constraining thread-level parallelism. The term $\alpha_t \log t$ models the thread synchronization overhead, where α_t is the synchronization latency, and $\log t$ captures the increasing coordination cost as more threads are used, due to factors like thread scheduling and shared memory access.

Therefore, the total parallel execution time is estimated as

$$\begin{aligned} T_{\text{parallel}}(p,t) = T_{\text{IO}}(p) + T_{\text{recon}}(p) + T_{\text{radix}}(p,t) \\ + T_{\text{sweep}}(t) + T_{\text{lsqr}}(t) + T_{\text{comm}}(p) \end{aligned} \tag{24}$$

where

$$T_{\text{radix}}(p,t) = T_{\text{radix,local}}(p,t) + T_{\text{comm,radix}}(p) \tag{25}$$

$$T_{\text{comm}}(p) = T_{\text{comm,radix}}(p) \tag{26}$$

Speedup. The speedup is defined as

$$S(p,t) = \frac{T_{\text{serial}}}{T_{\text{parallel}}(p,t)} \tag{27}$$

Substituting:

$$S(p,t) = \frac{c_1 L + c_2 Nd + c_3 Nd \cdot 64}{T_{\text{parallel}}(p,t)} \tag{28}$$

Efficiency. Efficiency is defined as

$$E(p,t) = \frac{S(p,t)}{pt} \tag{29}$$

$$E(p,t) = \frac{c_1 L + c_2 Nd + c_3 Nd \cdot 64}{pt \cdot T_{\text{parallel}}(p,t)} \tag{30}$$

Efficiency decreases with higher p and t due to communication and synchronization overheads. Optimizing t per process can mitigate this.

Practical Considerations.

- **Tuning p and t**: The model suggests a trade-off between processes and threads. Optimal values depend on N, d, and hardware parameters (α, β_f).
- **Impact of N**: Larger N amortizes fixed overheads, improving speedup.
- **Phase-Specific Optimization**: Radix sort and sweep dominate for large N. Reducing communication in radix sort merging or increasing t for local sorting can enhance performance.
- **Scalability Limits**: Network bandwidth and latency may limit scalability for large p, requiring careful tuning of communication patterns.
- **Validation**: Experimental results are needed to calibrate constants (c_i, α, β, α_t, β_f) and validate predictions.

7. Conclusions

The objective of this research is the parallelization of the FD3 application that can be used to estimate the values of the three well known fractal dimensions, namely, the capacity, the information and the correlation dimension from experimental time series data. The proposed model combines the use of the two most important parallel programming models, implemented by the MPI library and the OpenMP extension of the C language. The paper presents only the proposed parallel application and the theoretical model and does not give experimental results for the aforementioned reasons, but since the source code of the program is available to everyone, the implementation of the parallel application and its evaluation via the estimation of the speedup for different data sets is not considered a difficult task. In fact, this procedure can be considered as a future work for this research, together with the use of other parallel sorting algorithms instead radix sort, as well as the implementation of the parallel program using other parallel tools, such as the pthreads library and the CUDA programming environment. However, the proposed model provides a robust foundation for high-performance fractal dimension estimation, with applications in chaotic system analysis and computa-

tional physics.

Conflicts of Interest

The author declares no conflicts of interest regarding the publication of this paper.

References

[1] Russell, D.A., Hanson, J.D. and Ott, E. (1980) Dimension of Strange Attractors. *Physical Review Letters*, **45**, 1175-1178. https://doi.org/10.1103/physrevlett.45.1175

[2] Grassberger, P. and Procaccia, I. (1983) Characterization of Strange Attractors. *Physical Review Letters*, **50**, 346-349. https://doi.org/10.1103/physrevlett.50.346

[3] Farmer, J.D., Ott, E. and Yorke, J.A. (1983) The Dimension of Chaotic Attractors. *Physica D: Nonlinear Phenomena*, **7**, 153-180. https://doi.org/10.1016/0167-2789(83)90125-2

[4] Liebovitch, L.S. and Toth, T. (1989) A Fast Algorithm to Determine Fractal Dimensions by Box Counting. *Physics Letters A*, **141**, 386-390. https://doi.org/10.1016/0375-9601(89)90854-2

[5] Grassberger, P. (1983) On the Fractal Dimension of the Henon Attractor. *Physics Letters A*, **97**, 224-226. https://doi.org/10.1016/0375-9601(83)90752-1

[6] Barnsley, M. (1988) Fractals Everywhere. 2nd Edition, Academic Press.

[7] Hunt, F. and Sullivan, F. (1986) Efficient Algorithms for Computing Fractal Dimensions. In: Mayer-Kress, G., Ed., *Dimensions and Entropies in Chaotic Systems: Quantification of Complex Behavior*, Volume 32, Springer, 74-81. https://doi.org/10.1007/978-3-642-71001-8_10

[8] Giorgilli, A., Casati, D., Sironi, L. and Galgani, L. (1986) An Efficient Procedure to Compute Fractal Dimensions by Box Counting. *Physics Letters A*, **115**, 202-206. https://doi.org/10.1016/0375-9601(86)90465-2

[9] Theiler, J. (1987) Efficient Algorithm for Estimating the Correlation Dimension from a Set of Discrete Points. *Physical Review A*, **36**, 4456-4462. https://doi.org/10.1103/physreva.36.4456

[10] Sarraille, J.J. and Myers, L.S. (1994) FD3: A Program for Measuring Fractal Dimension. *Educational and Psychological Measurement*, **54**, 94-97. https://doi.org/10.1177/0013164494054001010

[11] Knuth, D.E. (1988) The Art of Computer Programming. Volume 3, Addison-Wesley.

[12] Gropp, W., Lusk, E., Doss, N. and Skjellum, A. (1996) A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, **22**, 789-828. https://doi.org/10.1016/0167-8191(96)00024-5

[13] Quinn, M.J. (2003) Parallel Programming in C with MPI and OpenMP. McGraw-Hill.