Efficient Encoding and Decoding Extended Geocodes for Massive Point Cloud Data

Taehoon Kim, Kyoung-Sook Kim, Jun Lee, Akiyoshi Matono and Ki-Joune Li

March 6, 2019

Efficient Encoding and Decoding Extended Geocodes for Massive Point Cloud Data

Taehoon Kim*†, Kyoung-Sook Kim*‡, Jun Lee*, Akiyoshi Matono*, Ki-Joune Li†
* Artificial Intelligence Research Center (AIRC), National Institute of Advanced Industrial Science and Technology (AIST)
Tokyo, Japan
{kim.taehoon, ks.kim, jun.lee, a.matono}@aist.go.jp
† Department of Computer Science, Pusan National University
Pusan, South Korea
{taehoon.kim, lik}@pnu.edu

***Abstract*—With the development of mobile surveying and mapping technologies, point cloud data has been emerging in a variety of applications including robot navigation, self-driving drones/vehicles, and three-dimensional (3D) urban space modeling. In addition, there is an increasing demand for the database management system to share and reuse point cloud data, unlike being treated as archive files in the traditional uses and applications. However, database scalability needs to be explored to process and manage a massive volume of point cloud data defined by a 3D (X, Y, and Z) coordinates system. The typical approach to handle big data and distribute it across multiple nodes is data partitioning. Geohashing is a popular way to convert a latitude/longitude spatial point into a code/string and has used for storing data into buckets of the grid. Many methods of handling big geospatial data, especially NoSQL databases, are based on the geohashing techniques. In this paper, we propose an efficient method to encode/decode 3D point cloud in a Discrete Global Grid System (DGGS) that represents the Earth as hierarchical sequences of equal area/volume tessellations, similar to geohash. The current geohash of base36 has the difficulties of working with high-resolution 3D point clouds for data storage, filter, integration, and analytics because of its limitation of cell size and unequal areas. We employ DGGS-based Morton codes with more than 64 bits for precise 3D coordinates of point cloud and compare the encoding/decoding performance between two implementations: using strings and using the combination of bit interleaving and lookup tables.**

***Index Terms*—Big data management, 3D point cloud, Discrete Global Grid System, Morton order**

I. Introduction

Big data has been a major topic in recent years to create value from data beyond the ability of typical database systems [1]. In geospatial research communities, big data has been discussed to capture, store, manage and analyze real-world phenomena from huge volumes of two-dimensional geospatial and spatio-temporal data. Not only data representation and integration, but also query processing have had to deal successfully with new challenges on volume, velocity, variety, or veracity. Parallel processing techniques based on data partitioning increases the efficiencies of geospatial operations across nodes in a cluster [2]. Geohash is the most popular method of expressing and indexing a location using a key generated by a Space Filling Curve (SFC), partitioning a domain into several grids. It is naturally adapted to harness big geospatial data [3]. However, the area/volume of the partitioned grids by geohash is not the same. These irregular sizes of the grid can be a problem of the projection and the seamless integration from various geospatial data sources. In order to solve the problem, the Discrete Grid Global System (DGGS) has recently been proposed for geo-referencing [4]. DGGS divides the Earths surface into grids with multiple levels of granularity and generates a code per each grid through a SFC [5], [6]. The Morton order has been commonly used for addressing each grid and access the geospatial data onto the grids in Geographic Information Systems (GIS) [7]–[13]. However, existing studies have limited the maximum size of the Morton code to within the number of bits of the CPU in order to process the conversion between the Morton code and the coordinates within a constant time $\mathcal{O}(1)$; e.g., when using the 64-bit CPU, the maximum size of the code is 64 bits.

In this study, we propose a method to effectively encode/decode Morton codes with sizes over 64 bits for high-resolution 3D point clouds. A point cloud is a collection of points defined by a given coordinates system to represent a 3D shape or feature with X, Y and Z coordinates and additional attributes such as intensity and angle. Point clouds are most often created by methods used in photogrammetry or remote sensing. The combination of Light Detection And Ranging (LiDAR) sensors and positioning systems like a Global Positioning System (GPS) are the most common instruments used to collect point cloud data to capture the real world. The point cloud is recently emerging as a fundamental data to make high definition maps [14]; however, the tremendous volume of the point data makes difficult to handle. An example of data size is about 11 TB of data in the Netherlands (about $40,000km^2$) area, although it varies depending on the detail of the data and type of LiDAR sensors [15]. It is almost 12,000 times the data capacity of OpenStreetMap, which is a two-dimensional vector map service, in the same area. Moreover, their relative or local coordinate systems depending on locations or sensors obstruct to process and integrate the point cloud data from various sources seamlessly.

We put forward Morton codes for large volumes of 3D point cloud data based on the DGGS. However, a point in

‡corresponding author

a cloud generated by LiDAR requires a very high-resolution Morton code. If we transform 3D coordinates of points into 64-bit Morton codes, we can use only up to 21 bits for each coordinate. This is insufficient to express LiDAR data because of the limitation of identification. For example, we need 96 bits of Morton code, assigning 32 bits for each coordinate representation, to enable the identification within $1\,mm^3$ scale of a grid. The encode/decode method we present enhances the resolution of location and improves the processing performance by using the combination of bit interleaving and lookup tables.

The rest of this paper is organized as follows. Section II briefly addresses the DGGS for reference and the SFC code. In Section III, we present how to handle 3D point cloud data with the DGGS and Morton order. Section IV describes the encode/decode methods using strings and bits lookup tables. We compare the performance of the two methods through experiments with various conditions in Section V. Finally, we conclude this paper with is concluded in Section VI.

II. Related Works

A. Discrete Global Grid System (DGGS)

Discrete Grid Global System (DGGS) is a spatial reference system that describes a position in the real world by a datum [4]. Two geospatial standard organizations of the Open Geo Consortium (OGC) and the International Organization for Standardization Technical Committees (ISO/TC) 211 are promoting DGGS as a new framework for coordinate conversion from an ellipsoidal coordinate system to a plane. A DGGS uses a hierarchical tessellation of cells to partition the surface of the Earth with equal areas/volumes for a grid based global spatial information framework to map data at a location on the map. The main objectives are as follows.

Minimize distortion: In a grid based global framework, it is important how to construct structure grid cells over the entire surface of the Earth. For example, geohash is a hierarchical spatial data structure where space is divided into rectangular cells, each with varying degrees of the area and/or shape distortion. Even though it contributes to the easy management and faster processing in a spatial database, geohash is difficult to guarantee accurate geospatial measurements like distance and area. Namely, the coordinate conversion and transformation should be carefully considered. The concept of DGGS is designed to solve this problem by dividing the surface of the earth with the equality of area and shape.

Data integration: The goal of the DGGS is to be able to seamlessly integrate spatial data from multiple sources and types in any location (e.g., Latitude/Longitude) at any scale by referring to a single reference framework consisting of tessellations; e.g., tetrahedrons, cubes, and octahedrons to model the earth. With a DGGS, we can consistently replicate the result of spatial analysis anywhere on the Earth. In addition, it provides statistically valid summaries based on any chosen selection of cells by optimizing the loss of geospatial data during the projection process.

Efficient Data analysis and visualization of big data: DGGS has a set of functional algorithms that enable rapid data analysis for very large numbers of cells and are inherently suited for parallel processing. It is also designed to more efficiently use CPU and GPU resources to parallel and distributed processing for geographically relevant data such as language, environment, and climate. Also, its hierarchical structure is suitable for the level of detail representation for visualization.

B. Space Filling Curves

A space-filling curve is a continuous mapping from an n-dimensional space into a list of unique codes (or one-dimensional range). There are many existing studies on the data management of point cloud through SFC codes and they can be classified into three types depending on the representation of code: constant type, string type, and others. The constant type represents an integer as a code [8]–[12]. Even though it is a common implementation in GIS, the maximum SFC code size must be less than or equal to the number of bits of the CPU. Psomadaki [10] used the NUMBER type of an Oracle database that supports 128 bit to solve this problem, but this is not a fundamental solution. Next, the string type is designed to overcome the restriction on the size of the SFC code [7], [16]. However, its encoding/decoding performance is lower than the constant-type code. In [17], Pavlovic proposed a compression method for point cloud data using the SFC. The method is based on a static space and a distribution of coordinates for input point cloud data. If the point cloud is dynamically inserted and integrated into the entire globe, the performance issue will likely arise. In this study, we consider a string-type SFC code based on the Morton curve to express point clouds on a global coordinate system.

Morton (or Z-order) code is a familiar SFC ordering method under the consideration of the data locality [18], [19]. It generates adjacent Morton codes to be spatially close to each other with a hierarchical structure. For example, a three-dimensional Morton code with a resolution of 1 is shown in Fig. 1. As seen in Fig. 1, Morton code can be computed by a simple bit-interleaving. The parameters needed to generate the Morton code are as follows: n-dimensional coordinates, length of Morton code (resolution). For example, for a three-dimensional point p, the x-coordinate can be expressed in binary form as $p^*_x = x_5x_4x_3x_2x_1x_0$. Similarly, the y and z coordinates can be expressed as $p^*_y = y_5y_4y_3y_2y_1y_0$, $p^*_z = z_5z_4z_3z_2z_1z_0$. At this time, the 18-bit Morton code is represented by bit-interleaving as follows:

$$m(p^*) = z_5y_5x_5z_4y_4x_4z_3y_3x_3z_2y_2x_2z_1y_1x_1z_0y_0x_0$$

III. DGGS for 3D Point Cloud

First of all, we describe the 3D DGGS method introduced in [16]. The method is designed for constructing a DGGS with respect to location, time, and densities of point cloud data.

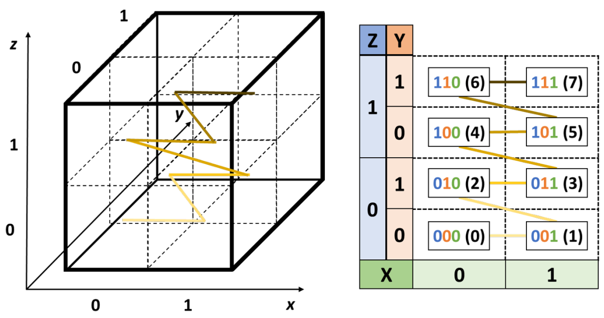


Fig. 1. Example of Morton code generation in a 3D space

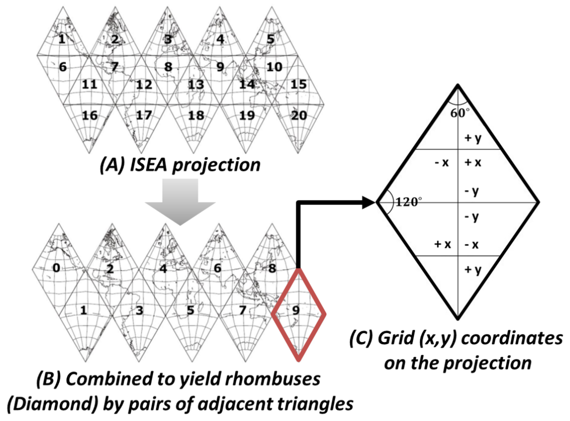
For the sake of simplicity, we call *P-code* the 3D DGGS code generated by the following processes:

1) Firstly, the (continuous) precision of a point in a standard map projection should be calculated. The precision associated with the point is linked to the closest discrete resolution of a DGGS.
2) Next, the point is converted from the initial CRS into geographic coordinates on the WGS84 ellipsoid, with the precision of the original CRS.
3) To compute the Morton code for use in a DGGS, the global indexing framework, the point is projected onto the Icosahedral Snyder Equal Area (ISEA) projection through the following steps:
   a) Find the icosahedral triangle face number on which the point is located as shown in Fig. 2 (A).
   b) Compute the grid coordinates originating from the geographic center of that face. At this time, the geographical coordinates of all triangular centers and the necessary equations to calculate are given from [20].
   c) Then, The points rhombus face number is then determined based upon its triangle face number as shown in Fig. 2 (B).
   d) Finally, the coordinates of the point are converted to the left vertices of the included rhombus, which is the origin of the Morton SFC for every face. Since the interior angle of a rhombus at the left vertex is 120° as shown in Fig. 2 (C), its X and Y axis also skewed with an interior angle. The equations to convert an coordinate (x, y) from the previous step into those of a skewed system on a rhombus are as follow:

$$X_{skewed} = x - \frac{y}{\sqrt{3}} \tag{1}$$

$$Y_{skewed} = x + \frac{y}{\sqrt{3}} \tag{2}$$

4) The x and y coordinate from rhombus are converted into binary form with the same number of bits as the DGGS resolution, through the following steps:



(A) ISEA projection
(B) Combined to yield rhombuses (Diamond) by pairs of adjacent triangles
(C) Grid (x,y) coordinates on the projection
60°
120°

Fig. 2. DGGS Reference Frame

   a) If a point has been assigned to a cell at resolution r, $X_{bin}\,(and\, Y_{bin}, Z_{bin})$ is defined as below:

$$X_{bin} = b_{r-1}b_{r-2}...b_1b_0 \tag{3}$$

   Note that b is a binary number.

   b) These binary coordinates are converted into integers. Then, the below formula can be used to generate a Morton code M_C for the point in 2D [21] as shown in Fig. 3:

$$M_C = 2 * (Y_{bin}) + (X_{bin}) \tag{4}$$

   And this formula can easily be extended into 3D:

$$M_C = 4 * (Z_{bin}) + 2 * (Y_{bin}) + (X_{bin}) \tag{5}$$

   c) Lastly, a P-code is made by the combination of the rhombus face number and the Morton code M_C. The header of the P-code is face number and the tail of the P-code is the Morton code.

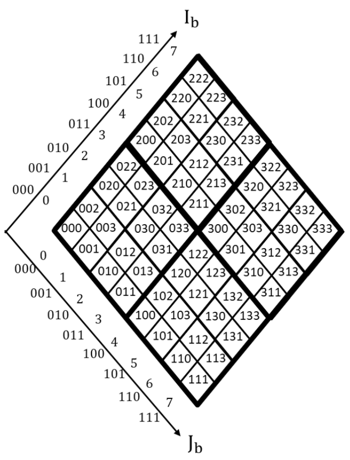


Fig. 3. Example of M_C for 2D coordinates with three resolution. Figure from Zhao et al.,2006 [21]

In this way, we can generate the P-code, the Morton code for handling the point cloud data in DGGS. However, when performing a conversion process on large point data, each process should be performed as quickly as possible, even if it is a simple process. In this paper, we propose a more efficient method for generating Morton code than the previously proposed method.

IV. Encoding/Decoding with P-code

In this section, we describe two methods to encode/decode P-code. Firstly, we show how to convert through the string handling which is intuitively implemented in the proposed method from the previous section. After that, we suggest how to convert using bit operation and lookup tables.

A. Encoding with P-code

There are two ways to encode the P-code;a method using string handling, a method using bit operation. Both methods have identical inputs and outputs; Inputs are the integer coordinates in the rhombus, the rhombus index f and the resolution of the DGGS, r, which are derived from the three-dimensional point cloud coordinates, as described in Section III. The output is a P-code corresponds to the input data. The flowchart of the encoding method is the same as Fig. 4. Commonly, the face index is attached to the front of the final generated P-code. Then, the string handling method is a somewhat intuitive implementation of the method described in Section III. The process of generating P-code using string handling is represented by Algorithm 1.

Algorithm 1 Encode P-code with string handling

```
Input:
- $x$: 32-bit integer coordinate on the X axis
- $y$: 32-bit integer coordinate on the Y axis
- $z$: 32-bit integer coordinate on the Z axis
- $r$: resolution of the DGGS
- $f$: index of the rhombus to which the coordinates belong
Output:
- $pcode$: DGGS based Morton code for point clouds
Begin
 1: $bin_X \leftarrow$ convert x coordinate to binary string
 2: $bin_Y \leftarrow$ convert y coordinate to binary string
 3: $bin_Z \leftarrow$ convert z coordinate to binary string
 4: Make the lengths of $bin_X, bin_Y$, and $bin_Z$ are same with
    $r$ using zero fill
 5: for $i = 0$ to $r$ do
 6:     $int_X \leftarrow$ get a $i$-th character of $bin_X$ as integer
 7:     $int_Y \leftarrow$ get a $i$-th character of $bin_Y$ as integer
 8:     $int_Z \leftarrow$ get a $i$-th character of $bin_Z$ as integer
 9:     $pcode_{part} \leftarrow int_X + int_Y * 2 + int_Z * 4$
10:     $pcode \leftarrow$ Add $pcode_{part}$ behind $pcode$
11: end for
12: Add a $f$ number at front of $pcode$
13: return $pcode$
End
```

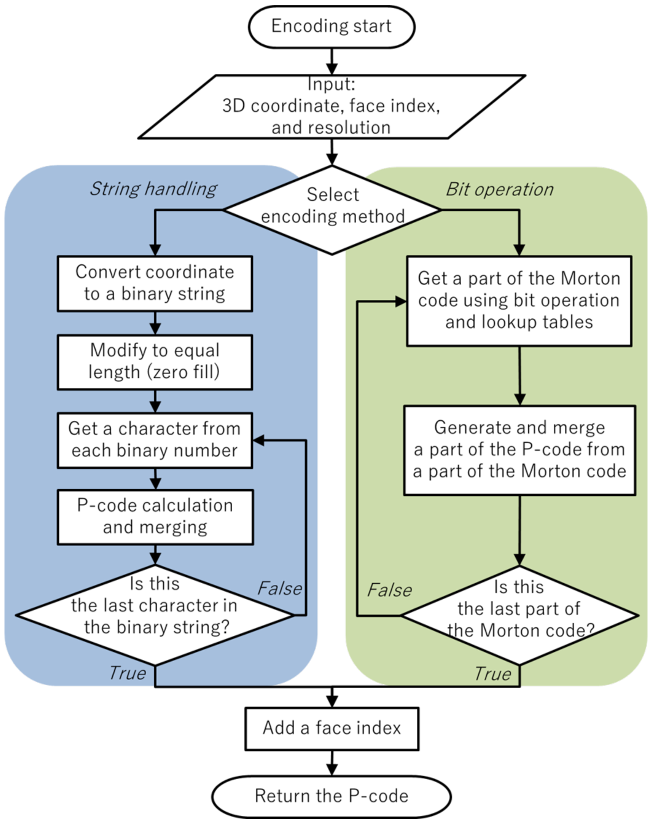


Fig. 4. Flow chart of P-code encoding processing

Algorithm 1 generates and merges one character at a time to generate a P-code string having a length of resolution (r). To do this, it is necessary to convert the input coordinates (x, y, z) into binary numbers and then making the length of P-code is equal to r. If the length of P-code is less than r, fill in the character '0' as much as the remaining length. On the other hand, if the length if P-code is greater than r, only substrings from the beginning to r are used. Finally, a P-code is generated by adding a rhombus index f to the front of generated $pcode$. The time complexity of Algorithm 1 is determined by the resolution, so it becomes $\mathcal{O}(r)$.

Next is how to encode P-code using bit operation. We had previously studied how to effectively encode/decode Morton codes using bit operations. However, the suggested method in Section III cannot use the existing method with the following issues; If the output resolution r is greater than 21, the size of the generated Morton code exceeds 64 bits. The existing method [8], [9], [11] has a limitation that the size of the generated Morton code cannot exceed 64 bits when 64-bit OS is used. In order to solve this limitation, we propose a method of merging the results after processing the existing Morton code into several parts, like divide and conquer.

The P-code generation process using bit operation is the same as Algorithm 2. Simply fetching the required value using only the bit operation doesn't reduce the time complexity of the algorithm. For more efficient processing, a fixed-size value must be processed at one time. In Algorithm 2, the encoding is performed through a lookup table that pre-computed encoding results in a 9-bit input object. The reason for choosing 9-bit are as follows:

1) When generating a Morton code for three-dimensional coordinates, the values are independently determined by 3-bit units.
2) If the size of the input object of the lookup table is n-bit, the size of the Morton code for the three-dimensional coordinates is $((n-1)*3+1)$ bits.
3) If the size of the input object of the lookup table is n-bit and the size of the element in the lookup table is 32 bits, then the required storage space is proportional to two to the power of n, as in the following formula: required storage space = $(2^b * 32)$ bits.
4) As the size of the input object of the lookup table increases, the number of operations does not decrease unconditionally.

In conclusion, when the 32-bit data structure is used as an element, the maximum size of the input object of the lookup table is 9-bit.

To generate P-code, two lookup tables are required; MT table for converting integers to Morton code, and DT table for converting Morton code to P-code. Algorithm 2 generates Morton code using the MT table for the input integer coordinate value and then generates P-code using the DT table with the generated Morton code. However, since the entire Morton code can't be generated immediately through the MT table, it has to be divided into 9-bit units. Since the input integer coordinate value is maximum of 32 bits, it must be repeated 4 times when processing in the 9-bit unit. Also, $mcode_{part}$, the part of Morton code generated by the MT table, it is maximum 27 bits, so it must be repeated 3 times in 9-bit units. All processes are processed in constant time ($\mathcal{O}(1)$). The P-code is generated by taking a substring of resolution r from the last generated $pcode$ and prefixing it with the rhombus index f. The time complexity of Algorithm 2 is $\mathcal{O}(12) \simeq \mathcal{O}(1)$.

Algorithm 2 Encode P-code with bit operation

```
Input:
- $x$: 32-bit integer coordinate on the X axis
- $y$: 32-bit integer coordinate on the Y axis
- $z$: 32-bit integer coordinate on the Z axis
- $r$: resolution of the DGGS
- $f$: index of the rhombus to which the coordinates belong
Output:
- $pcode$: DGGS based Morton code for point clouds
Begin
1:  $MT \leftarrow$ pre-computed 9-bit lookup table for encoding integer to Morton code
2:  $DT \leftarrow$ pre-computed 9-bit lookup table for encoding Morton code to P-code
3:  $bitmask = $ 0x1FF {9 bits mask}
4:  for $i = 4$ to $1$ do
5:      $shift = (i-1)*9$
6:      $mcode_{part} \leftarrow MT[(z>>shift) \wedge bitmask] >> 2$
                          $\vee MT[(y>>shift) \wedge bitmask] >> 1$
                          $\vee MT[(x>>shift) \wedge bitmask]$
7:      for $j = 3$ to $1$ do
8:          $shift = (j-1)*9$
9:          $pcode_{part} \leftarrow DT[(mcode_{part}>>shift) \wedge bitmask]$
10:         $pcode \leftarrow$ Add $pcode_{part}$ behind $pcode$
11:     end for
12: end for
13: $pcode \leftarrow pcode$ substring from header to $r$ length
14: Add a $f$ number at front of $pcode$
15: return $pcode$
End
```

B. Decoding with P-code

There are two ways to decode P-codes, same as encode; a method using string handling, a method using bit operation. Both methods have identical inputs and outputs; The input is the P-code, and the output is the three-dimensional coordinate corresponding to the input data. The flowchart of the decode method is the same as Fig. 5. Basically, both method processes in the reverse order of encoding. Also, a face index is commonly generated by taking one character from the front of P-code.

The String handling-based decoding method is the same as Algorithm 3. Algorithm 3 is processed as follows:

1) Create a variable $[x, y, z]$ for storing the three-dimensional coordinate values as binary numbers.
2) Take one character from beginning to end of the P-code ($pcode_i$), then add the coordinate values to $pcode_i$ to $[x, y, z]$; For example, if $pcode_i$ is '5', add ['1','0','1'] to $[x, y, z]$.
3) Finally, converts the coordinate values generated by binary numbers to integers and return these coordinates and the face index.

Algorithm 3 uses three **If** conditional statements to determine the value of $[x, y, z]$, but this can be replaced by one **Switch** conditional statement with eight different cases. As a result, the time complexity of the String handling-based decoding method is $\mathcal{O}(r)$ because a simple statement is repeated as much as the length of the P-code.

The Bit operation-based decoding method is the same as Algorithm 4. Similar to the encoding process, two lookup tables are required for decoding; DT_{de} table to convert P-code to Morton code, MT_{de} table to convert Morton code to integer coordinate. Algorithm 4 is processed as follows and the time complexity of the Bit operation-based decoding method is $\mathcal{O}(r/3)$.

1) To make the length of the input P-code is a multiple of 3, filling the '0' character at the head of the input P-code.
2) Obtains a substring of length 3 from the tail of the P-code.
3) Using the DT_{de} table, get the Morton code corresponding to the obtained substring from the previous step.

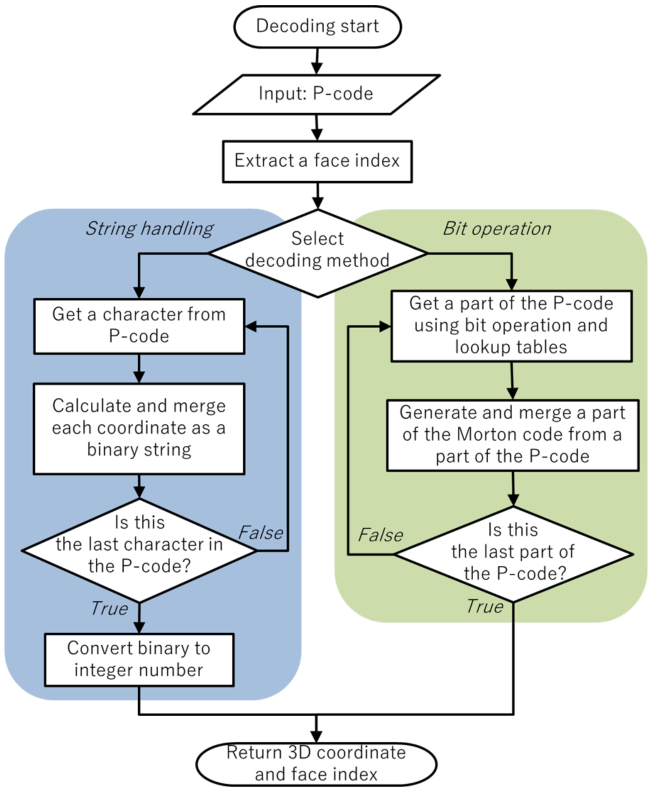

Decoding start
Input: P-code
Extract a face index
String handling
Bit operation
Select decoding method
Get a character from P-code
Calculate and merge each coordinate as a binary string
Is this the last character in the P-code?
False
True
Convert binary to integer number
Get a part of the P-code using bit operation and lookup tables
Generate and merge a part of the Morton code from a part of the P-code
Is this the last part of the P-code?
False
True
Return 3D coordinate and face index

Fig. 5. Flow chart of P-code decoding processing

4) Using the MT_{de} table, get the corresponding coordinates $[x, y, z]$ for the obtained Morton code from the previous step.
5) Since the coordinate value generated by step 4 is a partial value, the actual value is calculated using *SHIFT* and *OR* operation.
6) Return the final calculated coordinate values and face index together.

V. Evaluation

In this section, we introduce the results of evaluating the performance of implemented functions. Firstly, we show the benchmark results evaluated using Java Microbenchmark Harness (JMH). After that, performance is evaluated according to the number of points and resolution using point cloud data. Finally, we evaluate the performance according to the number of cores of Spark.

A. Experiment environments

The experimental environment is the same as Table I. All functions are implemented using Java. The point cloud data used in the experiment was taken from Shizuoka Point Cloud DB[1].

[1]https://pointcloud.pref.shizuoka.jp/

Algorithm 3 Decode P-code with string handling

```
Input:
- $pcode$: DGGS based Morton code for point clouds
Output:
- $coord$: 3-dimensional coordinate constituting the P-code
- $f$: index of the rhombus to which the coordinates belong
Begin
1:  $f \leftarrow$ extract one character from the head of $pcode$
2:  $r \leftarrow$ the length of $pcode$
3:  for $i = 0$ to $r$ do
4:      $pcode_i \leftarrow i$-th character of $pcode$
5:      if $pcode_i \wedge$ 0x01 = 0x01 then
6:          $coord_X \leftarrow$ add '1' as a character
7:      end if
8:      if $pcode_i \wedge$ 0x02 = 0x02 then
9:          $coord_Y \leftarrow$ add '1' as a character
10:     end if
11:     if $pcode_i \wedge$ 0x04 = 0x04 then
12:         $coord_Z \leftarrow$ add '1' as a character
13:     end if
14:     $coord \leftarrow$ convert $coord$ to integer
15: end for
16: return $coord$ and $f$
End
```

Algorithm 4 Decode P-code with bit operation

```
Input:
- $pcode$: DGGS based Morton code for point clouds
Output:
- $coord$: 3-dimensional coordinate constituting the P-code
- $f$: index of the rhombus to which the coordinates belong
Begin
1:  $DT_{de} \leftarrow$ pre-computed 9-bit lookup table for decoding
    Morton code from P-code
2:  $MT_{de} \leftarrow$ pre-computed 9-bit lookup table for decoding
    coordinate from Morton code
3:  $f \leftarrow$ extract one character from the head of $pcode$
4:  while the length of $pcode$ isn't a multiple of 3 do
5:      $pcode \leftarrow$ Add '0' to the head of $pcode$
6:  end while
7:  $cnt \leftarrow$ the result that length of $pcode$ divided by 3
8:  for $i = 0$ to $cnt$ do
9:      $i_S = (cnt - (i + 1)) * 3$
10:     $i_E = i_S + 3$
11:     $pcode_{part} \leftarrow$ a part of $pcode$ between $i_S$ and $i_E$ index
12:     $idx_M \leftarrow DT_{de}.get(pcode_{part})$
13:     $coord_X \leftarrow coord_X \vee (MT_{de}[idx_M]<<(3 * i))$
14:     $coord_Y \leftarrow coord_Y \vee (MT_{de}[idx_M>>1]<<(3 * i))$
15:     $coord_Z \leftarrow coord_Z \vee (MT_{de}[idx_M>>2]<<(3 * i))$
16: end for
17: return $coord$ and $f$
End
```

TABLE I
EXPERIMENT ENVIRONMENTS

feature	information
OS	Windows 10 Pro
CPU	Intel(R) Core i7-8700 CPU duo, 3.19 GHz
Memory	64 GB, 2666 MHz
Java version	1.8
JMH version	1.21
Spark version	2.3.0

TABLE II
BENCHMARK RESULTS OF EACH ENCODE/DECODE FUNCTIONS

function	mode	cnt	score & error	units
decodeWithBitOperation	avgt	25	0.324 ± 0.027	us/op
decodeWithString	avgt	25	1.387 ± 0.110	us/op
encodeWithBitOperation	avgt	25	1.363 ± 0.088	us/op
encodeWithString	avgt	25	2.073 ± 0.088	us/op

B. Benchmark

We benchmarked each function introduced in Section IV. Benchmarks were performed using JMH [2] provided by OpenJDK. The benchmark result is the same as Table II. *function* is the name of the function to be benchmarked. *mode* is the mode used to perform the benchmark. In this paper, we set the mode to average time. *cnt* means the number of iterations. In this paper, it is set to repeat 5 times using 5 threads. *score&error* means evaluation result according to mode. *units* means the time unit of the evaluation. In this paper, we used a microsecond unit.

C. Experiment with point cloud data

Fig. 6 shows the time to perform encoding/decoding on the number of points. We measured the runtime by varying the points from one million to eight million. For encoding, the target resolution is set to 32, the maximum resolution. The final run time is the average time of the same function 50 iterations. In all cases, we found that the Bit operation-based method is faster than the String handling-based method. Similar to the benchmark results shown in the previous section, the Bit operation-based method is faster than the String handling-based method by about two times, and the decoding is about four times faster than the String handling-based method. The reason for this result is confirmed by the difference in the time complexity of the algorithms. In the case of encoding, the time complexity of the string handling method is $\mathcal{O}(r)$. On the other hand, the time complexity of the bit operation method is $\mathcal{O}(1)$. Also, similarly, In the case of decoding, the time complexity of the string handling method is $\mathcal{O}(r)$, but the time complexity of the bit operation method is $\mathcal{O}(r/3)$.

Fig. 7 is the result of encoding/decoding by parallelism using the Apache Spark[3]. The number of spark cores was changed from 1 to 4 and the conversion execution time was compared. The final execution time was taken as the average value of 20 repeated experiments. The legend is structured as follows; Number of points entered - used method. Experimental results show that the number of spark cores and the execution time are almost inversely proportional.

Fig. 8 shows the change in execution time, according to the encoding resolution. For String handling-based encoding, the time required is linearly increased as the resolution is increased, but it is almost the same for Bit operation-based encoding. The reason for this is that the time complexity of the string handling encoding is determined by the resolution; $\mathcal{O}(r)$. On the other hand, the time complexity of the bit operation encoding is $\mathcal{O}(1)$.

VI. CONCLUSION

With the recent increase in the volume of 3D point cloud data produced, a method for managing the point cloud data becomes necessary. To manage point cloud data for the entire globe, a Morton code for 3D (and 4D) point cloud in DGGS has been proposed and is called P-code. However, for high precision, the code size can exceed 64 bits, which can cause performance problems. In this work, we proposed an efficient encoding/decoding method for high-precision point cloud data. To this end, we contributed the following points:

- Proposed methods to encode/decode high-resolution Morton code using the combination of bit interleaving and lookup tables.
- As a result of performance evaluation, encoding performance gains of up to 2x and decoding performance gains of up to 4x compared with String handling-based method.

However, this paper has several limitations. Firstly, the proposed methods are focused on only the 3D coordinate. But, P-code covered higher dimension;e.g., 3D coordinate with time. And we need methods for analyzing data, but P-code cannot be directly used to analyzing or computation. However, for managing and processing the massive volume of point cloud data, it will be necessary to solve these issues.

ACKNOWLEDGMENT

This work was partially supported by the New Energy and Industrial Technology Development Organization (NEDO).

REFERENCES

[1] R. V. Zicari, "Big data: Challenges and opportunities," *Big data computing*, vol. 1, pp. 103–128, 2014.

[2] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding, "Data mining with big data," *IEEE transactions on knowledge and data engineering*, vol. 26, no. 1, pp. 97–107, 2014.

[3] K. Lee, R. K. Ganti, M. Srivatsa, and L. Liu, "Efficient spatial query processing for big data," in *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2014, pp. 469–472.

[4] M. B. Purss, R. Gibb, F. Samavati, P. Peterson, and J. Ben, "The ogc® discrete global grid system core standard: A framework for rapid geospatial integration," in *Geoscience and Remote Sensing Symposium (IGARSS), 2016 IEEE International*. IEEE, 2016, pp. 3610–3613.

[5] X. Wang and A. Gruen, "A hybrid gis for 3-d city models," in *XIXth Congress, International Society for Photogrammetry and Remote Sensing (ISPRS 2000)*, vol. 33, no. B4/3. Gropher Publ., 2000, pp. 1165–1172.

[6] J. Bai, X. Zhao, and J. Chen, "Indexing of the discrete global grid using linear quadtree," in *Proceedings of ISPRS Workshop on Service and Application of Spatial Data Infrastructure*, 2005, pp. 267–270.

[2]http://openjdk.java.net/projects/code-tools/jmh/

[3]Lightning-fast unified analytics engine: https://spark.apache.org

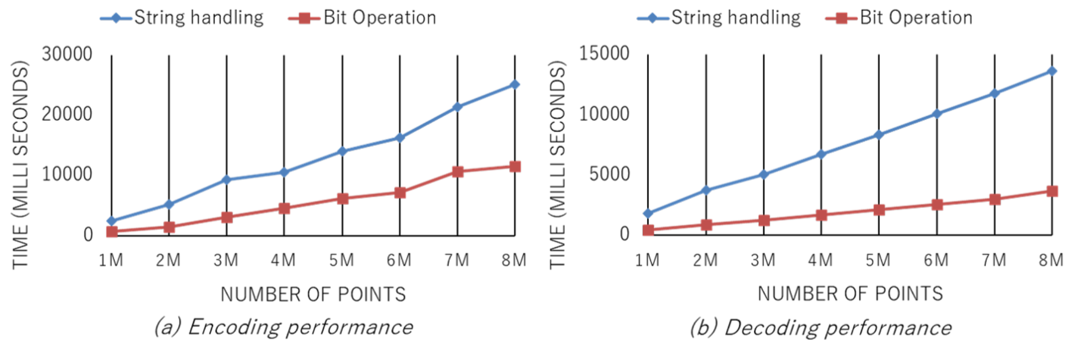


Fig. 6. Performance evaluation with point cloud data: (a) performance of encoding methods, (b) performance of decoding methods

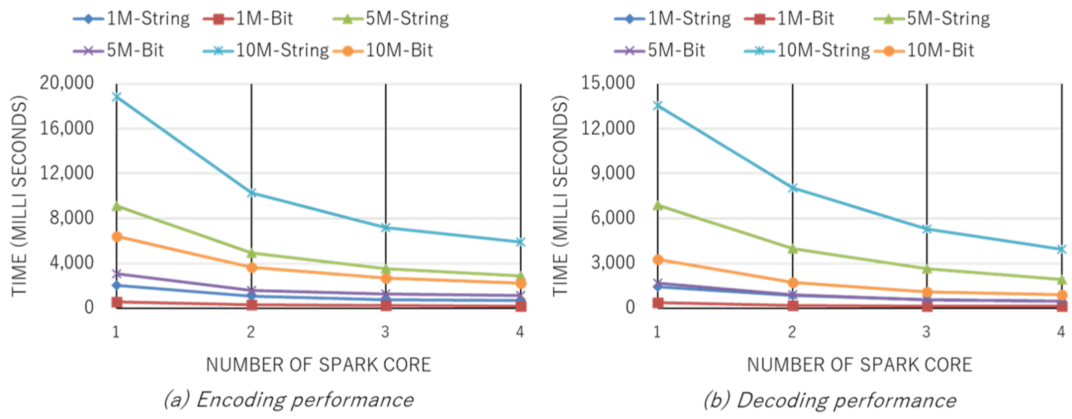


Fig. 7. Performance evaluation with Spark: (a) performance of encoding methods, (b) performance of decoding methods

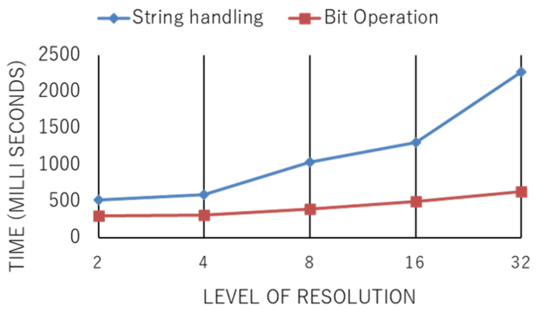


Fig. 8. Encoding performance evaluation based on resolution with a million points

[7] J. Wang and J. Shan, "Space filling curve based point clouds index," in *Proceedings of the 8th International Conference on GeoComputation*, 2005, pp. 551–562.

[8] J. Baert, A. Lagae, and P. Dutré, "Out-of-core construction of sparse voxel octrees," in *Proceedings of the 5th high-performance graphics conference*. ACM, 2013, pp. 27–32.

[9] T. Zäschke, C. Zimmerli, and M. C. Norrie, "The ph-tree: a space-efficient storage structure and multi-dimensional index," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 397–408.

[10] S. Psomadaki, P. Van Oosterom, T. Tijssen, and F. Baart, "Using a space filling curve approach for the management of dynamic point clouds," *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, IV-2 W*, vol. 1, pp. 107–118, 2016.

[11] T. Zäschke and M. C. Norrie, "Efficient z-ordered traversal of hypercube indexes," in *17. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW 2017)*. ETH Zürich, 2017.

[12] M. Vinkler, J. Bittner, and V. Havran, "Extended morton codes for high performance bounding volume hierarchy construction," in *Proceedings of High Performance Graphics*. ACM, 2017, p. 9.

[13] B. Lin, L. Zhou, D. Xu, A.-X. Zhu, and G. Lu, "A discrete global grid system for earth system modeling," *International Journal of Geographical Information Science*, vol. 32, no. 4, pp. 711–737, 2018.

[14] X. He, J. Zhao, L. Sun, Y. Huang, X. Zhang, J. Li, and C. Ye, "Line-based road structure mapping using multi-beam lidar," *CoRR*, vol. abs/1804.07028, 2018. [Online]. Available: http://arxiv.org/abs/1804.07028

[15] P. van Oosterom, O. Martinez-Rubi, M. Ivanova, M. Horhammer, D. Geringer, S. Ravada, T. Tijssen, M. Kodde, and R. Gonçalves, "Massive point cloud data management: Design, implementation and execution of a point cloud benchmark," *Computers & Graphics*, vol. 49, pp. 92–125, 2015.

[16] N. Sirdeshmukh, "Utilizing a discrete global grid system for handling point clouds with varying locations, times, and levels of detail," Master's thesis, Delft University of Technology, 6 2018.

[17] M. Pavlovic, K.-N. Bastian, H. Gildhoff, and A. Ailamaki, "Dictionary compression in point cloud data management," in *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2017, p. 45.

[18] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," IBM Ltd., 1966.

[19] P. van Oosterom and T. Vijlbrief, "The spatial location code," in *Proceedings of the 7th international symposium on spatial data handling, Delft, The Netherlands*, 1996.

[20] J. P. Snyder, "An equal-area map projection for polyhedral globes," *Cartographica: The International Journal for Geographic Information and Geovisualization*, vol. 29, no. 1, pp. 10–21, 1992.

[21] X. Zhao, J. Bai, J. Chen, and Z. Li, "A seamless visualizaton model of the global terrain based on the qtm," in *Advances in artificial reality and tele-existence*. Springer, 2006, pp. 1136–1145.