Front. Comput. Sci., 2017, 11(4): 608–621
DOI 10.1007/s11704-016-5003-y

RESEARCH ARTICLE

A parallel computing framework for big data

Guoliang CHEN[1,2], **Rui MAO** (✉)[1,2], **Kezhong LU**[1,2]

1 Guangdong Province Key Laboratory of Popular High Performance Computers, Shenzhen 518060, China
2 College of Computer Science and Software Engineering, Shenzhen University, Shenzhen 518060, China

Abstract Big data has received great attention in research and application. However, most of the current efforts focus on system and application to handle the challenges of "volume" and "velocity", and not much has been done on the theoretical foundation and to handle the challenge of "variety". Based on metric-space indexing and computationalcomplexity theory, we propose a parallel computing framework for big data. This framework consists of three components, i.e., universal representation of big data by abstracting various data types into metric space, partitioning of big data based on pair-wise distances in metric space, and parallel computing of big data with the NC-class computing theory.

Keywords NC-computing, metric space, data partitioning, parallel computing

1 Introduction

Today, big data is emerging from the network connected world, as well as conventional grand science and engineering applications. Further, new technology such as sensor and communication, and new application such as Internet of Things and Cloud Computing, also promote the development of big data. It is now the era of big data focusing on information technology research and application. However, most of the existing research of big data focuses on system technology and applications [1,2], not much has been done on the theoretical foundation. Further, among the three basic characteristics or challenges of big data, namely "volume", "velocity" and "variety", the first two have attracted much attention while "variety" is relatively less studied.

It is well known that most regular sequential computing problems are *P*-class problems, which can be solved in polynomial time on a deterministic Turing machine. However, in the context of big data, polynomial time is practically unacceptable, and thus *P*-class problems become intractable. That is, they are theoretically solvable, but the solving time is so long that the problems become unsolvable in practice. For example, even for a hard disk whose reading speed is as fast as 6GB/second, it still takes more than 5 years just to sequentially scan 1EB data [3].

How to make an intractable problem tractable? In this paper, we propose a simple framework leveraging parallel computing technology. That is, first abstract various data types into metric space to form a universal representation of data and to conquer the challenge of "variety"; second divide a big and complicated problem into a number of small and tractable sub-problems based on intra-distances of data in metric space; third solve those sub-problems in parallel using multiple processors to conquer the challenge of "volume", and finally achieve the solution to the original problem. If the number of sub-problems is polynomial, and can be handled on PRAM model in polylogarithmic time with polynomial processors, the computation is the so called Nick's class computing (NC-computing) [4,5]. The logic relationship among the three components of this framework can be studied in a reverse way. Given big data, a natural idea is to handle it in parallel. Therefore, we propose to use NC-Computing for applicable cases. To apply NC-Computing, one needs to partition data. Thus, we study the problem of data partitioning. Finally, to partition various data types universally, we propose

Received March 2, 2015; accepted March 21, 2016

E-mail: mao@szu.edu.cn

to abstract them into metric spaces. Although all the three components of the proposed framework have been studied before, to the best of our knowledge, this paper is the first piece of work combining the three components together to conquer the "volume" and "variety" challenges of big data.

The rest of this paper is organized as follows: we introduce the universal abstract representation of big data in metric space in Section 2. To impose coordinate system on metric space for the purpose of partitioning, pivot space iselaborated in Section 3, followed by partitioning techniques and algorithms in Section 4. A discussion of parallel computing methodsfor big data, especially the NC-computing, is presented in Section 5. Finally, conclusions are presented in Section 6.

2 Metric space: universal abstract representation of big data

"Variety" is one of the challenges of big data. That is, data is of various types and various formats. Generally, there are two types of solutions to conquer the "variety" challenge. One is customized solution and the other is universal solution [6].

Customized solutions build one system for each data type. Since each system is tailored for a particular data type, its performance can be expected to be high. However, its range of applicability is relatively narrow, and its price is thus relatively high. As a result, the performance-price ratio will be relatively low.Universal solutions build a single system to support a wide range of data types. If the performance is acceptable, universal system is more cost-effective.

As new data types are emerging rapidly, it is not cost-effective, or even not affordable, to adopt customized solutions. Universal solutions are of great important to most big data applications.

To build a universal solution, one usually first adopts a universal data type serving as an abstraction of a wide range of data types, and then builds a universal solution based on the properties of the universal abstraction [6].

As data is getting more and more complex, the relationship among them is usually represented by similarity or distance. In mathematics, the distance of pairs of elements of a nonempty set is called a Metric [7]. Therefore, to build a universal solution, one also needs to find a universal metric serving as an abstraction of a wide range of metrics.

Metric space provides such a universal data type and a universal metric. The idea to first abstract various data types into metric space and then manage and analyze data in metric space, which is known as Big Data Abstraction [6].

Definition 1 A metric space [7] is a nonempty set S with a metric, or a distance function, d defined on pairs of elements of S, with the following properties:

1) For all $x, y \in S, d(x, y) \geqslant 0$, and $d(x, y) = 0$ if and only if $x = y$. (Positivity)

2) For all $x, y \in S, d(x, y) = d(y, x)$. (Symmetry)

3) For all $x, y, z \in S, d(x, y) + d(y, z) \geqslant d(x, z)$. (Triangle inequality)

Metric space is a very universal data type, and covers a lot of common data types and their distance functions. Table 1 lists some special cases of metric space.

Table 1 Special cases of metric space

Data type	Distance function
Vector	Euclidean distance
Vector	Manhattan distance
Text	Edit distance
Image	Hausdroff distance
Protein sequence	Global alignment
DNA sequence	Hamming distance

Table 1 indicates that many common data types and their distance functions are special cases of metric space. Therefore, any solution built for metric space works for those special cases.

Metric space only requires a user-defined distance function satisfying the three metric properties. An interpretation of the data objects in a coordinate system is not required. That is, data has no coordinates.

The great universalness of metric space is also its problem. Since data has no coordinates, many mathematical tools are not directly applicable, and it is hard to define a criterion to partition data. In the next section, we show how to impose coordinates to metric space.

3 Pivot space: impose coordinates to metric space

To impose coordinates to metric space, one canfirst select a number of reference points, called pivots [8–10], and then use the distances from a particular point to the pivots as its coordinates. As a result, the data is mapped from metric space to a multi-dimensional space, called pivot space.

In this section, we first introduce the concept of pivot space and how coordinates can be imposed to it, and then present the concept of complete pivot space and discuss the distortion of the pair-wise distance of data incurred by the mapping.

3.1 Pivot space

Let (M, d) be a metric space, where M is the space containing the data, and d is the distance function, i.e., the metric. Let $S = \{x_i | x_i \in M, i = 1, 2, \ldots, n\}$, be the database, $n \geqslant 1$. S is a finite indexed subset of M. Duplicates are not allowed. Let $P = \{p_j | j = 1, 2, \ldots, k\}$ be a set of pivots. P is a finite indexed subset of M. Duplicates are not allowed.

The following mapping, $\mathrm{F}_{P,d}$ [11], maps a point in S to a point in the non-negative orthant of R^k, where the jth coordinate of the image represents the distance to p_j:

$$\begin{aligned}\mathrm{F}_{P,d} : \mathrm{M} \rightarrow \mathrm{R}^k : x^p \equiv \mathrm{F}_{P,d}(x) &= (f_1(x), \ldots, f_k(x)) \\ &= (d(x, p_1), d(x, p_2), \ldots, d(x, p_k))\end{aligned}$$

We say x^p, a k-dimensional vector in R^k, is the image of x.

The **pivot space** [11] of S is defined as the image set of S under the mapping $\mathrm{F}_{P,d}$, lying in the non-negative orthant of R^k. When there is no confusion, pivot space also refers to the k-dimensional space it lies in.

$$\begin{aligned}\mathrm{F}_{P,d}(S) = \{x^p | x^p = \mathrm{F}_{P,d}(x) = (d(x, p_1), d(x, p_2), \\ \ldots, d(x, p_k)), x \in S\}.\end{aligned}$$

Two examples of pivot space are presented next.

Example 1 Let A, B and C be three numbers of values 1, 2 and 3, respectively, and the distance function be the absolute value of the difference of two numbers. Therefore, these three numbers and the above distance function form a metric space. If A is selected as the pivot, then A, B and C are mapped to 0, 1 and 2, respectively. If B is selected as the pivot, then A, B and C are mapped to 1, 0 and 1, respectively. If C is selected as the pivot, then A, B and C are mapped to 2, 1 and 0, respectively.

Example 2 Data in this example are the center and the four corners of the unit square.

A metric space can be formed by these points and the Euclidean distance. Let's consider two sets of pivots, i.e., {(0, 0), {1, 1}} and {(0, 0), (0, 1)}. Figure 1 shows the original data and the two pivot spaces deduced from the two sets of pivots. For each set of pivots, since the number of pivots is 2, the dimension of the pivot space is 2. Each point from the unit square is mapped into the 2-dimensional pivot space, with its distance to the first pivot as the x-coordinate, and its distance to the second pivot as the y-coordinate. For example, the original coordinate of point D is (1, 1). Its Euclidean distance to (0, 0), the first pivot in both cases, is 1.414. Its Euclidean distance to (1, 1), the second pivot in Fig. 1(b) is 0, and therefore its coordinate in the pivot space shown in Fig. 1(b) is (1.414, 0). Similarly, point D's Euclidean distance to (0, 1), the second pivot in Fig. 1(c) is 1, and therefore its coordinate in the pivot space shown in Fig. 1(c) is (1.414, 1).

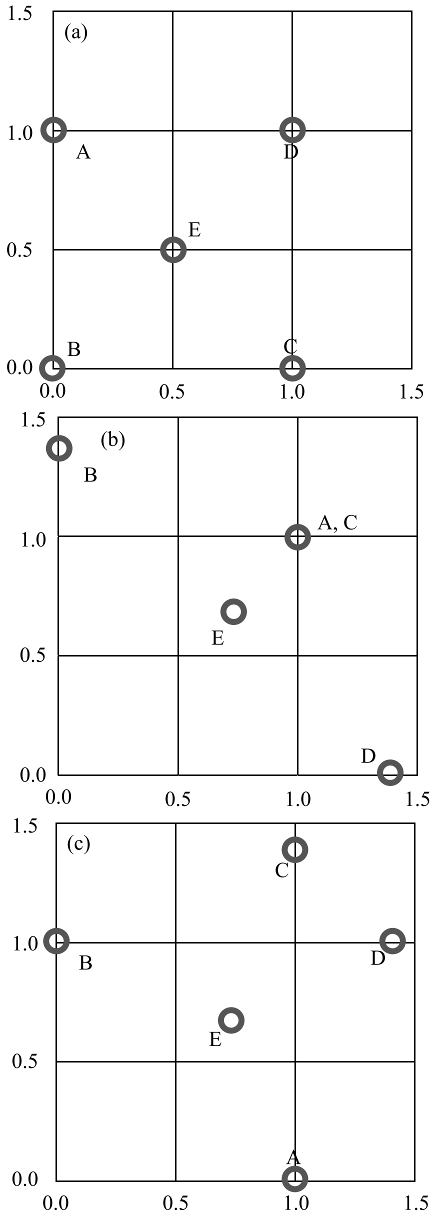

Fig. 1 Pivot spaces of the center and four corners of the unit square with Euclidean distance. (a) Original data; (b) pivots (0,0) and (1,1); (c) pivots (0,0) and (0,1)

With the mapping to pivot space, data in metric space is imposed with coordinates. Unfortunately, this mapping is usually distorted. That is, distance between two points in the metric space usually does not equal to the distance between the two points in the pivot space. For example, the distance between point A and point D is 1 in the original metric space (Fig. 1(a)). In the pivot space shown in Fig. 1(b), the coordinates of point A and D are (1, 1) and (1.414, 0), respectively, and their Euclidean distance is 1.37.

The complete pivot space is introduced next to handle this problem.

3.2 Complete pivot space

The **complete pivot space** of S, $\mathrm{F}_{S,d}(S)$, is the pivot space of S when all points in S are selected as pivots [11].

Example 3 shows the complete pivot space of the three points discussed in Example 1.

Example 3 Let A, B and C be three numbers of values 1, 2 and 3, respectively, and the distance function be the absolute value of the difference of two numbers. Therefore, the complete pivot space is 3-dimensional. The coordinates of A, B and C, as plotted in Fig. 2, are (0, 1, 2), (1, 0, 1) and (2, 1, 0), respectively.

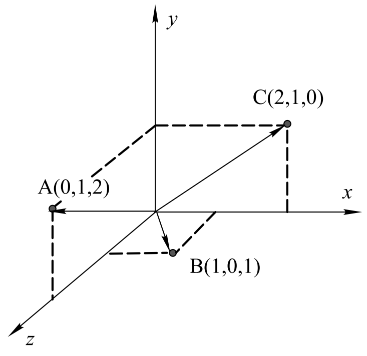

Fig. 2 The complete pivot space of three numbers, A, B and C of values 1, 2 and 3

Given two vectors A $= (a_1, a_2, \ldots, a_n)$ and B $= (b_1, b_2, \ldots, b_n)$, the infinity distance, L^∞, between A and B is defined as:

$$\mathrm{L}^\infty(\mathrm{A}, \mathrm{B}) = \max\{|a_i - b_i|, i = 1, 2, \ldots, n\}.$$

It has been shown that the mapping from a metric space to its complete pivot space has no distortion with respect to the infinity distance in the complete pivot space [11]. That is, give metric space (M, d), for any two point x and y in S, the following holds:

$$d(x, y) = \mathrm{L}^\infty(\mathrm{F}_{S,d}(x), \mathrm{F}_{S,d}(y)).$$

As a result, the data set S from any general metric space can be mapped without distortion into the complete pivot space with respect to the L^∞ distance without loss of any distance information. Instead of working on S in a black-box metric space without any domain information, it is equivalent to work on the complete pivot space of S in the much more perceptible vector space R^n with the L^∞ distance.

Please note that when Euclidean distance in the pivot space is considered, distortion is still produced, as discussed next.

3.3 Distortion with respect to Euclidean distance

We now consider the distortion of the mapping from the metric space to the pivot space when Euclidean distance, L^2, is used. Theorem 1 characterizes the distortion of the incomplete pivot space.

Theorem 1 Let (M, d) be a metric space. Let $P = \{p_j | j = 1, 2, \ldots, k\}$ be the set of pivots, and for $x, y \in M$, let $x^p = \mathrm{F}_{P,d}(x)$ and $y^p = \mathrm{F}_{P,d}(y)$. Then, $0 \leqslant \mathrm{L}^2(x^p, y^p) \leqslant \sqrt{k} d(x, y)$. Moreover, $\mathrm{L}^2(x^p, y^p) = 0$ when $d(x, p_i) = d(y, p_i)$ for all $i = 1, 2, \ldots, k$.

Proof Since

$$x^p = [d(x, p_1), d(x, p_2), \ldots, d(x, p_k)],$$
$$y^p = [d(y, p_1), d(y, p_2), \ldots, d(y, p_k)],$$

then,

$$\mathrm{L}^2(x^p, y^p) = \sqrt{\sum_{i=1}^{k} |d(x, p_i) - d(y, p_i)|^2}$$
$$\leqslant \sqrt{\sum_{i=1}^{k} |d(x, y)|^2} = \sqrt{k} d(x, y).$$

Clearly $\mathrm{L}^2(x^p, y^p) \geqslant 0$, and $\mathrm{L}^2(x^p, y^p) = 0$ when $d(x, p_i) = d(y, p_i)$ for all $i = 1, 2, \ldots, k$. □

Theorem 2 describes the distortion with respect to the complete pivot space.

Theorem 2 Let $x, y \in S$, $x^p = \mathrm{F}_{S,d}(x)$, and $y^p = \mathrm{F}_{S,d}(y)$. Then, in the complete pivot space $\sqrt{2} dx, y \leqslant \mathrm{L}^2(x^p, y^p) \leqslant \sqrt{n} d(x, y)$, and $\sqrt{2} d(x, y) = \mathrm{L}^2(x^p, y^p)$ when $d(x, x_i) = d(y, x_i)$ for all $i = 1, 2, \ldots, n$, $i \neq t$, $i \neq l$.

Proof Since

$$x^p = \mathrm{F}_{S,d}(x) = [d(x, x_1), d(x, x_2), \ldots, d(x, x_n)],$$
$$y^p = \mathrm{F}_{S,d}(y) = [d(y, x_1), d(y, x_2), \ldots, d(y, x_n)],$$

$\mathrm{L}^2(x^p, y^p) \leqslant \sqrt{n} d(x, y)$ follows from Theorem 1.

Since $x, y \leqslant S$, we ignore the trivial case that $x = y$ and let $x_t = x$ and $x_l = y$. Then

$$\begin{aligned}
\mathrm{L}^2(x^p, y^p) &= \sqrt{\sum_{i=1}^{n} |d(x, x_i) - d(y, x_i)|^2} = \sqrt{\sum_{i=1, i\neq t,l}^{n} |d(x, x_i) - d(y, x_i)|^2 + |d(x, x_t) - d(y, x_t)|^2 + |d(x, x_l) - d(y, x_l)|^2} \\
&= \sqrt{\sum_{i=1, i\neq t,l}^{n} |d(x, x_i) - d(y, x_i)|^2 + |d(x, x) - d(y, x)|^2 + |d(x, y) - d(y, y)|^2} = \sqrt{\sum_{i=1, i\neq t,l}^{n} |d(x, x_i) - d(y, x_i)|^2 + 2d(x, y)^2} \\
&\geqslant \sqrt{2}d(x, y)
\end{aligned}$$

Moreover, equality clearly holds when $d(x, x_i) = d(y, x_i)$ for all $i = 1, 2, \ldots, n, i \neq t, l$. □

4 Distance: the principle of big data partitioning

As a very common step of many data processing tasks and for the sake of parallel computing for big data, data partitioning should be studied for metric space. Basically, there are two types of partitioning for metric space. Canonical partitioning methods in parallel computing, which are also applicable to metric spaces, distribute data to multiple processors without considering their relationship or distances to each other [4]. On the other hand, distance-based partitioning divides data into partitions where all data in a partition form certain shape with respect to distance [8,9]. A common example of such shape is a ring, i.e., all data in a partition is within a range of distance to a pivot.

This section gives a comprehensive survey of existing partitioning methods for metric space. Canonical partitioning methods in parallel computing are first surveyed in Section 4.1. Then, existing distance-based partitioning methods are surveyed from two aspects, i.e., shapes of partitions (Section 4.2) and common partitioning algorithms (Section 4.3).

4.1 Canonical partitioning methods in parallel computing

A basic idea of parallel computing is divide-and-conquer. To do so, one should first divide the data, and distribute them to multiple processors for parallel processing. In terms of the number of data objects in partitions, canonical partitioning methods include uniform partitioning, square root partitioning, logarithmic partitioning and functional partitioning [4].

4.1.1 Uniform partitioning

For n data points and p identical processors, the goal of uniform partitioning is to achieve load balance. To do so, it divides the n data points into p segments evenly, where each of the segment consists of $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$ data points (Fig. 3). Since the processors are identical, they can finish the processing of segments simultaneously, and therefore the load balance is achieved. For example, the parallel sorting by regular sampling (PSRS) algorithm [12] employs uniform partitioning and achieves linear speedup.

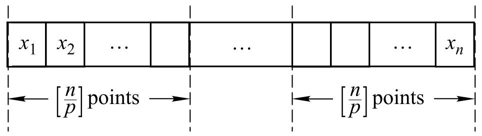
x_1
x_2
···
···
···
x_n
$\left[\frac{n}{p}\right]$ points
$\left[\frac{n}{p}\right]$ points

Fig. 3 Uniform partitioning

4.1.2 Square root partitioning

In parallel computing, if the number of processors is too large, the large number of fine grained subtasks leads to huge cost for subtask creation and large cost of communication among subtasks. On the contrary, if the number of processors is too small, the parallelism is not fully exploited. It is necessary to determine an "optimal" number of processors.

Square root partitioning is a special case of uniform partitioning. For n data points, square root partitioning takes use of $\sqrt{n}$ processors. It divides data into $\sqrt{n}$ segments, each of which consists of $\sqrt{n}$ data points (Fig. 4). For example, the Valiant' Merging Algorithm [13] employs square root partitioning. With n processors, it merges two ordered sequences of size n in O($\log \log n$) time.

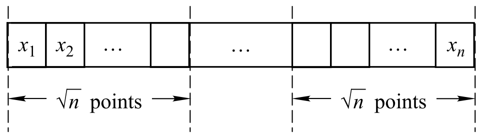

Fig. 4 Square root partitioning

4.1.3 Logarithmic partitioning

Logarithmic partitioning is another special case of uniform partitioning. For n data points, logarithmic partitioning divides the data into $n/\log n$ segments, each of which consists of $\log n$ data points (Fig. 5). For example, Shiloach's merge algorithm employs logarithmic partition [14].

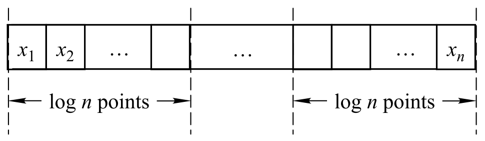

Fig. 5 Logarithmic partitioning

4.1.4 Functional partitioning

Usually, the size of segment to divide data should be determined according to the functionality of particular applications, case by case. For example, the (m, n)-selection problem selects first m smallest values from n data points. A functional partitioning strategy for this problem is to divide n data points into segments of size m, and then make pair-wise comparison of segments in parallel to get the answer [15].

4.2 The shapes of distance-based partitions

Those canonical partitioning methods in parallel computing discussed above do not consider the distances among data. For some data processing tasks, such as indexing and clustering, usually an order should be imposed on partitions, or partitions should be distinguishable from each other spatially. As a result, for these data processing tasks in metric space, data should be partitioned based on their distances to each other. In distance-based partitioning, usually partitions are of certain shapes with parameters. For example, a common shape is a ring such that all points in a partition is within a distance range to a pivot. The lower and upper bounds of the range are the parameters to the shape. In the following, the shapes are surveyed in Section 4.2.1 and their unification in Section 4.2.2. The algorithms to determine the parameters to the shapes are discussed in Section 4.3.

4.2.1 Survey of partitioning shapes

In terms of shape, there are at least two types of partitioning, i.e., ball partitioning and hyper-plane partitioning [8,9,16,17]. We show the shapes of these partitioning in metric space and pivot space [11] in the following.

4.2.1.1 Ball partitioning

Intuitively, ball partitioning uses balls with the same center and various radii to partition data. The primitive form of ball partitioning was proposed in the work of vantage point tree (VPT) [18,19].

Given n points, with a vantage point (or pivot), VP, predetermined, the distances from VP to points are calculated and the median, m, of the distances is obtained. Then the data is partitioned by a ball centered at VP with m as its radius. That is, VPT partitioning is a balanced binary partitioning, and points whose distances to VP are equal to or less than m go to one partition, while points whose distances to VP are larger than m go to the other partition (Fig. 6).

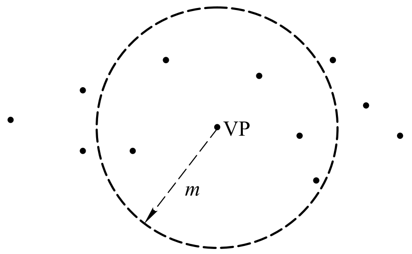

Fig. 6 VPT partitioning in the metric space

Obviously, the time complexity of one run of VPT partitioning is O(n). In constructing a VPT, VPT partitioning executes recursively. Consequently, the overall time complexity of partitioning in the construction of a VPT is O($n \log n$).

Mao et al. proposed to analyze metric-space partitioning in pivot space, a multi-dimensional space whose coordinates are distances to each pivot, respectively [20]. VPT partitioning in pivot space is illustrated in Fig. 7. Since there is only one pivot, the pivot space of the VPT partitioning is 1-dimensional, and the partition boundary changes from a ball in the original metric space to the point of value m.

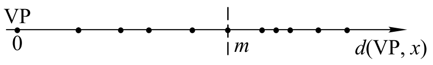

Fig. 7 VPT partitioning in the pivot space

Multiple vantage point tree (MVPT) [21], an extension of VPT, takes use of multiple vantage points and multiple radii for each vantage point, respectively. MVPT partitioning of two vantage points and two radii for each vantage point is illustrated for the cases of metric space and pivot space in Figs. 8 and 9, respectively. It can be seen that the partition boundary of MVPT partitioning changes from balls in the original metric space to straight lines in 2-dimensional pivot space.

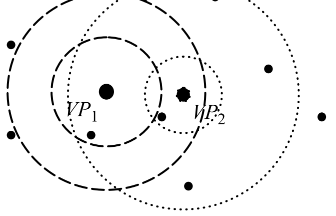

Fig. 8 MVPT partitioning in the metric space

Similarly, MVPT partitioning executes recursively to construct an MVPT. Obviously, if the partitioning is balanced, the time complexity of one run of MVPT partitioning is O(n), and the overall time complexity of partitioning in the construction of an MVPT is O($n \log n$).

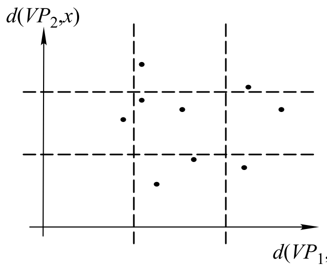

Fig. 9 MVPT partitioning in the pivot space

4.2.1.2 Hyper-plane partitioning

Intuitively, hyper-plane partitioninguses hyper-planes to partition data. The primitive form of hyper-plane partitioning was proposed in the work of general hyper-plane tree (GHT) [18].

Given n points, with two centers (or pivots), C_1 and C_2, predetermined, GHT partitioning uses a hyper-plane equal-distant to C_1 and C_2 to divide the data into two partitions. That is, points closer to C_1 or equal-distant to C_1 and C_2 go to one partition, while points closer to C_2 go to the other partition (Fig. 10) [20].

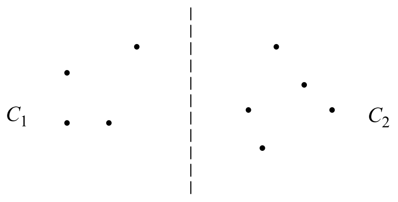

Fig. 10 GHT partitioning in the metric space

Obviously, the time complexity of one run of GHT partitioning is O(n). GHT partitioning executes recursively to construct a GHT. It should be noted that GHT partitioning is not guaranteed to be balanced. The overall time complexity of partitioning in the construction of a GHT is O($n \log n$) in the best case when every partitioning is balanced, and O(n^2) in the worst case.

GHT partitioning in the pivot space is illustrated in Fig. 11. The partition boundary is a straight line passing the origin of slope 1, which can be defined by:

$$d(C_1, x) - d(C_2, x) = 0$$

In addition to GHT, GNAT [22], M-Tree [23], SA-tree [24], and i-distance [25] all employ hyper-plane partitioning.

Complete general hyper-plane tree (CGHT) [20], an extension of GHT, takes full use of the values of $d(C_1, x)$ and $d(C_2, x)$, or $d(C_1, x) - d(C_2, x)$ and $d(C_1, x) + d(C_2, x)$, precisely, in partitioning the data. For a constant T, the curve defined by $d(C_1, x) - d(C_2, x) = \mathrm{T}$ is a hyperbola in the metric space and a straight line of slope 1 in the pivot space. The curve defined by $d(C_1, x) + d(C_2, x) = \mathrm{T}$ is an ellipse in the metric space and a straight line of slope -1 in the pivot space.

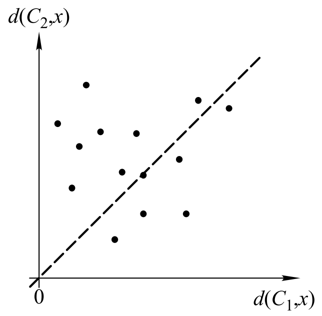

Fig. 11 GHT partitioning in the pivot space

CGHT partitioning with two hyperbolas and two ellipses for the cases of metric space and pivot space is illustrated in Figs. 12 [20] and 13 [20], respectively.

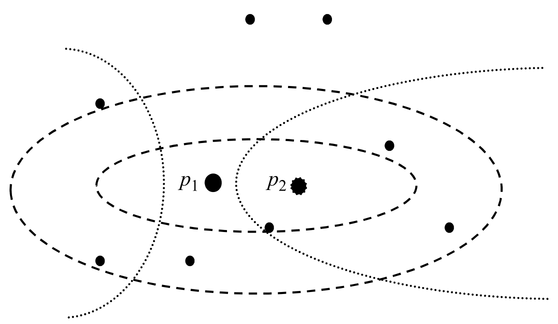

Fig. 12 CGHT partitioning in the metric space [20]

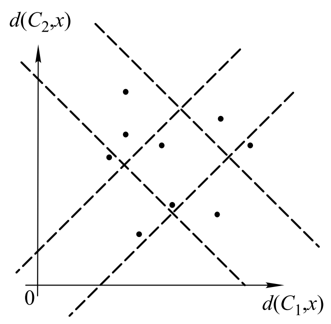

Fig. 13 CGHT partitioning in the pivot space [20]

Similarly, CGHT partitioning executes recursively to construct a CGHT. The time complexity of one run of CGHT partitioning is O(n). CGHT partitioning is not guaranteed to be balanced. The overall time complexity of partitioning in the construction of a CGHT is $O(n \log n)$ in the best case when every partitioning are balanced, and $O(n^2)$ in the worst case.

4.2.2 Unification of shapes of partitions

Mao et al. studied the relationship between ball partitioning and hyper-plane partitioning [20]. They first unified the two kinds of partitioning, and then compared their performance.

4.2.2.1 Unification of ball and hyper-plane partitioning

The partition boundaries of both partitioning in the original metric space, i.e., ball and hyper-plane (Figs. 6 and 10), are quite different from each other. However, if the pivot space case is considered (Figs. 7 and 11), and if the partition boundary in Fig. 11 be considered as a line $x = m$, one can easily see that the only difference between the two partition lines in Figs. 7 and 11 is the slope. That is, ball partitioning and hyper-plane partitioning are just rotations of each other.

Next, let's compare MVPT and CGHT, of which the numbers of pivots are both 2, and the numbers of partitions are both 9. Similarly, in the original metric, the partition boundaries, i.e., balls (Fig. 8) and hyperbolas or ellipses (Fig. 12), are quite different from each other. However, in the pivot space (Figs. 9 and 13), one can again easily see that the partition boundaries are rotations of each other.

Similar conclusion can be drawn for the case of more pivots.

As a result, ball partitioning and hyper-plane partitioning are unified so that they are both special cases of linear (to avoid confusion, the word "hyper-plane" is not used) partitioning in the pivot space. The linear partition boundaries of ball partitioning are perpendicular to axes, while the linear partition boundaries of hyper-plane partitioning are equal-distant to axes.

4.2.2.2 Performance analysis

Next, what is the optimal value of slop among all possible linear partitions in the pivot space?

Although there is no general answer for all cases, Mao et al. gave a partial answer for the case of similarity indexing for range query [20].

The performance metric is the average number of distance calculations, which is independent from implementation details and machine environment, to answer range queries. Mao et al. proposed the r-neighborhood analysis approach [20] to analyze the query performance.

Informally, r-neighborhood is the neighborhood of partition boundary. If a query object falls into the r-neighborhood, both sides of the boundary have to be further accessed without making any pruning [20]. Mao et al. show that the number of points in the r-neighborhood is a good indicator of query performance, given the assumption that the queries have the same distribution as the database [20]. Less number of points in the r-neighborhood indicates better query performance.

The following four results are shown in [20]:

1) In the pivot space, among all the slopes, VPT partitioning possesses the minimal width of the r-neighborhood, while GHT partitioning possesses the maximal one.
2) For 2-dimensional normally distributed data in the pivot space, the number of points in the r-neighborhood of VPT partitioning is less than that of GHT partitioning.
3) For a comprehensive test suite experimentally, MVPT has less number of points in the r-neighborhood than that of CGHT.
4) For a comprehensive test suite experimentally, MVPT outperforms CGHT in range query performance.

In summary, Mao et al.'s results indicate that ball partitioning potentially outperforms hyper-plane partitioning, at least for similarity indexing. Consequently, since hyper-plane partitioning is a rotation of ball partitioning, dimension rotation, an effective technique in multi-dimensional data processing, might not be as effective in metric space.

4.3 Distance-based partitioning algorithms

Given the unification of partitioning methods above, the partitioning problem can be formalized as follows: for n data points, k pivots, and fanout m for each pivot, recursively divide data into f partitions for each pivot so that the total number of partitions is f^k. In this section, common distance-based partitioning algorithms are surveyed.

4.3.1 Random partitioning

Random partitioning selects random split values to split data. Its algorithmic steps are show in Fig. 14.

4.3.2 Balanced partitioning

There are two meanings of "balance" in the context of partitioning. One meaning is identical to that of the traditional balanced partitioning in that the numbers of points in partitions are the same or almost the same. We call it "cardinality-balanced" partitioning. An example is the VPT partitioning [18,19].

```
(1) Select a random pivot $p$, scan the data and
    compute $l = \min(d(p, x))$ and $u = \max(d(p, x))$;
(2) Randomly generate $f_{-1}$ distinct values in $(l, u)$
    and order them such that
    $l < m_1 < \cdots < m_{f-1} < u$;
(3) Use $m_1, m_2, \ldots, m_{f-1}$ as the split values to split
    data into $f$ partitions based on $d(p, x)$;
(4) For all the partitions created in (3), repeat (1),
    (2) and (3) with another random pivot until all
    the pivots are exhausted and $f^k$ partitions are
    created.
```

Fig. 14 The algorithmic steps of random partitioning

The second meaning is that the ranges of data in partitions are the same, which is called "distance-balanced" partitioning [8]. It should be noted that cardinality-balanced partitions may not be balanced in space, vice versa. In this partitioning, the range of the distance function is partitioned into k equal size intervals, independent of the cardinality of the partitions. The time to build the tree is very fast, as the consideration of the actual data is minimal. This construction also leads to unbalanced trees, but without the benefits of considering the actual data distribution. The cardinality-balanced partitioning (original form of VPT partitioning) and the distance-balanced partitioning of VPT are illustrated in Fig. 15 [8].

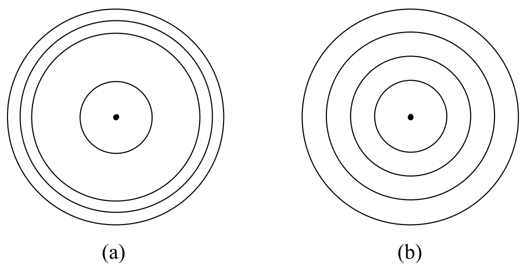

Fig. 15 The (a) cardinality- and (b) distance-balanced partitioning of VPT

The algorithmic steps of cardinality- and distance-balanced partitioning are shown in Figs. 16 and 17, respectively. The only difference is the generation of the split values. Cardinality-balanced partitioning uses percentiles as the split values, while distance-balanced partitioning just evenly divides the range of the data.

```
(1) For an arbitrary pivot $p$, let $m_i$ be the $(100i/f)$ th
    percentile of $d(p, x)$, $i = 1, 2, \ldots, f-1$;
(2) Use $m_1, m_2, \ldots, m_{f-1}$ as the split values to split
    data into $f$ partitions based on $d(p, x)$;
(3) For all the partitions created in (2), repeat (1)
    and (2) with the next pivot until all the pivots
    are exhausted and $f_k$ partitions are created.
```

Fig. 16 The algorithmic steps of cardinality-balanced partitioning

4.3.3 Balanced vs. unbalanced partitioning

All the partitioning methods discussed so far aim to be balanced, in cardinality or space. In construction of search trees, a principle is to make the tree balanced with respect to cardinality. The reason is that answering an exact match search essentially descends a path from the root to a leaf. If the tree is balanced, the average height of tree, which is proportional to the cost of search, is minimized.

```
(1) For an arbitrary pivot $p$, scan the data and
    compute $l = \min(d(p, x))$ and $u = \max(d(p, x))$;
(2) Let $m_i = l + i(u-l) / f$, $i = 1, 2, \ldots, f-1$;
(3) Use $m_1, m_2, \ldots, m_{f-1}$ as the split values to split
    data into $f$ partitions based on $d(p, x)$;
(4) For all the partitions created in (3), repeat (1),
    (2) and (3) with the next pivot until all the pivots
    are exhausted and $f_k$ partitions are created.
```

Fig. 17 The algorithmic steps of distance-balanced partitioning

However, for similarity indexing, multiple paths from the root to leaves could be descended to answer a similarity query. Therefore, in addition to the balance of the index tree, the number of paths descended is another important factor of query performance. Brin argued that the effectiveness of a metric-space index depends on the algorithm's ability to capture the intrinsic hierarchical structure of the data [22]. Chavez and Navarro showed that unbalancing could be helpful to improve query performance [26]. Further, as discussed by Mao et al. [20], the number of points in the r-neighborhood is a good indicator of query performance. Therefore, in the construction of the index tree, it is usually beneficial to place the partition boundaries in sparse regions of data. Following this idea, the clustering partitioning was proposed in [27].

4.3.4 Clustering partitioning

Clustering partitioning [27] aims at finding the intrinsic clustering of data and placing the partition boundaries in the sparse region in between the intrinsic clusters. In addition, clustering partitioning tries to achieve as much balance as possible.

The algorithm runs for pivots one by one. For a pivot p, the data to cluster is $d(p, x)$. Therefore, any clustering algorithm applicable to numbers, such as K-means [28] and DBSCAN [29] can be employed. If K-means is to be employed, the initial clustering can be determined by cardinality-balanced partitioning.

The algorithm achieves balance by determining the pivot leading to the most balanced partitioning from a set of available pivots, where the balance of a partitioning is measured by the variance of the sizes of partitions, the smaller the more balanced. The clustering partitioning algorithm is shown in

Fig. 18.

```
ClusteringPartition()
{
  // initially the whole data set is a partition
  while( there ex ists a partition  C whose pivots set P
        is not empty)
  {
    for ( each p in P)
    {
      compute the distances to p;
      find f−1 split values by a clustering algorithm;
      compute the variance of the sizes of partitions;
    }
    find p-mv, resulting in the smallest variance;
    split C based on the split values of p-mv;
    remove p-mv from P;
    copy P as the pivot set for each of the f partitions;
    remove C;
  }
  return all the partitions;
}
```

Fig. 18 The clustering partitioning algorithm

4.3.5 Time and space complexity

Obviously, the time complexities of one run of random and balanced partitioning are both O(n). Their overall time complexities of recursive partitioning are both O($n \log n$). Their space complexities are both O(1).

For clustering partitioning, if we assume the time complexity of the clustering algorithm is no more than O(n) and space complexity is no more than O(1), the time and space complexities of clustering partitioning are same as other partitioning algorithms discussed above.

5 Parallel computing: the key to make intractable big data problems tractable

Now that big data is partitioned into small subsets, parallel computing can be applied to solve big data problems. In this section, we start with a discussion of P-class problems and their tractability in the context of big data, then elaborate what kind of big data problems can be considered tractable, and finally introduce the NC-computing and its possible application to big data problems.

5.1 P-class problem and its tractability

In traditional computational complexity theory, problems are often divided into P-class problems, NP-class problems, NPC-class problems and so on according the difficulty, or the required time to solve them.Among them, P-class problems are defined as the class of problems that are solvable in polynomial time, namely in O(n^k) time, where k is a constant and n is the input size. Many common problems belong to P-class problems.

In traditional cases, the input sizes of problems are relatively small. If a problem can be solved in polynomial time, it is considered to be quickly solvable, and otherwise not. Therefore, P-class problems are tractable problems, and other problems are intractable problems. But in case of big data, the input sizes of problems may be very large. Even if they have linear or quadratic time complexities, their solving processes become very slow. Therefore, at this time P-class problems can no longer be deemed as tractable [3].

At present, parallel computing has been widely developed and applied. When the time complexity of the best algorithm for a problem is close to or reach its theoretical lower bound, it is difficult or impossible to reduce the computation time. In this case, parallel computing is possibly the only way to effectively shorten the computation time while ensuring exact solution. In case of big data, to quickly solve a problem with large input size, we can resort to parallel computing. However, due to the limitation of the number of available processors and the nature of the problems, not all problems can be quickly solved by parallel computing.

Assume that a problem K has a parallel algorithm A, the number of processors to run algorithm A is $P(n)$, the running time of algorithm A is $T(n)$, and the total workload of algorithm A is $W(n)$. It is straightforward that $P(n) \cdot T(n) \geqslant W(n)$ and algorithm A can be simulated on a serial computer in $W(n)$ time. Obviously, if K is not a P-class problem, then $W(n)$ must not be polynomial. Due to the hardware limitation of parallel computer, $P(n)$ cannot be greater than polynomial. Thus, $T(n)$ must not be polynomial, either. In other words, for a problem that is not P-class problem, even if it is solved on a parallel computer, its time complexity cannot be improved to polynomial. Therefore, in case of big data, we only study whether P-class problems can be quickly solved on a parallel computer.

5.2 Big data tractable problem

In the case of big data, if a problem can be quickly solved with appropriate number of processors, then we consider it **tractable**. Polynomial time is usually not acceptable for big data. Therefore, we want a problem can be solved in lower than polynomial time. A time complexity lower than polynomial is usually polylogarithmic, which is O($\log^k n$), where k is a constant and n is the input size. For any nonnegative constant ε, there is $\log^k n = o(n^\varepsilon)$. Sublinear complexity is O(n^k), where $0 < k < 1$. Under certain conditions, sublinear time may be faster than polylogarithmic time, e.g., when $n < 0.5 \times 10^9$, $n^{0.5} < \log^3 n$.

Accordingly, we can give a quantitative description of big data tractable problem. The so-called "quickly solved" refers to that the running time is polylogarithmic, namely $O(\log^k n)$. The so-called "appropriate number of processors" refers to that the number of processors is polynomial, which is linear or nearly linear in practice. Therefore, in the case of big data, if a problem can be solved with polynomial number of processors in polylogarithmic time, it is called a big data tractable problem, and otherwise a big data intractable problem.

In the above definition, we adopt polylogarithmic time rather than sublinear time based on the following three factors: 1) Only a small number of problems have sublinear time complexity, while relatively more problems have polylogarithmic time complexity, especially in parallel computing. 2) For many parallel computing models, if a problem can be solved on a model in polylogarithmic time, it is likely that it can be solved on another model in polylogarithmic time. However, it is not true for sublinear time. 3) The class of problems with polylogarithmic time complexity has good closeness. It is because that polylogarithmic is closed under addition, multiplication and composition operations. For example, if the output of an algorithm with polylogarithmic time is fed as the input of an algorithm with polylogarithmic time, the obtained combinatorial algorithm is also polylogarithmic. Moreover, if an algorithm with polylogarithmic time calls a subroutine with polylogarithmic time for constant times, the obtained combinatorial algorithm is also polylogarithmic.

5.3 NC-computing

In parallel complexity theory, an NC algorithm is defined as a parallel algorithm that runs on a PRAM machine with polynomial number of processors in polylogarithmic time. A problem that has an NC algorithm is called as a Nick's Class problem (NC problem) [5]. An algorithm on any sub-PRAM model is still an NC algorithm no matter what sub-model of PRAM is considered, e.g., PRAM-EREW, PRAM-CREW, or PRAM-CRCW. More generally, NC algorithm is also robust with respect to any other accepted models of parallel computation (Boolean circuit model, LogP [30], Parallel memory hierarchy model [31], etc.).

According to the above definition of big data tractable problem, big data tractable problems can be considered equivalent to NC problems. Fortunately, many common problems belong to NC problems, e.g., operation of several integers, prefix and scan computation, selection and sorting, matrix operations (multiplication, rank, inverse, rank, etc.), linear equations group, tree contraction and expression evaluation, Euler tour technique and derived problems, connected components of graphs, maximal match of graph, many problems in computational geometry, string matching, and so on. Even in case of big data, we can quickly solve these problems on a parallel computer. Many NC algorithms have been proved to be optimal. For example, parallel scan algorithm has $T(n) = O(\log n)$, $P(n) = n/\log n$ and $W(n) = O(n)$. Some cost optimal NC algorithms are superfast. For example, parallel string matching algorithm has $T(n) = O(\log\log n)$ and $P(n) = n/\log n$.

Given two problems K_1 and K_2, if there exists an NC algorithm that can translate problem K_1 to problem K_2, it is called that K_1 can be **NC-reducible** to K_2. The relation of NC-reduction satisfies reflexivity and transitivity. We can use the relation of NC-reduction to indirectly judge whether a problem belongs to NC problems. That is, if problem K_2 belongs to NC problems and problem K_1 can be NC- reducible toproblem K_2, problem K_1 also belongs to NC problems and problem K_1 can be indirectly solved by the NC algorithm of problem K_2. On the other hand, if problem K_1 does not belong to NC problems and problem K_1 is NC-reducible toproblem K_2, problem K_2 also does not belong to NC problems.

However, not all P-class problems have NC algorithms. Many problems, if they are inherently serial or due to additional cost caused by communication and synchronization, cannot be solved in polylogarithmic time. Among P-class problems, there exist a class of problems called P-complete problems. In 1975, Ladner showed that the circuit value problem (CVP) was P-complete [32]. Further examples include some graph problems (maximal independent set, depth-first search, maximum clique, etc.), and some combinatorial optimization problem (linear inequalities, linear programming, etc.) Any P-class problem can be NC-reducible to any P-complete problem. Nick Pippenger is believed to be the first to study the class of problems requiring polylogarithmic time and polynomial size circuits [33]. Therefore, Cook named this class of problems "Nick's Class" or NC for short [5]. A lot of works have been done to propose NC algorithms for various polynomial time sequential algorithms.

Chandra and Stockmeyer's work [34] and Goldschlager's dissertation [35] are believed to be the first to study the relationship between P-complete problems and problems that are unlikely to parallelize. P-complete problems are a class of problems that are most difficult to be parallelized. Today, it is generally conjectured but not proved that $P \neq NC$. That is, all P-complete problems do not have NC algorithm and they all belong to big data intractable problems.

5.4 NC-computing of big data

In case of big data, high order polylogarithmic time may be still slow. We can make further division for NC problems by limiting the order of polylogarithmic. Let NC^i problems be the class of problems that can be solved in $O(\log^i n)$ time on a parallel machine with polynomial number of processors. According to the above definition, $NC = NC^0 \cup NC^1 \cup NC^2 \cup \cdots \cup NC^i \cup \cdots$. Obviously, NC^i problems also satisfy the following hierarchy relation: $NC^0 \subseteq NC^1 \subseteq NC^2 \subseteq \cdots NC^i \subseteq \cdots \subseteq NC$. Let EREW^i, CREW^i, CRCW^i be the classes of problems that can be solved in $O(\log^i n)$ time on the parallel computing model of PRAM-EREW, PRAM-CREW, PRAM-CRCW with polynomial number of processors, respectively. Then, their relation with NC^i problems is

$$NC^i \subseteq \mathrm{EREW}^i \subseteq \mathrm{CREW}^i \subseteq \mathrm{CRCW}^i \subseteq NC^{i+1}, i \geqslant 0.$$

On PRAM model, one can first divide a big data set D into polynomial number of subsets $D_m, m = 1, 2, \ldots$, and then parallel processes each D_m in polylogarithmic time. If the above process is proved to be feasible, this computing is defined as NC-computing of big data. Further, if all D_m can be solved in polylogarithmic time of order i, this computing is defined as NC^i computing of big data. Generally speaking, in case of small data, NC-computing can be used to solve moderately sized problems (e.g., hundreds of thousands of input items) with moderate order polynomial number of processors in low order polylogarithmic time. In case of big data, NC-computing can be used to solve largely sized problems (e.g., millions of input items) with low order polynomial number of processors in moderate order polylogarithmic time.

At last, we demonstrate the progress of NC-computing of big data by a simple ranking problem. Given two arrays A and B whose length are both n, where array A is ordered and array B is unordered,the problem is to compute the rank of each element of array B in array A. In serial computing, for each element of array B, we can obtain its rank in array A by binary search in $O(\log n)$ time, so the total running time is $O(n \log n)$. In the case of big data, n may be very large and then the running time of $O(n \log n)$ is no longer acceptable. Therefore, parallel method must be used. When n processors are used, each processor can compute the rank of an element of array B in parallel. Ranks of all element of array B can be obtained in $O(\log n)$ time, so this is NC^1 computing of big data. When n^2 processors are used, each processor can compare an element of array B and an element of array A. Ranks of all elements of array B can be obtained in $O(1)$ time. Therefore, this is NC^0 computing of big data.

6 Conclusion

In this paper, the problem under consideration is that polynomial time problems which are theoretically tractable and practically not tractable in the context of big data. Nevertheless, these problems usually involve various data types. We propose a universal parallel processing paradigm to make these problems, if they are parallelizable. That is, first, find a universal abstraction for various data types. We propose to use metric space as the universal abstraction for big data in this paper. Second, partition a big data problem into small sub-problems. A number of data partitioning methods in metric space are surveyed. Last, the sub-problems can be handled in parallel. We propose to use NC-computing for these sub-problems if appropriate. With this framework, uses only need to define distance functions for their own data types. After plugging the data into the framework, data will be abstracted into metric spaces, partitioned, and processed in parallel. In conclusion, we try to lay down the foundation of parallel computing theory for big data.

Acknowledgements The authors would like to thank Prof. Wenfei Fan, Prof. Benjamin Wah, and Prof. Guihai Chen for their comments. This work was partially supported by the National High Technology Research and Development Program of China (863 Program) (2015AA015305), the National Natural Science Foundation (NSF) of China (Grant Nos. U1301252, 61471243), NSF of Guangdong (2013B090500055, 2014A030313553), Guangdong Key Laboratory Project (2012A061400024), and NSF of Shenzhen (JCYJ20140418095735561, JCYJ20150529164656096, JCYJ20150731160834611).

References

1. Zhou B, Liu W, Fan C. Big Data: Strategy, Technology and Practice. Beijing: Publishing House of Electronics Industry, 2013
2. Tu Z. Big Data. Guilin: Guangxi Normal University Press, 2012
3. Fan W, Geerts F, Neven F. Making queries tractable on big data with preprocessing: through the eyes of complexity theory. Proceedings of the VLDB Endowment, 2013, 6(9): 685–696
4. Chen G. Design and Analysis of Parallel Algorithms. Beijing: Higher Education Press, 2011
5. Cook S A. Deterministic CFL's are accepted simultaneously in polynomial time and log squared space. In: Proceedings of the 11th Annual ACM Symposium on Theory of Computing. 1979, 338–345
6. Mao R, Xu H L, Wu W B, Li J Q, Li Y, Lu M H. Overcoming the challenge of variety: big data abstraction, the next evolution of data management for AAL communication system. IEEE Communications

Magazine, 2015, 53(1): 42–47

7. Xiong J, Lu J, Tan F. Topology. Beijing: China Machine Press, 2013
8. Chávez E, Navarro G, Baeza-Yates R, Marroquín J L. Searching in metric spaces. ACM Computing Surveys, 2001, 33(3): 273–321
9. Zezula P, Amato G, Dohnal V, Batko M. Similarity Search: the Metric Space Approach. Springer Science & Business Media, 2006
10. Mao R, Zhang P H, Li X L, Liu X, Lu M H. Pivot selection for metric-space indexing. International Journal of Machine Learning and Cybernetics, 2016, 7(2): 311–323
11. Mao R, Miranker W, Miranker D P. Pivot selection: dimension reduction for distance-based indexing. Journal of Discrete Algorithms, 2012, 13: 32–46
12. Shi H M, Schaeffer J. Parallel sorting by regular sampling. Journal of Parallel and Distributed Computing, 1992, 14(4): 361–372
13. Valiant L G. Parallelism in comparison problems. SIAM Journal on Computing, 1975, 4(3): 348–355
14. Shiloach Y, Vishkin U. Finding the maximum. Merging and Sorting in a Parallel Computational Model, 1981, 2(1): 88–102
15. Chen G. Balanced (n, m)-selection networks. Computer Research and Development, 1984, 21(11): 9–22
16. Samet, H. Foundations of Multidimensional and Metric Data Structures. San Francisco: Morgan-Kaufmann, 2006
17. Hjaltason G R, Samet H. Index-driven similarity search in metric spaces. ACM Transactions on Database Systems, 2003. 28(4): 517–580
18. Uhlmann J K. Satisfying general proximity/similarity queries with metric trees. Information Processing Letter, 1991, 40(4): 175–179
19. Yianilos P N. Data structures and algorithms for nearest neighbor search in general metric spaces. In: Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms. 1993
20. Mao R, Liu S, Xu H L, Zhang D, Miranker D P. On data partitioning in tree structure metric-space indexes. In: Proceedings of the 19th International Conference on Database Systems for Advanced Applications. 2014, 141–155
21. Bozkaya T, Ozsoyoglu M. Indexing large metric spaces for similarity search queries. ACM Transactions on Database Systems, 1999, 24(3): 361–404
22. Sergey B. Near neighbor search in large metric spaces. In: Proceedings of the 21st International Conference on Very Large Data Bases. 1995
23. Ciaccia P, Patella M, Zezula P. M-tree: an efficient access method for similarity search in metric spaces. In: proceedings of the 23rd International Conference on Very Large Data Bases. 1997
24. Navarro G. Searching in metric spaces by spatial approximation. The VLDB Journal, 2002, 11(1): 28–46
25. Jagadish H V, Ooi B C, Tan K L, Yu C, Zhang R. iDistance: an adaptive B+-tree based indexing method for nearest neighbor search. ACM Transactions on Database Systems (TODS), 2005, 30(2): 364–397
26. Chavez E, Navarro G. Unbalancing: the key to index high dimensional metric spaces. Technical Report. 1999
27. Mao R, Xu W, Ramakrishnan S. Nuckolls G, Miranker D P. On optimizing distance-based similarity search for biological databases. In: Proceedings of the 2005 IEEE Computational Systems Bioinformatics Conference. 2005, 351–361
28. MacQueen J B. Some methods for classification and analysis of multivariate observations. In: Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability. 1967, 281–297
29. Ester M, Kriegel H P, Sander J, Xu X. A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceedings of the 2nd ACM SIGKDD Conference. 1996. 226–231
30. Culler D, Karp R, Patterson D, Sahay A, Schauser K E, Santos E, Subramonian R, von Eicken T. LogP: towards a realistic model of parallel computation. In: Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 1993, 1–12
31. Tu B B, Zou M, Zhan J F, Zhao X F, Fan J P. Research on parallel computation model with memory hierarchy on multi-core clusters. Chinese Journal of Computers, 2008, 31(11): 1948–1955
32. Ladner R E. The circuit value problem is log space complete for P. ACM SIGACT News, 1975, 7(1): 18–20
33. Pippenger N J. On simultaneous resource bounds. In: Proceedings of the 20th IEEE Annual Symposium on Foundations of Computer Science. 1979, 307–311
34. Chandra A K, Stockmeyer L J. Alternation. In: Proceedings of the 17th IEEE Annual Symposium on Foundations of Computer Science. 1976, 98–108
35. Goldschlager L M. Synchronous parallel computation. Dissertation for the Doctoral Degree. Toronto: University of Toronto, 1977

Guoliang Chen received his BS degree from Xi'an Jiaotong University, China in 1961. He is currently the dean of the College of Computer Science and Software Engineering of Shenzhen University, China and the dean of the School of Software, University of Science and Technology of China. Prof. Chen is a member of the Chinese Academy of Sciences, China. He has received more than 20 national and province level awards and authored about 20 books. His research interests include high performance computing and big data.

Rui Mao received his BS (1997) and MS (2000) in computer science from the University of Science and Technology of China, China and MS (2006) in statistics and PhD (2007) in computer science from the University of Texas at Austin, USA. After three years of working at the Oracle USA Corporation, he joined Shenzhen University (SZU), China in 2010. He is now an associate professor and an associate dean of the College of Computer Science and Software Engineering, SZU. His work on the pivot space model was awarded the SISAP 2010 Best Paper award. His research interest includes universal data management and analysis in metric space, and high performance computing.

Kezhong Lu received his BS and PhD degree in computer software and theory from the University of Science and Technology of China, China in 2001 and 2006, respectively. Currently he is a full professor in College of Computer Science and Software Engineering, Shenzhen University (SZU), China. Prior to current position, he worked as a lecturer from 2006 to 2007 and as an associate professor from 2007 to 2012 at SZU. His research interests include wireless sensor networks, parallel computing and big data.