An Optimal Balanced Partitioning of a Set of 1D Intervals

CHUAN-KAI YANG*
Department of Information Management
National Taiwan University of Science and Technology
No. 43, Section 4, Keelung Road, Taipei, 10607, Taiwan

Abstract

Given a set of 1D intervals and a desired partition number, this paper studies on how to make an optimal partitioning of these intervals, such that the number of intervals between the largest partition and smallest partition is minimal among all possible partitioning schemes. Though seemingly easy at the first glance, this problem has its difficulty due to the fact that an interval ``striding'' multiple partitions should be counted multiple times. Previously we have given an approximated solution to this problem by employing a *simulated annealing* approach [Yang & Chiueh, 2006], which could give satisfactory results in most cases; however, there is no theoretical guarantee on its optimality. In this paper, we propose a method that could both optimally and deterministically partition a given set of 1D intervals into a given number of partitions. This problem originates from dealing with the partition of a large volume data, where a more balanced partitioning could facilitate efficient *out-of-core* volume visualization faced in the computer graphics world. We will show that some types of *load balancing* problem could also be formulated as a balanced interval partitioning problem. As a result, seeking better solutions to this problem has become an important issue.

Keywords: Computational Geometry, Dynamic Programming, Interval Partitioning

1. MOTIVATION AND INTRODUCTION

How to partition a set of 1D intervals into balanced partitions is what to be discussed in this paper, and such a problem originally arisen from dealing with efficient volume visualization. Volume visualization, sometimes deemed a sub-field of computer graphics, aims to provide 2D images, or more specifically, 2D visualizations, from a so called *3D volume dataset*, which is normally composed of a stack of 2D images. Instead of distributing data values on 2D grid points like a 2D image, a volume dataset generalizes the distribution to 3D grid points. *Iso-surface extraction* is one of the most popular methods to visualize a volume dataset, where an *iso-value* is first given, and then the corresponding *iso-surface*, representing the surface intersecting with the given value, is extracted and represented by a set of *interpolated triangles*. These triangles could later be sent to a graphics card to be rendered into images. As technology advances, nowadays data acquisition devices tend to collect or generate data with higher and higher precision, thus leading to volume datasets of huge sizes. The visualization of a huge volume dataset

usually would cause problems as such a task normally needs to bring in the whole dataset into the main memory before any further processing could start. As the size of a huge volume dataset could be more than several GBytes, the scenario of an *out-of-core* processing arises [Cox & Ellsworth, 1997; Cox, 1997; Ueng et~al., 1997], which means the required memory for data processing is larger than a system could provide. To address this, data partitioning is one popular approach, such as the method proposed by Chiang et al. [Chiang et~al., 1998]. Since volume visualization is often the most important purpose that a dataset was collected or generated in the first place, it should be taken into account when dealing with data partitioning. In Chiang et al.'s approach, they partition a volume dataset *spatially* so that each resulting partition could be loaded into the main memory and processed individually without incurring any *virtual memory* overheads, namely, *memory swappings or pagings*. The extracted iso-surface is an aggregate result from each partition. Such an paradigm, though simple, has the following drawback: unless some clever pre-processing is employed, all the partitions need to be examined for iso-surface intersection. We argue that an different partitioning scheme may sometimes offer better processing efficiency. That is, instead of partitioning a volume dataset based on spatial information, we could decompose a dataset into *layers* based on the data values associated with the grid points, and each layer represents a portion of data bearing a particular value range. Such a partitioning has the advantage that for an iso-value query, *at most one layer is needed for extracting the iso-surface*, thus making it an attractive scheme. Furthermore, the result is a continuous or integrated representation, rather than a segmented one faced in the case of a spatial partitioning, where a *post stitching or merging* may be needed for smoothing or reducing the rendering traffic sent to graphics cards afterward. However, there are also two issues associated with this kind of partitioning approach. First, it is not suitable for a volume dataset with high data variation, such as some of the datasets obtained from CFD (computational fluid dynamics) simulation, since a value-based partitioning may lead to decompositions that are too fragmented, thus incurring extra representation overheads. Second, how to make a balanced partitioning is in fact more difficult that it seems, as will be detailedly explained in later sections. For the ease of ensuing discussion, we will refer to the first type of partitioning a *space-based* or *coordinate-based* scheme, while the second type of partitioning a *value-based* or *layer-based* scheme. Previously [Yang & Chiueh, 2006] have shown that how a balanced (with a slightly different definition of *balance*) partitioning could be achieved by a *simulated annealing* approach. Though efficient, such a *soft optimization* method can neither guarantee the optimality nor provide a theoretical analysis of the result. In this work, we provide a *dynamic programming* approach that could find an optimal solution to this problem.

Applications of this balanced interval partitioning problem are not limited to volume visualization only, as some types of *load balancing* problems could also be formulated as such a problem. For example, considering the following *load balancing* problem, where a number of activities are to be held in a hall, and each activity has its own specific starting time and duration. Now assuming there are only a fixed number of janitors available to assist these activities, a reasonable arrangement is to partition these activities into groups so that each janitor is responsible for roughly the same number of activities. Some balanced task scheduling problems could also be formulated in a similar fashion, thus making solutions to this problem more important.

The rest of the paper is organized as the following. Section 2 reviews the literature related to this work. Section 3 details our proposed algorithm. Section 4 analyzes the time complexity of our proposed approach. Section 5 concludes this work and points out some potential future directions.

2. RELATED WORK

As explained in the Introduction Section, the problem of balanced 1D interval partitioning comes from the field of volume visualization, and the visualization method involved is called *iso-surface extraction*, proposed by Lorensen et al. [Lorensen & Cline, 1987]. An iso-surface extraction process identifies the surface from a volume dataset that intersects with a given iso-value. Such an iso-surface query is basically a *stabbing query* and the original concept came from Edelsbrunner [Edelsbrunner, 1980] and McCreight [McCreight, 1980] in the *computational geometry* community, and there are ways to speed up this type of queries, such as the proposal of using *interval trees* [Cignoni et~al., 1997]. However, as a volume dataset could get larger and larger, or sometimes even larger than a machine's physical memory, the so called *out-of-core* situation would arise, and some dataset partitioning scheme may seem unavoidable. Chiang et al. [Chiang et~al., 1998] partitions a volume dataset into *meta-cells* using a *KD-tree* [Bentley, 1975] decomposition to make sure that each meta-cell could be loaded into the memory. Ma et al. proposed a similar scheme [Ma et~al., 2001] by *spatially* decomposing a volume dataset into *layers*, which should be noted to bear a different meaning from ours. Yang et al. [Yang & Chiueh, 2006] solves a similar interval partitioning problem by a simulated-annealing approach. By treating the number of intervals in a partition as a random variable, their approach to achieve a balanced partitioning is to minimize the variance of such random variables. Note that, with a straightforward modification, the minimization of variance could be easily converted to the minimization of the size difference between the largest and the smallest partitions.

3. A BALANCED LAYER PARTITIONING ALGORITHM

We formally define what our target problem is and propose our solution to it in this Section.

3.1 Problem Statement

Given a set of arbitrary 1D intervals, represented by I, and a partition number, represented by M, we want to partition I into M partitions such that in terms of number of intervals, the size difference between the largest and smallest partitions, is minimal.

3.2 A Divide-and-Conquer Approach

To illustrate why such a problem is more difficult than it seems, we first give a naive divide-and-conquer approach. Assume that $M = 2$, apparently the problem becomes to find one *cut* to make two partitions such that the difference between the numbers of intervals intersecting with each partition is minimal. Note that as some intervals

may intersect with the cut itself, these intersecting intervals will be counted twice, i.e., they will contribute to both partitions. In other words, making a cut will ``interfere" with the number of intervals intersecting with the affected partitions, and this is exactly why this problem is intricate. To proceed, we could adopt the following procedure which is intrinsically a *binary search* algorithm. The lower bound and upper bound of all intervals are first determined, denoted by *Min* and *Max*, respectively, and then we calculate the middle point, or the initial position for the cut, denoted by *Mid*, where $Mid = \frac{(Min+Max)}{2}$. Let the number of intervals intersecting with the closed interval [*Min*, *Mid*] be *Left* and similarly the number of intervals intersecting with the closed interval [*Mid*, *Max*] be *Right*, where [a, b] denotes the set of real numbers:$\{x|a \leq x \leq b\}$. If $Left = Right$, then we are done. On the other hand if $Left > Right$, the new cutting position is set to $\frac{(Min+Mid)}{2}$. Finally for the remaining case, the new cutting position is set to $\frac{(Mid+Max)}{2}$. Such a procedure could be carried out until the difference between *Left* and *Right* could not be further reduced.

Now let us turn to the case where $M = 4$. A naive or straightforward approach would be to first make two partitions, denoted by $Left_{Part}$ and $Right_{Part}$, respectively, as in the case where $M = 2$. Then we further partition $Left_{Part}$ into $Part_1$ and $Part_2$ by making one cut, denoted by $Left_{cut}$. Similarly $Right_{cut}$ denotes the cut to partition $Right_{Part}$ into $Part_3$ and $Part_4$. Now the problem arises. As the number of intervals intersecting with $Left_{cut}$and $Right_{cut}$ may be quite different, and the former will contribute to $Part_1$ and $Part_2$, while the latter to $Part_3$ and $Part_4$, thus leading to an overall imbalanced partitioning. Therefore a divide-and-conquer approach like this would fail. However, such an approach may be used as a starting configuration for further adjustment to achieve a balanced partitioning, such as the *simulated annealing* approach proposed by Yang et al. [Yang & Chiueh, 2006].

3.3 A Dynamic Programming Approach

To solve this problem deterministically, in this paper, we resort to a *dynamic programming* approach, which is very similar to the one used to solve the *matrix chain multiplication* problem [Cormen et~al., 2001]. The *recursive definition* of the dynamic programming is shown in Equation~1 while the *C-like* pseudo code is listed in Figure 1, where some terms should be explained in advance. First, *M* denotes the desired number of partitions, and *Num_I* is the number of intervals, while *N* represents the number of distinct 1D points coming from the corresponding end points of these *Num_I* intervals. To make *M* partitions, all we need to do is to determine $M - 1$ cutting positions. However, it does not take long for us to figure out that there are in fact at most $2N - 1$ positions need to be considered. That is, the positions of the *N* points, together with the $N - 1$ midpoints between any two adjacent points. The configuration produced by making any other cutting position could be shown to be equivalent to placing a cut at one of the above $2N - 1$ positions. For the ease of ensuing discussion, we sort these $2N - 1$ positions from left to right, and will number them from 1 to $2N - 1$. Second, the term $Part(i, j, k)$ stands for the best solution of making k partitions of the region starting from the *i-th* position to the *j-th* position. Therefore $Part(i, j, k)_{.part\_pos}$ represents the cutting position to further decompose this region into two partitions, say $Left_{Part}$ and $Right_{Part}$, and $Part(i, j, k)_{.part\_num}$ is the number of partitions to be made in

$Left_{Part}$, thus leaving $k - Part(i,j,k)_{.part\_num}$ partitions to be made in $Right_{Part}$. Furthermore, as the names suggest, $Part(i,j,k)_{.min}$ and $Part(i,j,k)_{.max}$ represent the numbers of intervals in the minimal and maximal partitions, respectively, and by definition, the difference between these two numbers is minimal among all possible partitioning schemes to make *k* partition in this region. Finally, *MIN* and *MAX* are two macros to extract the smaller and larger number, respectively, when two numbers are given. As mentioned previously, the basic flow of this dynamic programming approach follows the *matrix chain multiplication* problem closely by building up the calculations of $Part(i,j,k)$ in a *bottom-up* manner.

$$Part(i,j,k) = \begin{cases} 0 & if\ i > 0 \\ query(i,j) & if\ k = 1\ and\ i \le j \\ min_{i \le i_1 \le j, 1 \le k_1 < k} max_{size}(i,j,k,i_1,k_1) & otherwise \end{cases} \quad (1)$$

where $max_{size}(i,j,k,i_1,k_1)$ is obtained by calculating

$$max\{Part(i,i_1,k-k_1)_{.max}, Part(i_1,j,k_1)_{.max}\} - min\{Part(i,i_1,k-k_1)_{.min}, Part(i_1,j,k_1)_{.min}\} \quad (2)$$

```
for(k=2;k<=M;k++) {
    for(len=1;len<=2*N-1;len++) {
        for(i=1;i<2*N-l;i++) {
            j=i+len;
            diff_temp=Num_I*M;
            for(k1=1;k1<k;k1++) {
                for(i1=i;i1<=j;i1++) {
                    min_temp=MIN(Part(i,i1,k-k1).min,Part(i1,j,k1).min);
                    max_temp=MAX(Part(i,i1,k-k1).max,Part(i1,j,k1).max);
                    if(max_temp-min_temp<diff_temp) {
                        diff_temp=max_temp-min_temp;
                        Part(i,j,k).part_pos=i1;
                        Part(i,j,k).part_num=k-k1;
                        Part(i,j,k).min=min_temp;
                        Part(i,j,k).max=max_temp;
                    }
                }
            }
        }
    }
}
```

Fig. 1. *The pseudo code of a* dynamic programming *approach.*

The code in Figure 1 presents a *five-loop* structure, where the first loop increases the partition number, i.e., the term *k* in $Part(i,j,k)$, while the second loop controls the *length*, represented as *len* in the code, of each involved partition, i.e., how far apart *i* and *j* are in $Part(i,j,k)$. Note that in our code, *j* is never less than *i*, and thus is always set to

be $i + len$. The third loop enumerates all possible positions of *i* in $Part(i,j,k)$, together with the previously enumerated length, therefore the region to be partitioned in $Part(i,j,k)$ is completely determined. The fourth and fifth loops are used to divide the partition of $Part(i,j,k)$ into two smaller partitions where the left one, denoted by $Part(i,i_1,k-k_1)$, and the right one, denoted by $Part(i_1,j,k_1)$, are compared to find the best values of i_1 and k_1 such that the combined results from both partitions could form the most balanced partitioning, as far as $Part(i,j,k)$ is concerned. The search for optimality is to minimize the difference, denoted by *diff_temp*, between the largest partition and the smallest partition, and is initialized to be *Num_I∗M*, which is apparently an upper bound that could not be exceeded.

The initialization of the pseudo code is shown in Figure 2, where the *bottommost* values of $Part(i,j,k)$ are laid down, and such an initial setting should be self-explanatory, except the term $query(i,j)$. Here $query(i,j)$ is used to record the number of intervals intersecting with the region defined by the *i-th* and *j-th* cutting positions, under the constraint that $i \leq j$.

```
for(i=1;i<2*N;i++) {
    for(j=i;j<2*N;j++) {
        Part(i,j,1).min=part(i,j,1).max=query(i,j);
        Part(i,j,1).part_num=0;
        Part(i,j,1).part_pos=i;
    }
    for(i=1;i<2*N;i++)
        for(k=2;k<=M;k++) {
            Part(i,i,k).min=part(i,i,k).max=query(i,i);
            Part(i,i,k).part_num=0;
            Part(i,i,k).part_pos=i;
        }
```

Fig. 2. *The initialization of the pseudo code in Figure 1.*

Note that, compared with the naive divide-and-conquer approach, where a cut could introduce side-effects and therefore may complicate the partitioning, this dynamic programming algorithm takes into account the cutting overheads by counting values in the form of $Part(i,j,k)$, through which the numbers of intervals intersecting with the *cuts* will be handled implicitly and correctly.

4. COMPLEXITY ANALYSIS

In this section, we analyze the time complexity of our proposed algorithm, which is composed of two parts, as shown in Figure 1 and Figure 2. It is evident that the first part demands an $\boldsymbol{O}(N^3M^2)$ time complexity. On the other hand, the time complexity of the second part depends on how we calculate $query(i,j)$ each time, given different combinations of *i* and *j*. To have a more efficient implementation, we choose to calculate and store all possible $query(i,j)$ in advance, therefore in Figure 2 the query result of $query(i,j)$ could be answered with a constant time complexity. As for the total time complexity involved to pre-calculate all possible values of $query(i,j)$, it could be

shown by the following argument that it is bounded by $\boldsymbol{O}(N^2)$. To begin with, we first define $left(i)$ as the number of intervals use the *i-th* position as their left end point. All $left(i)$ values could be calculated in $\boldsymbol{O}(N)$ time as we could just enumerate each interval and collect required information accordingly. Secondly, it is not difficult to see why the equality $query(i,j) = query(i,j-1) + left(i)$ holds, as the inclusion of the *i-th* position brings in exactly $left(i)$ new intervals, with respect to the $query(i,j-1)$, intervals that are already counted. This indicates that the calculation of $query(i,j)$ could be carried out by an iterative computation. However, such an iteration requires the value of $query(i,j)$, which is exactly the form of a *stabbing query* and could be answered in $\boldsymbol{O}(logN)$ time by building an *interval tree* at the pre-processing time with an $\boldsymbol{O}(NlogN)$ total time complexity [Cignoni et~al., 1997]. As the forms of $query(i,j)$ are only calculated $\boldsymbol{O}(N)$ times, the total time complexity in Figure 2 is therefore $\boldsymbol{O}(N^2)$. Consequently, the overall algorithm complexity is dominated by the first part, and thus is $\boldsymbol{O}(N^3M^2)$. Note that the associated *space complexity*, evidently dominated by the memorization of $Part(i,j,k)$, is $\boldsymbol{O}(N^2M)$.

5 CONCLUSIONS AND FUTURE WORK

We have proposed a deterministic algorithm to find an optimal solution to the interval partitioning problem. There are two future directions that we plan to pursue. First, how to determine the most suitable number of partitions for a given dataset is worth studying, as a *partitioning cut* could cause extra intersection overhead, and therefore too many partitions may lead to an overall inefficient scheme for data storage. Potential solutions may include some hierarchical data structures or hybrid schemes of space-based and value-based approaches. Second, we must admit that an overall $\boldsymbol{O}(N^3M^2)$ time complexity is still too high, especially when we are faced with huge datasets where the associated values of *N* larger than one million are not uncommon. There exist techniques to reduce the time complexity of the *matrix chain multiplication* problem from $\boldsymbol{O}(N^3)$ [Cormen et~al., 2001] to $\boldsymbol{O}(NlogN)$ [Hu & Shing, 1982; Hu & Shing, 1984] by employing a *divide-and-conquer* approach, and due to its similarity to our problem, we are currently surveying the possibility for time complexity reduction in this regard. Furthermore, space complexity reduction is also a very important concern, as a huge dataset may render the whole algorithm inoperable, as the required memory cannot be provided.

REFERENCES

[Bentley, 1975] Bentley, J.~L. (1975). Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509--517.

[Chiang et~al., 1998] Chiang, Y., Silva, C., and Schroeder, W. (1998). Interactive Out-of-Core Isosurface Extraction. In *IEEE Visualization '98*, pages 167--174.

[Cignoni et~al., 1997] Cignoni, P., Marino, P., Montani, C., Puppo, E., and Scopigno, R. (1997). Speeding Up Isosurface Extraction Using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158--170.

[Cormen et~al., 2001] Cormen, T.~H., Leiserson, C.~E., and Rivest, R.~L. (2001). *Introduction to Algorithms*. MIT Press, second edition.

[Cox, 1997] Cox, M. (1997). Managing Big Data for Scientific Visualization. *ACM SIGGRAPH '98 Course*.

[Cox and Ellsworth, 1997] Cox, M. and Ellsworth, D. (1997). Application-Controlled Demand Paging for Out-of-core Visualization. *IEEE Visualization '97*, pages 235--244.

[Edelsbrunner, 1980] Edelsbrunner, H. (1980). Dynamic Data Structures for Orthogonal Intersection Queries. *Technical Report Report F59, Inst. Informationsverarb.,Tech. University Graz*.

[Hu and Shing, 1982] Hu, T.~C. and Shing, M.~T. (1982). Computation of matrix chain products. part i. *SIAM Journal on Computing*, 11(2):362--373.

[Hu and Shing, 1984] Hu, T.~C. and Shing, M.~T. (1984). Computation of matrix chain products. part ii. *SIAM Journal on Computing*, 13(2):228--251.

[Lorensen and Cline, 1987] Lorensen, W.~E. and Cline, H.~E. (1987). Marching Cube: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, 21(4):163--169.

[Ma et~al., 2001] Ma, K., Abla, G., and Lum, E. (2001). Layer Data Organization for Visualizing Unstructured-Grid Data. In *Proceedings of SPIE, Visual Data Exploration and Analysis VIII*, pages 111--120.

[McCreight, 1980] McCreight, E.~M. (1980). Efficient Algorithms for Enumerating Intersecting Intervals and Rectangles. *Technical Report Report CSL-80-9, Xerox Palo Alto Res. Center*.

[Ueng et~al., 1997] Ueng, S.~K., Siborski, K., and Ma, K.~L. (1997). Out-of-Core Streamline Visualization on Large unstructured meshes. *ICASE Report*.

[Yang and Chiueh, 2006] Yang, C. and Chiueh, T. (2006). Integration of Volume Decompression and Out-of-Core Iso-Surface Extraction from Irregular Volume Data. *The Visual Computer*, 22(4):249--265.