Using Bitmap Index for Interactive Exploration of Large Datasets[*]

Kesheng Wu[†] Wendy Koegler[‡] Jacqueline Chen[‡] and Arie Shoshani[†]

Abstract

Many scientific applications generate large spatio-temporal datasets. A common way of exploring these datasets is to identify and track regions of interest. Usually these regions are defined as contiguous sets of points whose attributes satisfy some user defined conditions, e.g. high temperature regions in a combustion simulation. At each time step, the regions of interest may be identified by first searching for all points that satisfy the conditions and then grouping the points into connected regions. To speed up this process, the searching step may use a tree based indexing scheme, such as a kd-tree or an octree. However, these indices are efficient only if the searches are limited to one or a small number of selected attributes. Scientific datasets often contain hundreds of attributes and scientists frequently study these attributes in complex combinations, e.g. finding regions of high temperature yet low shear rate and pressure. Bitmap indexing is an efficient method for searching on multiple criteria simultaneously. We apply a bitmap compression scheme to reduce the size of the indices. In addition, we show that the compressed bitmaps can be used efficiently to perform the region growing and the region tracking operations. Analyses show that our approach scales well and our tests on two datasets from simulation of the autoignition process show impressive performance.

1 Introduction

One method available for scientific data exploration is to identify and track regions of interest [1, 7, 14]. The goals are to identify regions for further analysis and to visualize their evolution. In a spatial-temporal dataset, such as the results from a combustion simulation [5, 6], the physical phenomena to be studied can be described as a number of scalar or vector fields on a domain, e.g. temperature and velocity throughout a volume of air. Regions of interest are then identified as meeting some conditions based on these properties, e.g. regions in the volume where the air is very hot. Datasets can have hundreds of attributes such as pressure, concentrations, etc. Regions of interest may be defined based on any of the attributes in the dataset as well as combinations of different attributes. A scientist usually has to explore a number of different conditions before proceeding to subsequent analysis steps.

The full tracking process, called feature tracking, is illustrated by Silver and Wang [14]. The spatio-temporal dataset is organized on the top level according to time. For each time step, snap shots of the fields are recorded. These fields are usually discretized and recorded as attribute values on grid points or cells. In [14], a region of interest was defined to be a connected region where an attribute is above a specified threshold value. A region of this type is also known as the thresholded region. The thresholded regions were identified in each time step and then tracked through time by comparing overlaps of regions from consecutive time steps. An octree was used to facilitate the visualization and identification of regions of interest. Since the octree is based on a partition of space, searching for points that satisfy conditions on attribute values typically requires one to examine a large portion of the tree. For this reason, identifying regions of interest using an octree is considered an $\mathcal{O}(N)$ process, where N is the total number of points. A number of alternative approaches scale better, as we shall discuss next.

For visualization purposes, identifying a thresholded region is equivalent to identifying its boundary. Therefore, we may choose to use an isocontouring algorithm to determine the boundary instead of identifying the entire region. There are efficient algorithms for generating isocontours for visualization. For example, the NOISE algorithm is shown to have the worst case complexity of $\mathcal{O}(\sqrt{N}+s)$, where s is the size of the isocontour [8]. More recent researches have demonstrated that the optimal complexity of $\mathcal{O}(s)$ is achievable with suitable preprocessing [4, 11]. Although potentially faster than the thresholded region algorithm [14], isocontouring algorithms only identify points on the boundary. Additional work is required to identify all the interior points.

The process of identifying regions of interest can be divided into two steps. The first step is to identify points

[*] This work was supported by the Director, Office of Energy Research, Office of Laboratory Policy and Infrastructure Management, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

[†] Lawrence Berkeley National Laboratory, Berkeley, CA. Email: {KWu,AShoshani}@lbl.gov.

[‡] Sandia National Laboratories, Livermore, CA. Email: {WKoegler, JHChen}@ca.sandia.gov.

that satisfy the specified conditions and the second step is to group these points into connected regions. In later discussions, we refer to the first step as the searching step and the second as the region growing step. After the regions have been identified in each time step, the last step of feature tracking is to determine the relationship among the regions between time steps. We refer to this step as region tracking.

One important limitation of the searching schemes employed by the thresholded region algorithm [14] and the isocontouring algorithms [4, 8, 11], is that they are only designed for regions of interest defined on one attribute. To remove this limitation, a number of multidimensional indexing schemes can be applied to speed up the search step. For example, Shi and Jaja (2002) applied a packed Hilbert R-Tree to an earth science dataset [12]. In database terms, each grid point is a record, and the spatial coordinates and the associated fields are attributes of the record. The algorithm used by Shi and Jaja can process multi-attribute conditions. Its time complexity is $\mathcal{O}(v)$, where v is the volume of the regions of interest. However, the packed Hilbert R-Tree and most of the multidimensional indexing schemes either suffer from the "curse of dimensionality" or are only efficient when queries involve all attributes. The "curse of dimensionality" refers to the fact that their effectiveness rapidly deteriorate as number of attributes increases. Most of them are only useful when the number of attributes are small, say ≤ 10.

Most multidimensional indexing schemes reorder the records according to their attribute values. Although this reordering is crucial to achieve efficient search operations, it causes the region growing procedures to be very slow. The reordering of records according to values destroys spatial relation among the neighbors. This relation essentially has to be recovered during region growing. For example, in Shi and Jaja's study [12], the search step is quite fast (~ 0.1 seconds), but the region growing step is much slower (2–4 seconds).

Among the multidimensional indexing schemes, the bitmap indexing scheme is known to not suffer from the "curse of dimensionality" and is especially efficient when the queries do not involve all of the attributes [9, 19]. More importantly, the bitmap index does not require one to reorder the records in a particular way, thus we can keep the records ordered to preserve neighborhood relation for the region growing step. In this paper, we apply a compressed bitmap index to the feature tracking problem. Applying the bitmap index to the searching step is relatively straightforward. The crucial point of this paper is that the compressed bitmaps can also be used efficiently for region growing and region tracking. Our complexity analyses show that we achieve the optimal complexity in all three steps of the feature tracking process. Tests on two sets of data from autoignition process verifies the complexity analyses. On a 33 GB dataset, it takes on the average 10 seconds to identify and track regions of interests defined on four different attributes. Even in the worst cases, it takes less than a minute.

The remaining of this paper is organized as follows. In Section 2, we review the basic concepts of bitmap indexing and give the optimal scheme for the searching step. In Section 3, we review the compression scheme used and give the basic reasons that it can perform bitwise logical operations efficiently. The most important point of this section is that we can represent regions of interest compactly using $\mathcal{O}(s)$ words. Since the optimal bitmap index allows us to read only one compressed bitmap to answer a threshold condition, this searching step has a complexity of $\mathcal{O}(s)$ in both space and time. In Section 4, we discuss how the region growing process can be implemented in the same complexity. We also show experimental evidence to support the theory. The region tracking procedure used in our tests is discussed in Section 5. The overall performance characteristics of the bitmap based feature tracking algorithm is discussed in Section 6. In section 7, we summarize the current work on uniform grids and discuss how we may extend the algorithms to irregular grids and adaptive grids.

2 Bitmap indexing

The bitmap index scheme has a long history in the database field [9, 10, 2]. It is very efficient for data warehousing applications where the data records are read-only and the queries usually produce a large number of data items [2, 3, 19]. The feature tracking problem is similar to the data warehousing applications. In both cases, the data are read-only and the results of queries are usually large. Another reason for considering a bitmap index is that it does not require one to reorder the records. We can simply leave the records in the order used in the computational procedure that generated the data. Compared to those that require reordering, it requires less time to create a bitmap index. In addition, the existing order of records is well-suited for the region growing and region tracking steps.

A bitmap index can be considered as a set of precomputed answers to some simple conditions. Let X be an integer attribute with values in the range of 0 to 9. The range-encoded bitmap index precomputes all answers to the query of the form "$X \geq p$", where p is an integer [2, 3]. Let N denote the number of records, each bitmap contains N bits of 0s and 1s. For the bitmap representing "$X \geq 1$", the ith bit is 1 if the attribute X in the ith record is greater than or equal to 1, otherwise the bit is 0.

In the bitmap index for X, there are 10 bitmaps each corresponding to one distinct value of X, 0 to 9. For any user specified condition of the form "$X \geq p$", to find all the points satisfying the condition, we only need to read one of the bitmaps. Clearly, the size of this bitmap determines the time and space complexity of this searching step. In the next section, we discuss the use of compression to reduce bitmap sizes. In the remainder of this section, we discuss how to deal with continuous values and conditions involving multiple attributes.

A bitmap index is usually defined for one attribute, however it is easy to process conditions involving more than one attribute. For example, with two range-encoded bitmap indices for both X and Y, to answer the query "$X \geq p_x$ and $Y \geq p_y$" one simply uses the indices for X and Y to first get the partial solutions to "$X \geq p_x$" and "$Y \geq p_y$", and then performs a bitwise AND operation on the two partial solutions.

The bitmap indexing scheme can also be used for attributes with continuous values. In scientific applications, these attributes are typically represented as floating-point values. The most straightforward option is to build one bitmap for each distinct value. This option ensures that all possible threshold conditions can be answered by reading one bitmap, however, it may generate a compressed bitmap index that is larger than a B-tree index. To reduce the size of the bitmap index, we usually bin the floating-point values [13, 18]. For example, assuming an attribute Z can be of any real value between zero and one, we may build a range-encoded bitmap index for the following set of conditions, $Z \geq 0$, $Z \geq 0.1$, $Z \geq 0.2$, ..., $Z \geq 0.9$. This corresponds to having 10 bins with bin boundaries 0, 0.1, and so on. In this case, the performance of searching step depends critically on the selection of the bin boundaries. For example, if the threshold is 0.85, i.e., the query condition is $Z \geq 0.85$, all records satisfying $Z \geq 0.9$ definitely satisfy the query condition, but all the records satisfying $Z \geq 0.8$ but not $Z \geq 0.9$ have to be examined. If commonly used conditions involving an attribute are known, it is possible to devise an set of optimal boundaries to minimize the number of records to be examined. In the cases where a human explorer is expected to use the system, there are certain heuristics for maximizing the chance that a user query fall on a bin boundary. For example, a human explorer is more likely to enter a condition like "$Z \geq 0.1$" than "$Z \geq 0.1001$", unless of course 0.1001 happens to be a magic number in the particular application. In the autoignition datasets, most of the attributes have real values and the bin boundaries are typically of the form $10^{-8}, 2 \times 10^{-8}, \ldots, 10^{-7}, 2 \times 10^{-7}$, and so on.

In general, users can select their own preferred bin boundaries when creating the bitmap indices. By selecting bin boundaries that fit a specific application, conditions involved in identifying thresholded regions can be answered by retrieving one bitmap from a bitmap index. This process is expected to take $\mathcal{O}(N)$ space and $\mathcal{O}(N)$ time. However, it can be significantly reduced if the bitmaps are compressed. Most of the commonly used bitmap compression schemes are based on the run-length encoding. If the physical problem has continuity in space, then preserving the neighborhood relation among the records tend to also improves the compressibility of the bitmaps. For a simple threshold condition on one attribute, we can show that the size of the compressed bitmap is $\mathcal{O}(s)$, where s is the size of the boundaries of the thresholded regions.

3 Bitmap compression

As mentioned in the previous section, compression is the crucial technique that reduces the complexity of the search operations when using the bitmap index. In this section, we demonstrate that on a uniform grid, the compressed bitmap size is proportional to the size of the boundaries of the regions of interest, i.e., $\mathcal{O}(s)$. To find thresholded regions using a range-encoded bitmap index we simply retrieve an appropriate bitmap from the bitmap index. If the size of this bitmap is $\mathcal{O}(s)$ bytes, the complexity of the search operation is theoretically comparable to the best algorithm for identifying the boundaries (iso-contour) of these regions. The following analyses are for 3D uniform grids and the illustrations use a 2D grid.

On a uniform 3-D grid, each grid point can be marked with its indices along x, y, and z directions, i, j, and k. A simple way of ordering these grid points is the raster scan ordering. Let the grid size be $n_x \times n_y \times n_z$, and the indices i, j, and k go from 0 to n_x-1, n_y-1 and n_z-1 respectively. The global index of a grid point at (i, j, k) is $i+j*n_x+k*n_x*n_y$ using the raster scan order. A number of connected regions can be recorded by marking whether a point belongs to one of the regions. Using the bitmap index, the solution from the searching step is a bitmap of N bits where the points inside the regions of interest are marked with 1.

Let's define the *order line* to be a line going through all the grid points in order 0, 1, ..., $N-1$. The boundary of the regions will cut the order line into many line segments. A segment that contains grid points within the regions corresponds to a group of 1s in the bitmap. A group of consecutive identical bits is called a fill. A fill with only 0s is called a 0-fill and a fill with only 1s is called a 1-fill. The 0-fills correspond to line segments outside of the regions of interest and the 1-fills correspond to line segments inside the regions. If the points of the regions fall on s_l line segments, the bitmap representing the regions

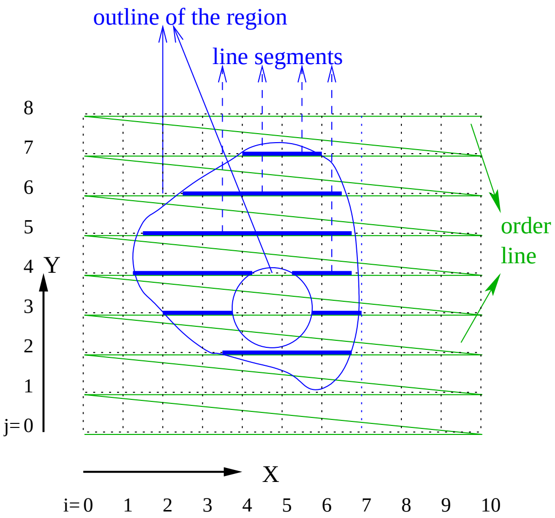


Figure 1: A 2-D regular grid (11 × 9).

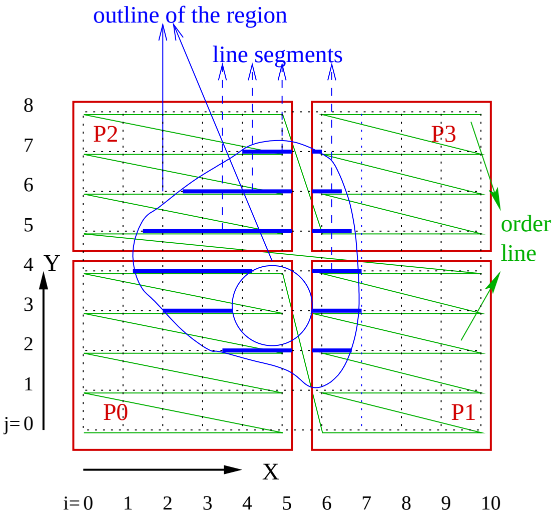


Figure 2: A 2-D regular grid (11 × 9) partitioned into four blocks.

contains no more than s_l 1-fills separated by no more s_l-1 0-fills. There might be two additional 0-fills, one at the beginning and one at the end of the bitmap. Altogether there is a maximum of $2s_l$+1 fills. Using the run-length encoding, each fill can be represented in one word. A total of at most $2s_l$+1 words may be used. Many of the commonly used bitmap compression schemes are based on the run-length encoding, therefore using one of these schemes the compressed bitmap should use $\mathcal{O}(s_l)$ words. Figure 1 shows an example of a 2-D grid. Only one connected region is shown, but it is not a convex region. In this example, there are eight line segments, $s_l = 8$, and 17 fills in the bitmap representing the region. The run-length encoding of this bitmap is shown in Figure 3.

Figure 1	26*0, 3*1, 6*0, 2*1, 2*0, 1*1, 6*0, 3*1, 1*0, 1*1, 6*0, 5*1, 7*0, 4*1, 8*0, 3*1, 15*0
Figure 2	16*0, 2*1, 2*0, 2*1, 4*0, 3*1, 11*0, 1*1, 4*0, 1*1, 4*0, 1*1, 6*0, 4*1, 3*0, 3*1, 4*0, 2*1, 6*0, 1*1, 4*0, 1*1, 4*0, 1*1, 9*0

Figure 3: The run-length encoding of the bitmaps representing the regions shown in Figures 1 and 2.

Figure 1	0000001C 0640E81F 00F00E00 00000000
Figure 2	0000661C 0021081E 1C302108 00000000

Figure 4: The Word-Aligned Hybrid (WAH) encoding of the bitmaps representing the regions in Figures 1 and 2. Code words are expressed as hexadecimal digits. In both cases, the last word contains only six useful bits.

To improve the efficiency of logical operations on the compressed bitmaps, we use a compression scheme called the Word-Aligned Hybrid (WAH) code [15, 16, 17]. The WAH scheme differs from the basic run-length encoding in two ways. First, it only records long groups of 0s or 1s using the run-length encoding, the short groups are represented literally. Second, it requires the groups to be of certain size so that operations on the compressed data are efficient. If a word contains 32 bits, WAH first groups the bits of a bitmap into 31-bit groups. Consecutive groups with only 0s or 1s are encoded in one word called the fill word. The remaining groups are represented literally, where the 31 bits plus a flag bit are stored in a 32-bit word. In the worst case, the $2s_l$+1 fills might take $4s_l$+3 words in WAH because each fill may turn into one literal word plus a fill word and the last few (N modulo 31) bits require a word to store. Typically, it takes less space than the run-length encoding scheme, especially if many of the groups are small. For the example shown in Figure 1, WAH encodes all bits in literal words and only four words are needed instead of 17. The WAH compressed version is shown in Figure 4. In this case, there are only six useful bits in the last word.

We say a point in a region is *exposed* if one of its neighbors is outside of the region. We define the *boundary size* s of a region to be the number of points exposed. For most line segments, only their two end points are exposed. This suggests that the boundary size is about twice the number of line segments, $s \sim 2s_l$. The example shown in Figure 1 has 16 points exposed to the outside while it has eight line segments. A line segment can have more than two points exposed if the whole or part of the line segment is exposed. For example, the line segment with $j = 2$ in Figure 1 has three exposed points. It may contribute less than two points if it has only one point. For regions with complex outlines, we would expect s to be slightly greater than $2s_l$. A connected region with many single-point line segments along the x-axis should be rare. In short, an algorithm with complexity of $\mathcal{O}(s_l)$ also has a complexity of $\mathcal{O}(s)$.

The datasets we used were computed on a parallel machine and their grids were distributed to many processors, one piece of the grid, or a block, to each processor. In this case, the points and the blocks form a two-level grid. Within each level, the points are numbered in the raster scan order, see an illustration in Figure 2. This makes the order line more complex and increases the number of line segments inside a region, and therefore increase the size of the compressed bitmaps. The region and the grid shown in Figure 2 are exactly the same as shown in Figure 1, except that the domain has been partitioned into four nearly equal-size blocks. By partitioning the grid into four blocks, the number of line segments increases from 8 to 12. In the bitmap, the number of fills increases from 17 to 25. We may reduce the size of the bitmaps by reordering the grid points to follow the simple raster scan order, however we are not sure whether the extra preprocessing time is worthwhile at this time.

Logical operations on WAH compressed bitmaps can work directly on the compressed data and generate compressed answers. Because of this, the time to perform these operations is proportional to the sizes (bytes) of the bitmaps involved [15, 16, 17]. Proving this in detail is beyond the scope of this paper, the following example illustrates the main points of the proof. Let A be the bitmap representing the region shown in Figure 2 and B be the bitmap representing a region the occupying the first block only. The two compressed bitmaps are shown in Figure 5. The last words for both bitmaps contain six 0 bits and are not shown to save space. In B, the 62 0-bit in the middle is represented by a WAH code word `80000002`. The first bit is 1, indicating it is a fill word. The second bit is 0, indicating all the bits of the fill are 0. The remaining 30 bits contains the integer 2, indicating the fill consists of 2 31-bit groups. In other word, the fill is 62 bits long. The procedures to perform bitwise logical operation can directly read these code words and generate code words of the results. In general, to generate one word of a result, at least one word of the two operands is consumed. For instance, the first code word of $A\&B$ is generated by using the bitwise AND operator on the first code words from A and B. One word from A and one word from B are consumed in this case. The second code word in $A\&B$ is the same as that of B because it is a fill of 0 bits. There is no need to examine the two corresponding code words of A. In this case, one word from B and two words from A are consumed to generate one word of the result. If there are n_A words in A and n_B words in B, in the worst case, a bitwise logical operation may generate $n_A + n_B$ words in the result. Because of this linear relation, bitwise logical operations on WAH compressed bitmaps have the potential to outperform the same operations on the uncompressed bitmaps. Our earlier tests demonstrated that this is indeed the case [15, 16, 17]. Overall, using the WAH compression scheme not only reduces the storage requirement, but also improves the computational efficiency.

A	`0000661C`	`0021081E`	`1C302108`
B	`7FFFFFFE`	`80000002`	
$A\&B$	`0000661C`	`80000002`	
$A \mid B$	`7FFFFFFE`	`0021081E`	`1C302108`

Figure 5: Examples of bitwise logical operations on WAH compressed bitmaps.

An additional benefit of using WAH compression is that we can generate the bitmap indices efficiently. Using WAH compression, it is possible to only insert bits that are 1 when creating a bitmap index. In most cases, we found that generating the (equality encoded) bitmap index[1] has the computational complexity of $\mathcal{O}(N \log(b))$, where b is the number of bitmaps generated. This is significantly better than the complexity of generating an uncompressed bitmap index, $\mathcal{O}(Nb)$. It is also better than multidimensional indexing schemes that require sorting. These schemes have a complexity of at least $\mathcal{O}(N \log(N))$.

4 Region growing

The searching step generates a bitmap representing all grid points satisfying the specified conditions. In previous sections, we have shown that this bitmap can be produced and stored efficiently. For visualization purposes, we need to produce the boundaries of all connected regions. On uniform grids, the beginning and the end of every group of 1s must correspond to the two end points of a line segment in the regions of interest. If the user does not demand a high quality border or the grid is fine enough, simply displaying these end positions might be sufficient. However, to display a smooth boundary, we need to determine the points that are exposed and their neighbors

[1] Our favorite range-encoded bitmap index can be generated from the equality encoded index by $b - 1$ bitwise OR operations. Our observation is that the logical operation time is much smaller than the time to generate the equality encoded index.

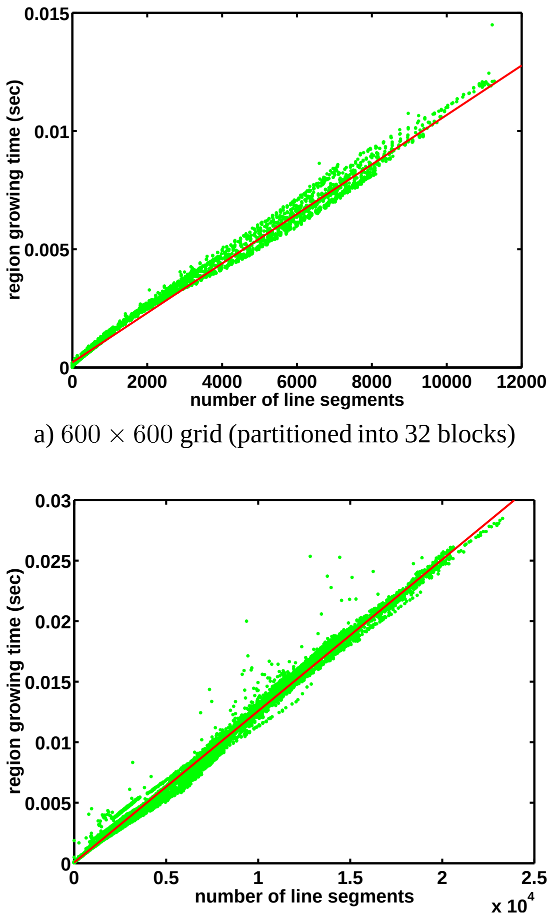
outside so that we can perform the necessary interpolation. This would require us to associate each grid point to a connected region. A straightforward approach is likely to yield an algorithm with complexity of $\mathcal{O}(v)$ because it has to work on each grid point [12]. However, on uniform grids, we can directly work with the line segments instead of working with each individual grid point. This reduces the complexity of the region growing step to $\mathcal{O}(s)$.

For data involving uniform grids, converting a compressed bitmap into lists of line segments is a straightforward task. This work involves visiting every word that represents 1s, converting the global index of the first 1 into its i, j, k coordinates, and determining the number of 1s that follow. Since the number of words containing 1s is proportional to the number of line segments, s_l, this conversion process has a time complexity of $\mathcal{O}(s_l)$. As we decode the compressed bitmap, the line segments are discovered in order of j and k, and can be used in the following comparisons without any addition work.

After the line segments are identified, the next task is to generate a list of pointers to these segments for each connected region. This process can go according to the indices j and k. For each grid line defined by (j, k), we need to compare the line segments on it against those on (j-1, k) and (j, k-1) and possibly (j-1, k-1) depending on the definition of neighbors. Because the line segments along each (j, k) are ordered and do not overlap, determining whether two segments from (j-1, k) and (j, k) are connected requires at most two comparisons. Let $s_{j,k}$ be the number of line segments on grid line (j, k), then the number of comparisons required to identify whether segments on line (j, k) are connected to segments on line (j-1, k) is proportional to $s_{j,k} + s_{j-1,k}$. This operation is repeated for two other lines. The overall complexity of matching segment on line (j, k) is $\mathcal{O}(3s_{j,k}+s_{j-1,k}+s_{j,k-1}+s_{j-1,k-1})$. Since $s_l = \sum s_{j,k}$, the overall complexity of grouping line segments into connected regions is $\mathcal{O}(s_l)$.

While comparing line segments from different grid lines, we can also determine what part of a line segment is exposed to the outside. This is a fixed amount of work each time two line segments are compared. Therefore this does not change the complexity of the region growing algorithm. Displaying the positions of all the exposed points should produce a better boundaries than simply displaying the end points of the line segments. However, if this is still insufficient, we may further generate a smooth boundary through interpolation. This requires one to identify the outside neighbors of the exposed points and perform interpolation. This additional work is also linear in the number of grid points exposed. Therefore, displaying a smooth boundary also requires $\mathcal{O}(s)$ time.

In preparation for the feature-tracking step, we convert the lists of line segments back to compressed bitmaps. In this case, each bitmap represents the grid points belonging to one connected region. We have implemented a procedure that takes $\mathcal{O}(s_l)$ time. The size of a bitmap is proportional to the number of line segments it represents. Since the region growing step does not change the number of line segments, the total size of the compressed bitmaps produced should be about the same as the size of the input bitmap, i.e., their total size is $\mathcal{O}(s_l)$.



a) 600 × 600 grid (partitioned into 32 blocks)

b) 1344 × 1344 grid (partitioned into 256 blocks)

Figure 6: Time used by the region growing procedure plotted against the number of line segments in each test case. Each test case is shown as a green point, the red lines are based on linear regression.

Overall, the region growing process takes one bitmap and generates a number of bitmaps, one for each connected region. Most of the steps in this process have a complexity of $\mathcal{O}(s_l)$ and others have a complexity of $\mathcal{O}(s)$. Since $s \sim 2s_l$, the total time complexity is $\mathcal{O}(s)$ and so is the memory requirement.

To verify that the time required by the region growing procedure is proportional to the number of line segments, we have collected timing results from a large number of different test cases (see details in Section 6). The results are plotted in Figure 6. We observe that the points fall close to the regression lines. Given that time reported is wall-clock time and the figure contains more than 66,000 test cases on two sets of autoignition data, it is remarkable that there are so few outliers. Also note that the datasets use distributed grids, see Figure 2. This verifies that the linear relation holds for nontrivial grids. Another important observation is that the slopes of the two regression lines are remarkably similar, 10^{-6} and 1.25×10^{-6}, even though the two grids are significantly different in size. This indicates that the coefficients in the linear relation do not depend on the grid size and that the time complexity of the region growing algorithm is indeed $\mathcal{O}(s_l)$.

5 Region tracking

When analyzing a spatio-temporal dataset, one important task is to track the evolution of the regions of interest in time [14]. One common approach is to determine the correspondence among the regions from the neighboring time steps based on the spatial overlap of regions. The most time consuming part of this process is the computation of overlaps. Let v_1 and v_2 denote the volumes of two regions to be compared. The straightforward approach of comparing every pair of points has the complexity of $\mathcal{O}(v_1 v_2)$. Given that we represent each region as a compressed bitmap, for uniform grids, computing overlap is equivalent to counting the number of 1s in the result of a bitwise AND operation between two bitmaps. Let s_1 and s_2 denote the boundary sizes of the two regions, the sizes of the compressed bitmaps are $\mathcal{O}(s_1)$ and $\mathcal{O}(s_2)$. Since we can directly operate on the compressed bitmaps, the complexity of the bitwise logical operation is $\mathcal{O}(s_1 + s_2)$. The result of this is also compressed and its size is also $\mathcal{O}(s_1 + s_2)$. Because WAH is a simple compression scheme, counting the number of 1s is a linear operation. Overall, computing overlap between two regions has a complexity of $\mathcal{O}(s_1 + s_2)$ both in space and in time. Given that the regions are represented in $\mathcal{O}(s_1)$ and $\mathcal{O}(s_2)$ bytes, computing the overlap in $\mathcal{O}(s_1 + s_2)$ time and space is optimal because one must examine every byte representing the two regions to determine their overlap.

In our test software, we use the overlaps to define a global identifier (an integer) for each connected region [7]. At the first time step, the regions are arbitrarily numbered. The regions of time step p are compared to the regions of time step p-1, a region in the later time step takes on the number of the region that it has the maximal overlap with; a region that does not overlap with any in the previous time step is again numbered arbitrarily. This is a very simple region tracking algorithm. The purpose of this test is to demonstrate that the bitmap based approach can compute the overlaps efficiently.

		data	index	create
grid size	time	size	size	index
	steps	(MB)	(MB)	(sec)
600×600	69	795	495	621
1344×1344	335	19,364	3,351	6,912

Figure 7: Basic information about the compressed bitmap indices on eight attributes of the autoignition data.

In previous sections, we have compared our approach with the isocontouring algorithm. Most isocontouring algorithms cannot be used easily for region tracking because they do not provide information about the interior of the regions. However, our approach can handle feature tracking with ease because a compressed bitmap is a compact representation of all points in the regions.

A byproduct of our region growing algorithm is that it can produce bounding boxes easily. These bounding boxes can be used to reduce the overall region tracking cost since we only need to compute the overlap of regions whose bounding boxes overlap.

6 Performance on autoignition data

This work was motivated by a need to efficiently analyze the datasets produced from a direct numerical simulation of hydrogen-oxygen autoignition process [5, 6]. We have applied our bitmap indexing software to identify and track the regions of interest on two datasets from this application. Here we report some timing results.

Two sets of data have been used to test the program. Both sets are produced on 2D models and both are produced on uniform grids partitioned into a number of blocks. The smaller dataset uses a 600×600 grid and contains 69 time steps. The larger dataset uses a 1344×1344 grid and contains 335 time steps. Figure 8 shows the regions of interest from two different time steps of the larger dataset. The outlines are labeled using the region tracking algorithm describe in the previous section.

For our performance test, we used randomly generated conditions on eight chemical species involving hydrogen and oxygen. We built indices for these eight attributes. The indices built were the range-encoded bitmap indices and 100 bins were used for each attribute. The total size of the indices and the time needed to create them are listed in a table in Figure 7. In this table, the data size refers to the total size of the eight attributes indexed, not the

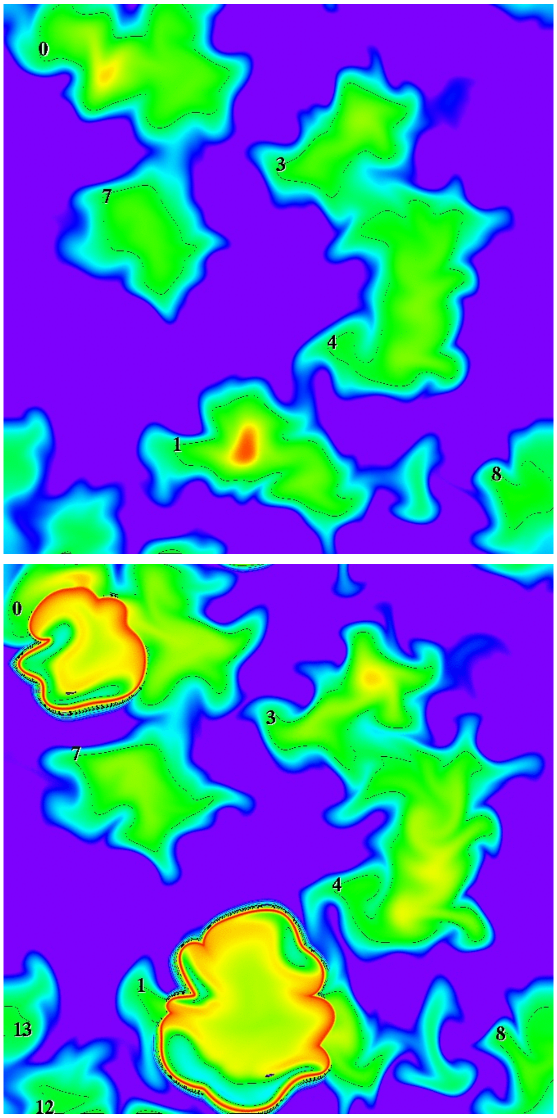
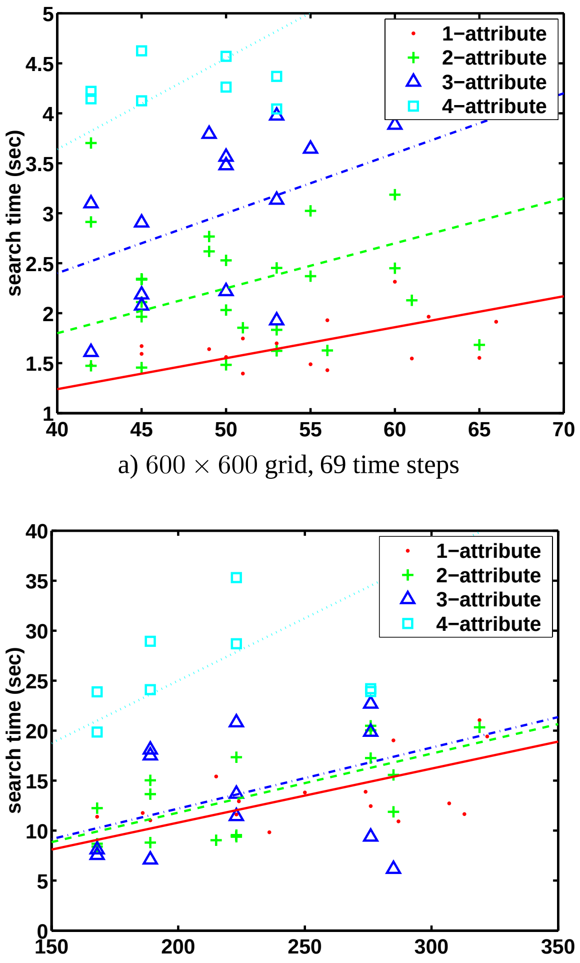


Figure 8: Regions of interest ($HO_2 > 10^{-7}$ on HO_2 background) at selected time steps.

total size of all attributes. The total size of the smaller dataset is about 1.6 GB and the larger one is about 33 GB. Without compression, the bitmap index sizes would be more than three times of the size of the original data, which is close to the sizes of B-tree indices on the same set of attributes. However, the sizes of our compressed indices are significantly smaller than the data sizes.

The timing results reported in Figure 7 are wall clock time measured on a Sun e-450 machine. It uses an UltraSPARC II CPU with a clock rate of 450 MHz and 4 GB of memory. The files reside on a disk suite including five disks. We can usually expect about 20 MB/s throughput from the disk system. The index creation time should be linear in the data size. However, because the larger dataset produced relatively smaller compressed bitmaps, the time needed to create indices for the larger dataset was only 11 times that of the smaller dataset even though their sizes differ by a factor of about 25.



a) 600 × 600 grid, 69 time steps

b) 1344 × 1344 grid, 335 time steps

Figure 9: Search time used by the compressed bitmap indexing scheme to answer queries involving different number of attributes. Each test case is shown as a symbol in the plot, the lines with the same color as the symbols are based on linear regression.

Figure 9 shows the time used by the search step. The time reported is the time used to search through all the time steps in a dataset. More than 200 tests were performed on each dataset[2]. To reduce clutter, only test cases that require more than one second on the smaller dataset and five seconds on the larger dataset are shown. The hor-

[2] On the large dataset, since each test is applied to all 335 time steps, this generates a total of more then 66,000 test cases for the region growing procedure

a) 600×600 grid, 69 time steps

# att	search	grow	track
1	1.06	0.22	0.02
2	1.67	0.17	0.01
3	2.12	0.14	0.01
4	2.62	0.14	0.01

b) 1344×1344 grid, 335 time steps

# att	search	grow	track
1	5.71	2.05	0.12
2	7.39	1.24	0.12
3	8.92	0.58	0.11
4	10.30	0.47	0.10

Figure 10: The average time (seconds) of different random tests.

a) 600×600 grid, 69 time steps

# att	search	total
1	1.91	2.51
2	3.70	3.82
3	3.98	4.13
4	4.62	4.77

b) 1344×1344 grid, 335 time steps

# att	search	total
1	19.41	26.72
2	20.33	26.08
3	22.68	23.42
4	35.30	35.95

Figure 11: The test cases that used the maximal amount of total time (seconds).

izontal axes in the figure are the number of time steps containing nonempty regions of interest. The tests are conducted using conditions of the form “X $>$ a and Y $>$ b” by randomly selecting the attributes and the threshold values. The linear regression lines are presented not to suggest a linear relation, but merely to guide the comparisons among the queries involving different numbers of attributes.

Figure 10 shows the average time used by the three steps of feature tracking, searching, region growing and region tracking. In most test cases, the time required by the search step is a factor of 5–10 larger than the time required by the region growing step which is in turn about a factor of 5–10 larger than the time required by the region tracking step. This is dramatically different from performance data reported in the literature [12] where the region growing time is 20–40 times that of searching time. This is because the bitmap indexing scheme allow the data to be in an order that is efficient for region growing. Most other multidimensional indexing schemes do not allow this option.

As the number of attributes in a query increases, the searching time generally increases. We expect the average search time to be proportional to the total index size.

record oriented	75
attribute oriented	5

Figure 12: Time (seconds) used to search the smaller dataset without an index.

The total size of indices for the larger dataset is about 6.7 times that of the smaller one. The average search times only differ by a factor of about 5. This is close to what we expected. The time required by region growing step and the region tracking step decreases as the number of attributes in a query increases. This is because the conditions involving more attributes typically produce smaller regions of interest.

Figure 11 shows the maximum time used by the extreme test cases. The total time is less than 40 seconds in all test cases on the large dataset. Much of this time is spent in waiting for the disk system to respond to read requests.

For a typical thresholded region defined on one attribute, the search step on the smaller dataset takes less than two seconds and the search step on the larger one takes less than 20 seconds. As more attributes are specified, the worst-case time grows slower than linear. For example, for regions of interest defined on four attributes, it takes less than five seconds on the smaller dataset and less than 40 seconds on the larger dataset, see Figures 9 and 11.

Figure 12 lists the time required to perform the searching step using the the most naive algorithms on the smaller dataset. Clearly, these naive algorithms take more time than the numbers presented in the Figures 10 and 11. Depending how the data is organized the searching time is different. If data is organized in a record oriented fashion like in a relational database system, to search one attribute effectively requires one to read all pages. The alternative is to organize the data in an attribute oriented fashion, i.e., all values of an attribute are together on consecutive pages. This strategy is suitable for read-only data and is more efficient for feature tracking. In this case, it requires about 5 seconds to perform the searching step without an index. Since the average search step takes about one second with the compressed bitmap index, there is a saving of about four seconds. From Figure 7, we see that generating one bitmap index for the smaller dataset takes about 77.6 (621/8) seconds. It would take about 20 queries for the savings of using the bitmap index to equal to the time needed to create the index. Building the bitmap indices may not be worthwhile if the data is only searched a few times.

7 Summary and future work

In this paper, we reported a set of compressed bitmap based techniques for feature tracking. The bitmap index is known to be efficient for searching read-only data like those in feature tracking problems. In this paper, we demonstrate that compressed bitmaps can also be efficiently used to perform region growing and region tracking tasks. On uniform grids, our bitmap based approaches for region growing and region tracking are theoretically optimal. Our tests on two sets of autoignition data confirm their efficiencies. On a 33 GB dataset, it takes less than a minute to process complex conditions involving four different attributes. On the average it only took about 10 seconds.

We have shown that our algorithm for identifying regions of interests has the same theoretical complexity as the optimal isocontouring algorithm. However, because isocontouring algorithms can not be directly used in feature tracking, we did not actually implement any isocontouring algorithm. It would be interesting to actually implement one of them and compare the execution time.

We might be able to improve the performance of the feature tracking process by reordering the grid points so that the grid is not blocked. This should reduce the size of the bitmap indices and the space requirement during feature tracking. The question to be answered is whether the overhead of reordering is worthwhile.

In this paper, both the analyses and the test data are for uniform grids. However, the algorithms used can be extended. For example, the region growing algorithm can be applied to any regular grid including the nonuniform regular grid and the rectilinear grid with affecting the complexity ($\mathcal{O}(s)$). The searching step using bitmap index can be applied on any kind of grid. If the order line passes through space in a reasonable manner, the bitmap representing the regions of interest can still be $\mathcal{O}(s)$. The algorithms may also be adopted to AMR (automatic mesh refinement) meshes. However, more work is required to implement and to verify the approaches.

The compressed bitmap is a compact representation for line segments. On 3D grids, working with rectangles or cubes should be more efficient. Developing compact data structure for these geometric objects might further enhance the overall performance of feature tracking programs.

References

[1] C. Bajaj, A. Shamir, and B.-S. Sohn. Progressive tracking of isosurfaces in time-varying scalar fields. Technical Report TR-02-4, CS & TICAM, University of Texas at Austin, 2002.

[2] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *SIGMOD 1998*. ACM press, 1998.

[3] C. Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *SIGMOD 1999*. ACM Press, 1999.

[4] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Optimal isosurface extraction from irregular volume data. In *Volume Visualization Symposium*, pages 31–38, 1996.

[5] T. Echekki and J. H. Chen. Direct numerical simulation of autoignition in non-homogeneous hydrogen-air mixtures, 2003. to be published in Combustion and Flame.

[6] H. G. Im, J. H. Chen, and C. K. Law. Ignition of hydrogen/air mixing layer in turbulent flows. In *Twenty-Seventh Symposium (International) on Combustion, The Combustion Institute*, pages 1047–1056, 1998.

[7] W. Koegler. Case study: Application of feature tracking to analysis of autoignition simulation data. In *IEEE Visualization '01*, pages 461–464, 2001.

[8] Y. Livnat, H. W. Shen, and C. R. Johnson. A near optimal isosurface extraction algorithm for structured and unstructured grids. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, 1996.

[9] P. O'Neil. Model 204 architecture and performance. In *2nd International Workshop in High Performance Transaction Systems, Asilomar, CA*, volume 359 of *Lecture Notes in Computer Science*, pages 40–59, Sept. 1987.

[10] P. O'Neil and D. Quass. Improved query performance with variant indices. In *SIGMOD 1997*, pages 38–49. ACM Press, 1997.

[11] H. W. Shen, C. D. Hansen, Y. Livnat, and C. R. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). In *IEEE Visualization '96*, pages 287–294, 1996.

[12] Q. Shi and J. F. Jaja. Efficient techniques for range search queries on earth science data. In *SSDBM 2002*, pages 142–151. IEEE Computer Society, 2002.

[13] A. Shoshani, L. M. Bernardo, H. Nordberg, D. Rotem, and A. Sim. Multidimensional indexing and query coordination for tertiary storage management. In *SSDBM 1999*, pages 214–225. IEEE Computer Society, 1999.

[14] D. Silver and X. Wang. Tracking and visulizing turbulent 3D flow. *IEEE Transcations on Visualization and Computer Graphics*, 3(2):129–141, 1997.

[15] K. Wu, E. J. Otoo, and A. Shoshani. A performance comparison of bitmap indexes. In *CIKM 2001*, pages 559–561. ACM, 2001.

[16] K. Wu, E. J. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *Proceedings of SSDBM'02*, pages 99–108, 2002. LBNL-49627.

[17] K. Wu, E. J. Otoo, A. Shoshani, and H. Nordberg. Notes on design and implementation of compressed bit vectors. Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory, Berkeley, CA, 2001.

[18] K.-L. Wu and P. Yu. Range-based bitmap indexing for high cardinality attributes with skew. Technical Report RC 20449, IBM Watson Research Division, Yorktown Heights, New York, May 1996.

[19] M.-C. Wu and A. P. Buchmann. Encoded bitmap indexing for data warehouses. In *ICDE 1998*, pages 220–230. IEEE Computer Society, 1998.