# Isosurfacing in Span Space with Utmost Efficiency (ISSUE)

Han-Wei Shen[†] Charles D. Hansen[‡] Yarden Livnat[†] Christopher R. Johnson[†]

[†] Department of Computer Science
University of Utah
Salt Lake City, UT

[‡] Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, New Mexico

## Abstract

*We present efficient sequential and parallel algorithms for isosurface extraction. Based on the Span Space data representation, new data subdivision and searching methods are described. We also present a parallel implementation with an emphasis on load balancing. The performance of our sequential algorithm to locate the cell elements intersected by isosurfaces is faster than the Kd-tree searching method originally used for the Span Space algorithm. The parallel algorithm can achieve high load balancing for massively parallel machines with distributed memory architectures.*

## 1 Introduction

Scientific visualization has played an important role in understanding three-dimensional scalar data. As cost-effective high performance computers with large amount of memory and disk space become more accessible, the sizes of these scalar data also continue to increase. To visualize these large-scale data sets, generally two different paradigms are used. One paradigm is to transfer the data onto graphics workstations and perform the visualization as a postprocessing step. Alternatively, visualization can be performed on the same, typically parallel, machines that run the simulation thereby providing the user faster feedback necessary for computational steering. In this paper, we propose an efficient sequential isosurfacing algorithm and a load balanced parallel isosurfacing algorithm to fulfill the requirements of both visualization paradigms.

Isosurfacing is an effective technique to explore three-dimensional scalar fields. A simple and effective method is the *Marching Cubes* algorithm, proposed by Lorensen and Cline [1]. The algorithm has a complexity of $O(N)$ since it is necessary to visit each cell* in the three-dimensional field. When the data set is large, visiting each cell is too costly and recent research efforts have investigated the acceleration of the isosurfacing process, namely Wilhelms and Van Gelder's *octree* spatial subdivision [2], Gallengher's *span filter* [3], Itoh and Koyamada's *extreme graph* method[4], Shen and Johnson's *sweeping simplices* algorithm [5],and Livnat et al.'s *near optimal isosurface extraction* (*NOISE*) algorithm [6].

Among the above accelerating techniques, the NOISE algorithm is near optimal. This algorithm has a worst case complexity of $O(\sqrt{N} + K)$ to locate the cells that are intersected by the isosurfaces, where $N$ is the total number of cells in the scalar field, and $K$ is the number of isosurface cells. The crux of this algorithm is a novel data representation, termed the *Span Space*. Using this representation, the isosurface extraction process can be reduced into a range searching problem. Livnat *et al.* proposed a classical Kd-tree searching method [7] to locate, in that space, the cells that contain an isosurface.

In this paper, we use the Span Space as the underlying representation to design high performance isosurface extraction algorithms for both single processor workstations and massively parallel machines with distributed memory architectures. Rather than using the Kd-tree searching method, we subdivide the Span Space into a two-dimensional regular lattice and propose a new searching method. Our new sequential algorithm leads to a average case time complexity of $O(log(\frac{N}{L}) + \frac{\sqrt{N}}{L} + K)$ to locate the isosurface intersected cell elements, where $L$ is an user specified parameter, as explained in Section 3, with a value typically between 200 to 500. In practice, this new method is faster than the NOISE algorithm in locating the isosurface cells. Our parallel isosurfacing algorithm adopts a static load balancing scheme to distribute the cells among Processing Elements (*PEs*). Each PE executes the sequential algorithm locally leading to an average difference be-

* In a uniform three-dimensional field, a cell is sometimes referred to as a *voxel*. We use the term *cell* to indicate elements of a three-dimensional grid that may be a uniform or regular structured grid or an unstructured grid. The cells may be tetrahedra, hexahedra, prisms or other polyhedra. The methods described in this paper are useful for any type of grid.

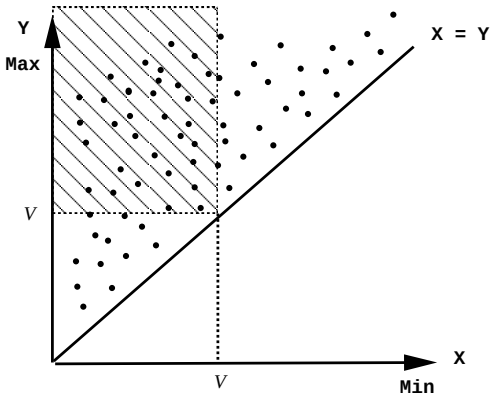


Figure 1: Span Space

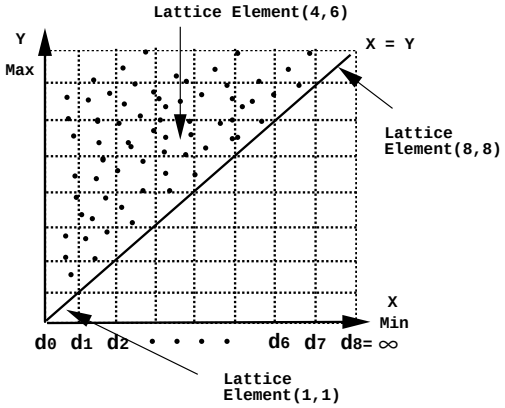


Figure 2: Lattice Subdivision

tween the maximum and minimum workloads of lower than 2%.

We begin the paper by providing details of the Span Space data representation. Next, we describe the new lattice subdivision method with a fast searching algorithm. We then discuss some implementation details. Building upon this, we present the parallel algorithm with an emphasis on the load balancing. Finally, we conclude the paper with several experimental results.

## 2 Span Space

For each cell element in the three-dimensional scalar field, there exists an interval $[a, b]$ representing the scalar range of the data at the cell's vertices, where $a$ is the cell's minimum value and $b$ is the cell's maximum value. For a given isovalue $v$, the cell $C_i$ that has interval $[a_i, b_i]$ such that $a_i < v$, and $b_i > v$ is intersected by the isosurface. To accelerate the isosurfacing process, researchers have proposed different methods to decompose the data domain such that for each isovalue, there is only a small number of subdomains that need to be examined [3, 5].

Livnat *et al.* provide an interesting perspective for the isosurfacing problem [6]. For a cell with minimum value $a$ and maximum value $b$, instead of treating the $[a, b]$ as an interval, they map the cell into an unique point position, $(a, b)$, in an $R^2$ value space, termed the Span Space. Figure 1 illustrates the Span Space. The horizontal axis $X$ depicts a cell's minimum value, and the vertical axis $Y$ depicts a cell's maximum value. Note that cell elements can be mapped only to the half space above the $X = Y$ line because a cell's maximum value is always greater than or equal to its minimum value. Using the Span Space data representation, the isosurfacing problem is then reduced into a classical range search problem. The problem is stated as follow:

- For a given isovalue $v$, the cell $C_i$ that has associated points $(x_i, y_i)$ in the Span Space, such that $x_i < v$ and $y_i > v$ is an isosurface cell.

In Figure 1, cells having points within the shaded area are the isosurface cells.

Unlike the interval representation for a cell that poses difficulties for subdividing the cells in the scalar field, the point representation in the Span Space provide a much simpler way to subdivide the data domain. This advantage lays down the basis for us to develop an efficient searching algorithm.

## 3 New Searching Algorithm

In this section, we describe a data subdivision scheme and a new searching algorithm to locate the isosurface cells. Based on the Span Space representation, the new subdivision scheme organizes the cells in such a way that the isosurface cells can be easily located.

### 3.1 Lattice Subdivision

Our algorithm decomposes the data domain by subdividing the Span Space into a two dimensional $L \times L$ lattice. Assuming that the scalar field has a global minimum value $m$, and a global maximum value $M$, we define a set of *dividing points* $\{d_i\}_{i=0}^{i=L}$ such that $d_0 = m$, $d_L = \infty$, $d_i < d_{i+1}$, and $\{d_i\}_{i=1}^{i=L-1} \in (m, M]$. A lattice element$(i, j), i = 1..L$ and $j = 1..L$ is defined as a square region in the Span Space containing point $(x, y)$ such that $x \in [d_{i-1}, d_i)$ and $y \in [d_{j-1}, d_j)$. Figure 2 shows a $8 \times 8$ lattice subdivision imposed upon the Span Space. Note that the $X = Y$ line crosses the diagonals of lattice element$(i, i), i = 1..L$. Also, all the lattice elements with indices $(i, j), i > j$ are empty because the minimum values can not be greater than the maximum values.

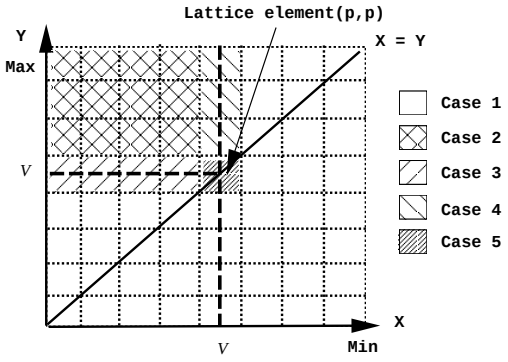


Figure 3: Lattice Classification

### 3.2 Searching Algorithm

Using the lattice subdivision, we can quickly locate the candidate lattice elements that contain the isosurface cells. Given an isovalue $v$, $v \in [d_{p-1}, d_p)$, we classify the lattice elements in the Span Space into five cases based on their indices $(i, j)$ as follows:

1. $i > p$ or $j < p$: All the cells in this region have either a higher minimum value or lower maximum value than the isovalue. Hence these lattice elements trivially do not contain any isosurface cells.

2. $i < p$ and $j > p$: All the cells in these lattice elements are isosurface cells.

3. $i < p$ and $j = p$: All the cells in this region have a lower minimum value than the isovalue. Hence only those cells that have a higher maximum value than the isovalue are isosurface cells.

4. $i = p$ and $j > p$: All the cells in this region have a higher maximum value than the isovalue. Hence only those cells that have a lower minimum value than the isovalue are isosurface cells.

5. $i = p$ and $j = p$: This is the only lattice element that requires a min-max search to locate the isosurface cells. Any isosurfacing algorithm, such as a Kd-tree searching method or sweeping simplices, will do.

Figure 3 shows the five cases in the Span Space.

From the above description, the lattice elements in case 1 can be immediately rejected. Locating isosurface cells from the case 2 region requires no searching operation since every cell in the region is an isosurface cell. The cells can be directly collected from the *Lattice Element* data structures that contain cell indices.

To locate isosurface cells in the lattice elements of case 3, we design a *Row* data structure. Row $[R]$ contains indices and maximum values of cells in lattice elements $(i, j)$, $i < R, j = R$. The cell indices are sorted by their maximum values. To collect the isosurface cells, we apply a binary search to Row $[p]$ and find the cells with maximum values greater than the isovalue $v$.

To collect isosurface cells in the lattice elements of case 4, we design a *Column* data structure. Column $[C]$ contains indices and minimum values of cells in lattice elements $(i, j)$, $i = C, j > C$. The cells in each column structure are sorted by their minimum values. Those cells in Column $[p]$ with minimum values lower than the isovalue $v$ are isosurface cells and can be located with a binary search.

The lattice element in case 5 is the only region that we need to employ regular isosurface searching, i.e., finding cells with minimum values lower, and maximum values higher than the isovalue. To achieve this, we can use any efficient isosurface extraction algorithm. For instance, we can build a Kd-tree structure for lattice element $(p, p)$ and apply Kd-tree search to locate the isosurface cells or we could employ the Sweeping Simplices algorithm [5].

The search phase of our isosurfacing algorithm includes two binary searches in the regions of case 3 and case 4, and one min-max search in the lattice element of case 5. Since the entire Span Space contains $L$ rows, $L$ columns, and $\frac{L\times(L+1)}{2}$ lattice elements above the $X = Y$ half space, the average number of cells in each row and column is $\frac{N}{L}$, and the average number of cells in each lattice element is $\frac{2N}{L(L+1)}$. The binary search for each row and column requires $O(log(\frac{N}{L}))$, and the Kd-tree mix-max search for the lattice element in case 5 requires $O(\frac{\sqrt{N}}{L})$. Hence, the overall average case performance for our new algorithm is then $O(log(\frac{N}{L}) + \frac{\sqrt{N}}{L} + K)$, where $K$ is the number of the isosurface cells.

### 3.3 Implementation Details

In this section, we provide important implementation details of our searching algorithm. First, we describe how to determine the dividing points $\{d_i\}$. Second, we describe a sparse manipulation method to avoid visiting the empty lattice elements when collecting the isosurface cells.

From the earlier description, we know that a lattice element $(i, j)$ is a region in the Span Space containing points $(x, y)$ such that $x \in [d_{i-1}, d_i)$ and $y \in [d_{j-1}, d_j)$. Assuming that the value range of the field is $[m, M], m, M \in R$, and that the Span Space is subdivided into an $L \times L$ lattice, a straightforward way to determine $\{d_i\}$ is to evenly cut the interval $[m, M]$, that is, $\{d_i = m + i \times \frac{(M-m)}{L}\}_{i=0}^{i=L-1}$ and $d_L = \infty$. However, this method does not produce a uniform data point distribution at each interval $[d_i, d_{i+1}]$ which results in an uneven cell distribution among the

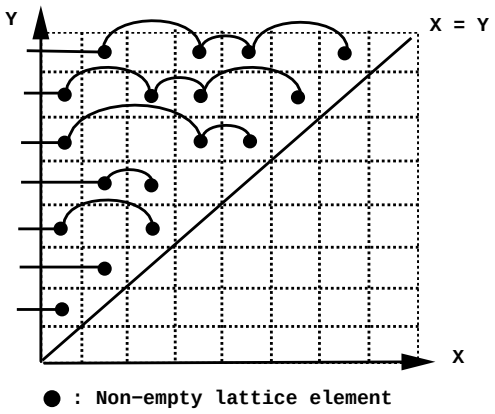


Figure 4: Sparse Manipulation

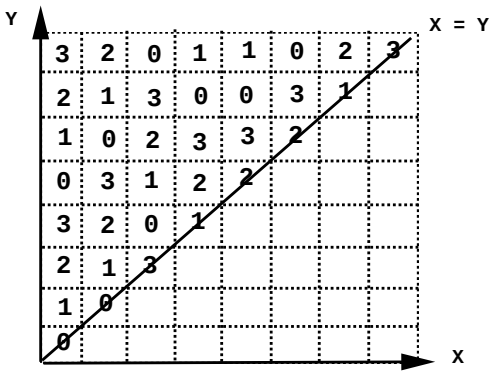


Figure 5: Lattice Distribution

lattice elements. To avoid this, we find $\{d_i\}_{i=0}^{i=L-1}$ in such a way that the number of data points at each interval $[d_i, d_{i+1}]$ is approximately the same. We achieve this by sorting all data points into a list and dividing the list into $L$ sublists having approximately the same lengths. The scalar values which bound those sublists are the dividing points.

As mentioned earlier, only lattice elements in cases 3, 4, and 5 require searching operations to locate the isosurface cells. The finer we subdivide the Span Space, the smaller the areas of the regions defined by those cases. This results in a greater number of cells which are located in the case 2 region and therefore can immediately be collected. However, as we more finely subdivide the Span Space, there can be a larger number of empty lattice elements. This has the potential to degrade the algorithm's performance since time would be spent checking those empty lattice elements when we collect the isosurface cells. To overcome this limitation, we use a sparse manipulation method on the lattice. As we pre-process the data field and distribute the cells into the lattice, the non-empty lattice elements are marked. The lattice elements at each row are then connected together with pointers. Figure 4 illustrates the sparse manipulation method. We note that using sparse manipulation, the number of non-empty lattice elements is bounded by the number of cells in the 3D scalar field no matter how fine we subdivide the Span Space. In the results section, we show the relationship between the resolution of the lattice subdivision and the performance of the searching algorithm.

## 4 Parallel Algorithm

In this section, we present a parallel isosurfacing algorithm. The underlying architecture model is massively parallel machines with distributed memory such as the Cray T3D. The algorithm can be divided into three phases: cell distribution, initialization, and isosurface extraction. In the cell distribution phase, cells are partitioned into several subsets and distributed to the processing elements (*PEs*). In the initialization phase, each PE builds lattice, row, and column data structures based on the local data. In the isosurface extraction phase, each PE locally employs our searching algorithm to extract the isosurfaces.

Our emphasis is on paradigms of cell distribution achieving load balancing. For any given isovalue, we want the PEs to spend a balanced amount of time in isosurfacing and to produce balanced amount of triangles. In this way, not only does our isosurfacing algorithm exhibit good scalability, it can also be directly connected to a parallel rendering process, which requires an even distribution of primitives for the initial geometry processing[8].

We achieve the load balancing by carefully designing a cell distribution scheme. Ideally, if cells within any scalar range $[a, b]$ are evenly scattered, each PE would have approximately the same number of isosurface cells for any isovalue. To achieve this, we use a cell distribution method built on top of the lattice subdivision of the Span Space. Assuming that there are $L \times L$ lattice elements in the Span Space, and that there are $N$ PEs available, numbering from PE$[0]$ to PE$[N-1]$, we unfold the lattice elements in the half space above the $X = Y$ line column by column into a 1D list and distribute these $\frac{L\times(L+1)}{2}$ elements into the PEs using a round-robin method. Figure 5 shows a lattice distribution of $8 \times 8$ lattice with 4 available PEs. To express our round-robin method in terms of indices of lattice elements and PEs, our method distributes the cells in the lattice$(i, j)$ into PE$[(j-1+\frac{(2L-i)\times(i-1)}{2}) mod N]$. As a result, each PE receives a balanced work load because the lattice elements in cases 2,3,4,5 are evenly distributed.

The resolution of the lattice subdivision is crucial to the load-balance of the algorithm since a finer subdivision exhibits better cell scattering. However, in the isosurfacing algorithm, creating a fine subdivision implies that we have to create more lattice data objects, which would incur higher memory overhead. To overcome this, we decouple the lattice subdivision used for the cell distribution from the one used for isosurfacing algorithm. Initially, a finer lattice subdivision is used for the round-robin distribution scheme. After each PE receives its local data, a coarser lattice subdivision is used to create the lattice, row, and column data structures. In this way, we can exploit a fine subdivision which achieves good cell scattering, but not invoke excessive memory overhead in performing isosurfacing. We refer to the elements of this subdivision for the cell distribution as *buckets* to distinguish from *lattice elements* used for the isosurfacing algorithm.

## 5 Results and Discussion

In this section, we present empirical results to evaluate our algorithms. The sequential algorithm was tested on a 150 MHz MIPS R4400 processor. The parallel algorithm was tested on a Cray T3D parallel machine. All the results presented were obtained by averaging one thousand executions with randomly assigned isovalues.

### 5.1 Sequential Algorithm

We used three unstructured grid data sets to test our sequential algorithm. These data were generated from bioelectric field problems solved using finite element methods. The data sizes range from 69 thousand to 1.3 million elements. Table 1 gives a summary of the data sets. Figures 11–13 depict a single iso-surfaced image for each of the data sets. The performance of the searching phase of the algorithm is affected by the resolution of the lattice subdivision. The finer we subdivide the Span Space, the smaller the area of the regions covered by case 3,4,5 while the greater the area of the region covered by case 2. However, This is mitigated by the overhead of constructing the necessary data structures. Figure 6 demonstrates the relationship between the time to search for isosurface cells and the resolution of the lattice subdivision. We can see that the search time dramatically decreases as we increase the number lattice elements up to $256 \times 256$. After that, the performance degraded slightly due the overhead incurred by using a very fine lattice structure. Figure 7 shows the total isosurfacing time, including the time for triangulation, verses the resolution of the lattice subdivision. Because we used the sparse manipulation method mentioned in the section 3.3, the overhead incured by a very fine subdivision is not overwhelming.

| Data Set | Vertices | Cells |
|---|---|---|
| Heart | 11,504 | 69,892 |
| Torso | 201,142 | 1,290,072 |
| Brain | 74,217 | 471,770 |

Table 1: Data Sets

| Method | Heart | Torso | Brain |
|---|---|---|---|
| Lattice | 0.017 | 0.129 | 0.052 |
| Kd-tree | 0.4 | 2.2 | 1.5 |

Table 2: Comparison of the lattice method with the Kd-tree method in locating the isosurface cells.(in msecs)

512x512 lattice elements were used in our experiments. Table 2 shows the times for locating which cells contain an isosurface for both the Lattice based algorithm and the Kd-tree algorithm. Note that the time to locate the isosurface cells is an order of magnitude faster. Table 3 compares the total isosurfacing time: locating which cells contain an isosurface, traversing those cells to perform triangulation, and the triangulation time. It can be seen that the Lattice based search improves the overall performance by approximately 25%. The triangulation time begins to dominate which is why the time to locate the isosurface cells is an order of magnitude faster but overall the system exhibits only a 25% increase in performance.

The initialization complexity for our algorithm is $O\left(2L \times \frac{n}{L}\log(\frac{n}{L}) + n\right)$, where $L$ is the number of bins used at each dimension of the span space, and $n$ is the number of cells. The is because that we need to sort the cells at each row and column based on their minimum or maximum values, and put the cell indices into appropriate lattice elements. Table 4 gives the initialization time for the test data sets.

The memory requirement for our algorithm is $O(n)$. This includes cell indices , minimum, and maximum val-

| Method | Heart | Torso | Brain |
|---|---|---|---|
| Lattice | 4.65 | 33.47 | 41.33 |
| Kd-tree | 7.0 | 43.8 | 53.9 |

Table 3: Comparison of the lattice method with the Kd-tree method in total isosurfacing time.(in msecs)

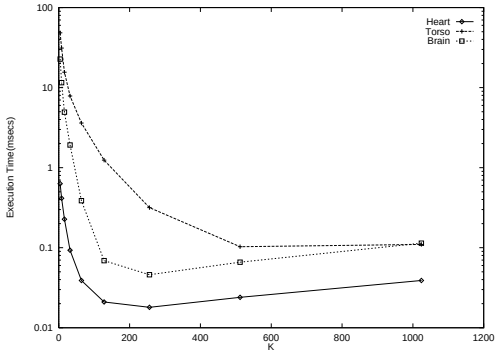


Figure 6: Searching Time v.s. K x K Lattice Subdivision

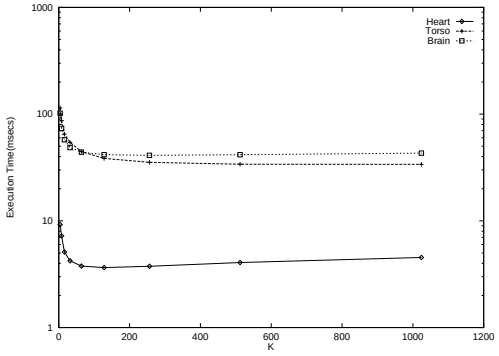


Figure 7: Total Isosurfacing Time v.s. K x K Lattice Subdivision

ues stored in the row and column date structures, cell indices stored in lattice elements, and memory consumed by the algorithm applied to cells in the lattice elements of case $5$. We assume that the memory requirement for the algorithm chosen to extract isosurfaces from cells in case $5$ is bound by $O(n)$, which is true if we use either KD-tree or Sweeping Simplices searching methods.

## 5.2 Parallel Algorithm

We have implemented our parallel algorithm using C++ on a Cray T3D supercomputer in the Advanced Computing Laboratory at Los Alamos National Laboratory. The Cray T3D is a massively parallel computer with a distributed memory architecture. Each processing element has a 64 bit DEC Alpha microprocessor and 8M words local memory.

| Method | Heart | Torso | Brain |
|---|---|---|---|
| Time | 1.88 | 53.19 | 13.01 |

Table 4: Initialization Time.(in secs)

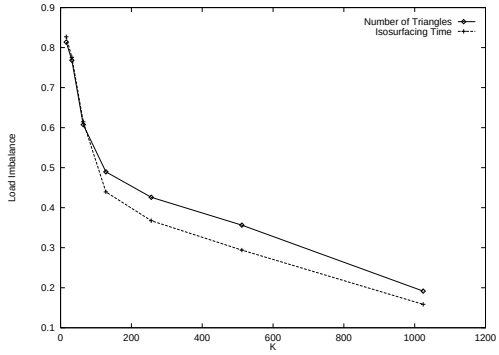


Figure 8: Load Imbalance for K x K Bucket Subdivision

Our implementation uses the message passing paradigm by employing the ACLMPL message passing library [9] which is a high throughput, low latency communications library.† In this section, we show the load balancing characteristics of our parallel algorithm and give the speedup factors obtained from executions using 4 to 64 processing elements. We used the brain data set which has 471,770 cell elements.

To measure the load balance of our parallel algorithm, we use two different metrics. One is a formula of load imbalance used by Ma [10]:

- Load Imbalance = $1 - \frac{load_{Average}}{load_{Max}}$

The other is a load difference formula:

- Load Difference = $(100 \times \frac{load_{Max} - load_{Min}}{load_{Total}})\%$

Two different measurements are used to define the workload for each PE. One is the isosurfacing times for each PE, the other is the number of triangles produced by each PE. We present both of the workload measurements to evaluate our algorithm.

From our earlier discussion, we know that the load balance is affected by the resolution of bucket subdivision. Figure 8 and Figure 9 show the load imbalance and load difference, for both workload measurements, using 32 PEs. We increased the resolution of the bucket subdivision from $16 \times 16$ to $1024 \times 1024$. The results show that we can obtain a highly balanced load, namely under 0.2 of load imbalance and 2% of load difference for a $1024 \times 1024$ bucket subdivision. Remember that the bucket subdivision is a subdivision of the Span Space used to distribute the cell elements, which is different from the lattice subdivision used to perform the isosurfacing algorithm.

†We used ACLMPL since the MPI implementation on the T3D is not yet mature. The message passing library employed will effect the performance but is independent of the isosurfacing algorithm.

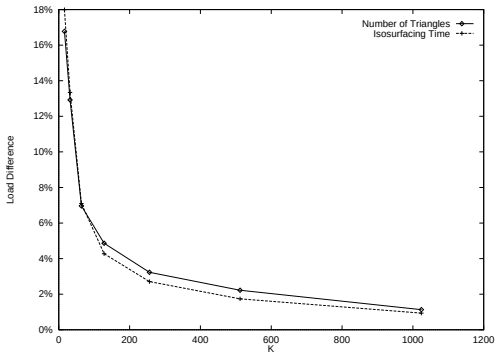


Figure 9: Load Difference for K x K Bucket Subdivision

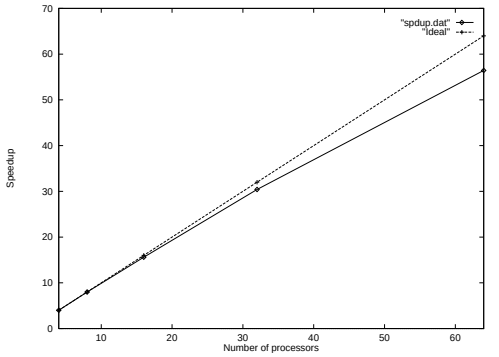


Figure 10: Speedup Factors

Figure 10 gives the speedup factors for T3D partitions with 4 to 64 PEs. The test was performed with a $256 \times 256$ lattice subdivision.

## 6 Conclusion and Summary

We have presented a high performance isosurfacing algorithm using a regular $L \times L$ lattice subdivision of the Span Space. The algorithm has a average case time complexity of $O(log(\frac{N}{L}) + \frac{\sqrt{N}}{L} + K)$, where the $N$ is the total number of cells in the scalar field, $K$ is the number of isosurface cells, and $L$ is a user specified parameter. In practice, it is faster than the Kd-tree searching method. Empirically, the algorithm has its best performance when the value of $L$ is about 200 to 500 for scalar data sets with sizes ranging from hundreds of thousands to millions of cell elements. We have also presented a load balanced parallel isosurfacing algorithm. In addition to the lattice subdivision, we use a bucket subdivision of the Span Space and a round-robin method to distribute the cell elements. Our experimental results show that the higher the resolution of the bucket subdivision, the better the load balance. Our sequential and parallel isosurfacing algorithm can satisfy the needs of both post-processing and computational steering visualization.

## Acknowledgments

The work was performed under the auspices of the United State Department of Energy, Los Alamos National Laboratory and was supported in part by the National Science Foundation and the National Institutes of Health. The research was performed in part using the resources located in the Department of Computer Science at the University of Utah and at the Advanced Computing Laboratory of Los Alamos National Laboratory. Furthermore, the authors appreciate access to facilities that are part of the NSF STC for Computer Graphics and Scientific Visualization.

## References

[1] W.E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987.

[2] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[3] R. S. Gallagher. Span filter: An optimization scheme for volume visualization of large finite element models. In *Proceedings of Visualization '91*, pages 68–75. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[4] T. Itoh and K. Koyyamada. Isosurface generation by using extreme graphs. In *Proceedings of Visualization '94*, pages 77–83. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[5] H.W. Shen and C.R. Johnson. Sweeping simplices: A fast isosurface extraction algorithm for unstructure grids. In *Proceedings of Visualization '95*. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[6] Y. Livnat, H.W. Shen, and C.R. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Transaction on Visualization and Computer Graphics*, 2(1), March 1996.

[7] F.P. Preparata and M.I. Shamos. *Computational Geometry, an introduction*. Springer-Verlag Publishing Company, 1985.

[8] S. Molnar, M. Cox, D Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, pages 23–32, July 1994.

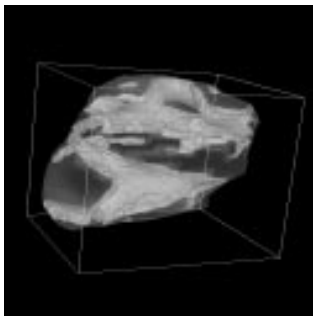

Figure 11: A yellow isosurface within a multi-colored semi-transparent heart model

[9] J. Painter, P. McCormick, M. Krogh, C. Hansen, and G. Colin de Verdière. The acl message passing library. *EPFL Supercomputing Review*, 7, November 1995.

[10] K.-L. Ma. Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. In *Proceedings of 1995 Parallel Rendering Symposium*, pages 23–30, 1995.

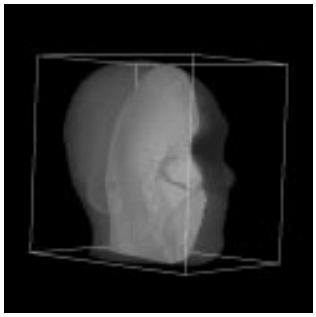

Figure 12: A green isosurface within a multi-colored semi-transparent head model

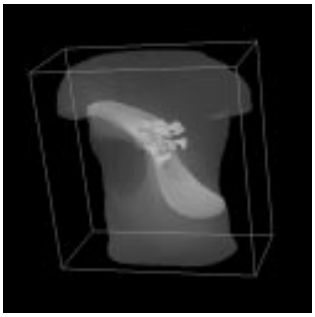

Figure 13: A yellow isosurface within a multi-colored semi-transparent torso model