Parallel Accelerated Isocontouring for Out-of-Core Visualization

C. L. Bajaj V. Pascucci D. Thompson X. Y. Zhang
Department of Computer Sciences and TICAM
University of Texas, Austin, TX 78733
http://www.ticam.utexas.edu/CCV

Abstract

In this paper we introduce a scheme for static analysis that allows us to partition large geometric datasets at multiple levels of granularity to achieve both load balancing in parallel computations and minimal access to secondary memory in out-of-core computations. The idea is illustrated and fully exploited for the case of isosurface extraction, but extendible to a class of algorithms based on a small set of parameters and for which an appropriate static analysis can be performed.

1 Introduction and Related Work

Given a Scalar Field, $w(\mathbf{x})$, defined over a d-dimensional bounded volume mesh ($\mathbf{x} \in \mathbb{R}^d$), we often visualize the data by rendering a $(d-1)$-dimensional surface satisfying $w(\mathbf{x})$ =*const*. This visualization technique is popularly known as *isocontouring*. In order to obtain a good understanding of the volume data, isosurfaces of multiple representative isovalues need to be computed. To compute a single isosurface without preprocessing one needs to visit all the cells of the input. However, if you are querying for multiple isosurfaces, it pays to preprocess the data to avoid visiting cells that don't contain any part of the current query isocontour. This approach has come to be known as *accelerated isocontouring*.

Early work has predominantly focused on algorithms for extracting a single isosurface from volume data [14]. The same goal was addressed more recently with the particle-based method [7]. Several algorithms have been developed to addressed the problem of accelerated isocontouring [21, 10, 12, 19, 13, 20, 1], where the volumetric data is preprocessed to allow for multiple fast queries. Papers [1, 5, 13] use interval, segment, or k-d trees to index the cells in the scalar field. The advantage of the approach in [1] lies in the fact the only a small set of "seed" cells is indexed in the search structure. A set of seed cells is a set of cells guaranteed to intersect every isolated portion of an isosurface for any isovalue. The search tree structure stores seed cells according to the range of values spanned by $w(\mathbf{x})$ in each cell. In this way, when an isovalue is given, the tree returns efficiently seed cells that intersect the desired isosurface. Contour propagation is then used to generate the entire isosurface from the seed cells. The complexity of this algorithm is $O(\log n' + k)$, where n' is the size of the seed set and k is the number of cells intersecting the output isocontour. This seed cell algorithm is the basis of our parallel accelerated isocontouring.

As the size of the input data increases, isocontouring algorithms need to be executed out-of-core and/or on parallel machines for both efficiency and data accessibility. An I/O optimal implementation of the search tree was presented in [3]. The method has been later improved with a better empirical tradeoff for improving I/O speed [4]. Hansen and Hinker describe parallel methods for isosurface generation on SIMD machines [11]. Ellsiepen describes a parallel isosurfacing method for FEM data by distributing working blocks to a number of connected workstations [9]. Shen, Hansen, Livnat and Johnson give a sequential and parallel algorithm called isosurfacing in span space with utmost efficiency (ISSUE) [18]. Parker et al. present a parallel isosurface rendering algorithm using ray tracing [17]. Our approach parallelizes the accelerated isocontouring in both the seed set generation and isosurface extraction phases for multiple isocontour queries. A very important issue of parallel computation is load balancing [6] that can be achieved mostly with two fundamental approaches: (i) static balancing, where the data is partitioned priori with criteria that should guarantee load balancing at runtime [15], or (ii) dynamic balancing, where processors are given small chunks of data as they become available [8]. The partitions can take the shape of slices, shafts, or slabs [16].

Moreover, data can be so large that it cannot even be loaded into the primary memories of a large parallel computer. For example, the entire size of visible human female cryogenic data is larger than $16GB$. For this case we introduce a combined out-of-core/parallel computation scheme that scales with the number of processors and main memory available to take full advantage of the available hardware. In order to minimize secondary memory accesses in multiple queries we partition the data by its function value. For the isocontour queries in a certain range, only cells intersecting this range are read from secondary memory into primary memory. With a good partition all cells of one range may fit into the primary memory of parallel computer so that multiple queries within such range can be processed without extra accesses to disk.

The data partitioning is based on a static analysis that aims to maximize data coherency in functional space to achieve an efficient tradeoff between (i) load balance in parallel computations and (ii) minimal access to secondary memory in out-of-core computations. We have implemented our parallel, accelerated isocontouring algorithm for multiple queries of large datasets on Cray T3E. Our ap-

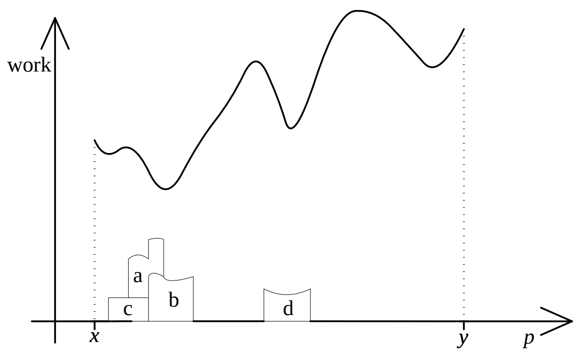


Figure 1: Static analysis diagram of algorithm $\mathcal{A}$ on a fixed dataset D with respect to the varying parameter p (horizontal axis). Notice that the overall diagram is the sum of the diagram computed for each cell $(a, b, c, d, \ldots)$.

proach however can be generalized to other visualization algorithms (e.g. multiple volume rendering) which have similar algorithmic characteristics.

The rest of this paper is as follows: Section 2 details our computational framework for achieving load balancing in parallel computations and minimal disk access for out-of-core computations. Section 3 describes the static data partitioning and reorganization specialized for the case of accelerated isocontouring algorithm. Section 4 provides details on our parallel implementation. Section 5 provides experimental results.

2 The Computational Framework

Consider a dataset D and an algorithm $\mathcal{A}(D, p)$ that takes as input D and a parameter p. We consider the problem of optimizing multiple evaluations of $\mathcal{A}(D, p)$ with a fixed D and different values of p. That is we allow a preprocessing $\mathcal{P}(D, \mathcal{A})$ to produce an evaluator $\mathcal{A}^D(p)$ that, for any p, produces the same output as $\mathcal{A}(D, p)$ but more efficiently. In the case of isosurface extraction, p is the value of isosurface query, and $\mathcal{A}^D(p)$ is the accelerated isocontouring algorithm. In particular we concentrate on the case where D is a mesh that is too large to be stored in the main memory of a single processor computer. Hence one major optimization problem is to partition D so that it is possible to achieve both (i) load balancing in parallel computations and (ii) minimal access to secondary memory in out-of-core computations.

For example we assume that p is a real number defined in the range $[x, y]$ and that D is a collection of small elementary units called cells. Our objective is to build a diagram as in Figure 1, by which one can estimate the cost of execution of the algorithm $\mathcal{A}$ on the fixed dataset D for different values of p. The analysis is performed at the level of single cells of D so that it is possible to determine which cells are involved in the evaluation of $\mathcal{A}$ for a given parameter p. The diagram of D is the sum of the diagrams of its cells. In figure the diagrams of the cells a, b, c, d are added.

2.1 Load Balancing

The static analysis described above allows immediate evaluation of the quality of a data partitioning scheme in terms of load balancing during parallel computations. In fact the ideal load balancing for k processors would be achieved if the analysis histogram of the data assigned to each processor is same scaled version ($\frac{1}{k}$ times) of the global histogram. Figure 2 shows the ideal partitioning of D in the case of two-processors where the cells b, c, d are assigned to the first processor and the cell a is assigned to the second processor (and hence not summed on top of b and c). In this ideal data partition, each processor does the same amount of work for every value of the parameter p.

The first result here is that this analysis allows to evaluate the quality of a data partitioning scheme by comparing its diagram with the ideal one. In our algorithm we will use the ideal diagram to derive a data partitioning that balances the work load.

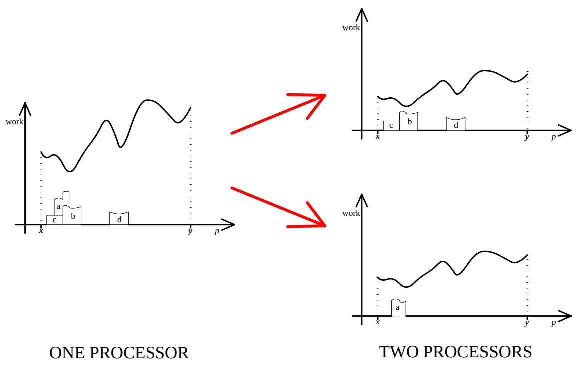


Figure 2: Optimal data partitioning for load balanced parallel computations (two processors case).

2.2 Minimal Secondary Memory Access

As in the case of parallel computations one can determine the diagram of an ideal data partitioning for out-of-core computations. In order to minimize disk access at execution of the algorithm $\mathcal{A}$ on multiple values of the parameter p, one can make use of the coherence of the parameter p. It's ideal that only the cells relevant to the evaluation of $\mathcal{A}$ on the current value of p are loaded from secondary memory into primary memory. Therefore we reorganize the data by range partitions such that we load only the cells in one partition for queries with in its range. Figure 3 shows the ideal partition for the case of three secondary memory blocks.

Again the ideal diagram can be used to determine the quality of any given partition. We precompute the diagram to help the construction of an actual partition trying to minimize the difference from the ideal one. We will describe how we do the actual partition for the isosurface computation in next section.

3 Accelerated Isocontouring Query Processing

In this section we demonstrate the ideas introduced in the previous section by applying the scheme to the accelerated isocontour query processing in which the parameter p is the isovalue query. Similar specialization could be done for multiple viewpoint volume rendering (p would be the vector of viewing parameters).

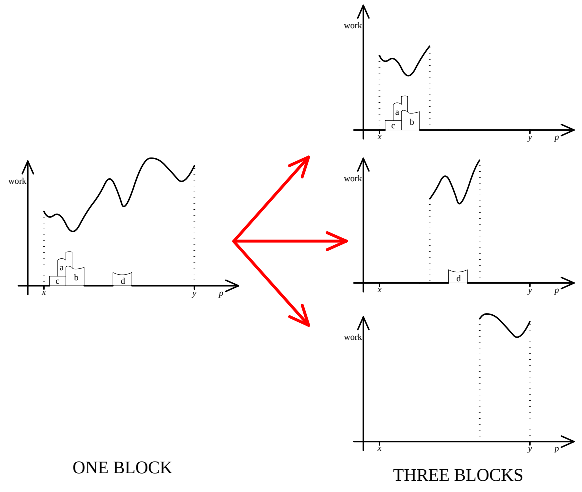


Figure 3: Optimal data partitioning for minimum disk access in out-of-core computations (three disk-blocks case).

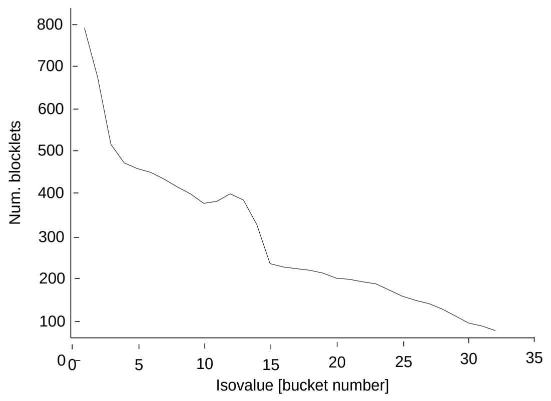


Figure 4: Static analysis histogram for a real dataset (foot of the visible human).

3.1 Static Analysis

The static analysis of isosurface extraction can be achieved by computing as pre-processing the *contour spectrum* [2]. The question here is to choose the appropriate signature function that represents the actual computation load. Here we consider the number of cells intersected with certain isosurfaces of value p. This is a piecewise constant function that can be computed in linear time. Figure 4 shows the histogram for a real dataset.

3.2 The Greedy Decomposition Algorithm

The basic assumption we make is that the size of the disk blocks is much larger than the Blocklets, small sets of adjacent cells that we consider our atomic processing element. Moreover we assume that the disk blocks are much smaller than the main memory of each processor. This assumption is satisfied by our target machine Cray T3E.

In our algorithm, the range partitioning of the data is performed in a preprocessing stage and each partition is saved in a separate file. We build the range partitions in a way that makes the access of disk efficient. The load balancing

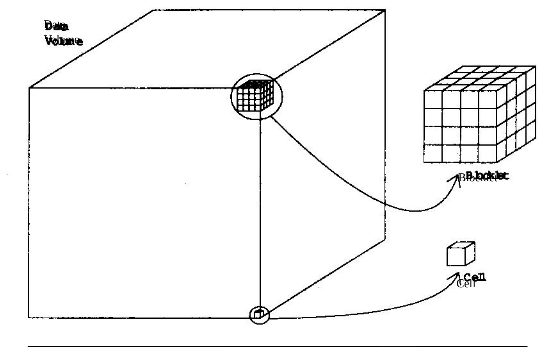


Figure 5: A regular grid volumetric dataset, its cell and the atomic processing element Blocklet.

of parallel computations is done using the unit of cell block (collection of blocklets) at the loading time. It can also be done at preprocessing stage if each processor has its own secondary memory. So the actual data decomposition algorithm can be described as a two-stage greedy scheme as follows.

In the first stage we decide the range partitions according to the total main memory and the contour spectrum as shown in Figure 4. For the blocklets of a range partition, a fixed number of blocklets are stored in each cell block that is integral multiple of disk blocks. Blocklets of similar spectrum are distributed among different cell blocks. If multiple choices are available one blocklet is chosen according to spatial coherence to blocklets already stored in the cell block. After this preprocessing stage the out-of-core decomposition is achieved.

In the second stage we aim for load balancing of parallel computation. On each processor a spectrum diagram is maintained for the blocks currently assigned to the processor. One by one the cell blocks are selected and assigned to the processor for which the spectrum has the most improvement with respect to the ideal case. Again if multiple choices are available we try to keep in the same processor blocks that are spatially coherent.

4 Implementation

In this section we describe some details of our implementation of parallel accelerated isocontouring on the Cray T3E. The atomic unit of data that is handled at data decomposition is called a *blocklet*. A blocklet is a small rectangular slab of cells and an associated offset into the original data volume recording where the cells were taken from. Blocklets will be collected into *cell blocks* (CBs), which are simply collections of blocklets stored on disk so that all the information necessary to generate isosurfaces for the cells is stored in one place (for fast disk access). The size of CB is chosen to be an integral multiple of disk block size of the machine.

Because large data sets may not always fit into main memory, the range of function values will be partitioned so that cells in the interval covered may all be loaded into main memory at once. This is called a *range partition* (RP). Range partitions allow for interactive isosurface visualization in main memory when isovalues are limited to the interval of values covered. As described in section 3, CBs will be formed by adding blocklets that reduce the variance of the resulting spectrum among cell blocks.

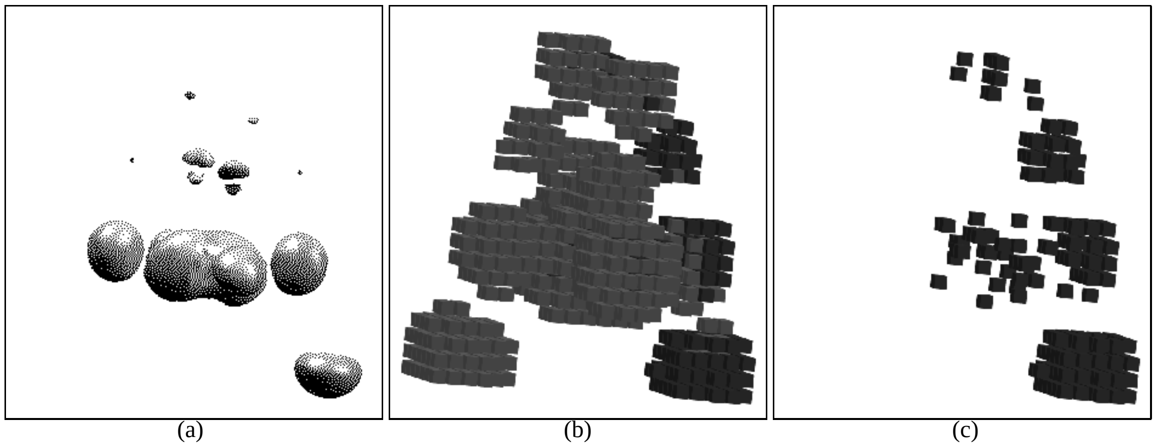


Figure 6: (a) An isosurface. (b) All the blocklets that are loaded in memory to compute the isosurface (two processors). (c) The blocklets processed by one of the processors.

Because the sampling of $w(\mathbf{x})$ is regular, the vertices may be indexed with a vector of integer coordinates $[m_0, m_1, \ldots, m_d]$. A range partition contains cell blocks which contain blocklets. Range partitions hold all the cell blocks necessary to cover some segment of the range of $w(\mathbf{x})$. Cell blocks are sized to match disk blocks and filled with blocklets so that the spectrum of a cell block matches that of the range partition. Blocklets will be stored with offsets, $[m_0^0, m_1^0, \ldots, m_d^0]$, from the global indices. This vector is used to perform the transform from local to global cell and vertex coordinates.

A triangular matrix is constructed to help the data partition, as shown in Figure 7. One axis of the matrix represents the function value over the entire domain which is divided into a specified number of *buckets*. The second axis is the number of buckets that a blocklet spans. In this way, the lowest function value and the number of buckets spanned become coordinates with which a blocklet ID is stored in the array. This array lets us create cell blocks that have spectrum similar to the whole dataset by evenly distributing all of the blocklets in each entry of the matrix across all cell blocks. Furthermore, this matrix lets us quickly identify all of the cells that span a given range of the function. This means that we may divide the matrix into a set of range partitions, each of which can fit entirely into main memory.

The number of buckets that partition the entire function value range is set by the smallest segment of the range that allows user interaction. If the range segment is too small, only few queries will fall into such range. The overhead of data duplication in different range partitions may outweigh the performance improvement from multiple isocontour queries.

The blocklets are sized so that each spans only a small portion of the total range of the data. The cell blocks should be sized so that there are much more of them than there are PEs. This is for better possible load balancing among the processors. Ideally, CBs would be sized so that each processor obtains the same number of CBs. Cell blocks should contain at least as many blocklets as there are entries in the triangular histogram matrix, so that if every entry in the matrix has many blocklets, the cell block can have a representative sample. In the current implementation, cell blocks are sized to be

$$n_{BL/CB} = \frac{b_{disk}}{\gcd(b_{BL}, b_{disk})}$$

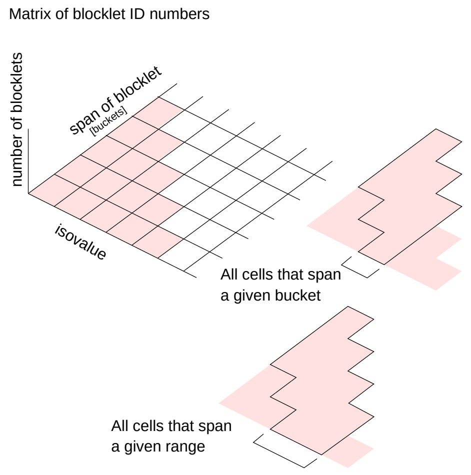


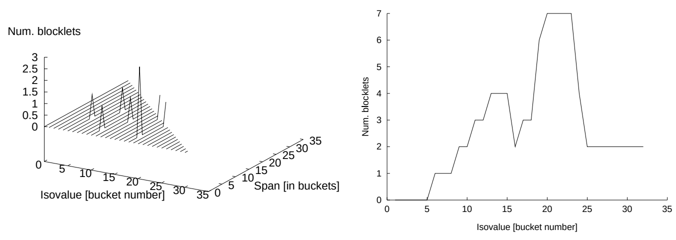
Figure 7: The storage structure for blocklet IDs.

where $n_{BL/CB}$ is the number of blocklets per cell block, b_{disk} is the number of bytes in a disk block, and b_{BL} is the number of bytes used to store a blocklet. If $n_{BL/CB}$ is less than the number of entries in the histogram table, it is doubled until this is no longer true. This ensures that no disk space is wasted and that a cell block can hold at least one blocklet from each entry in the histogram table.

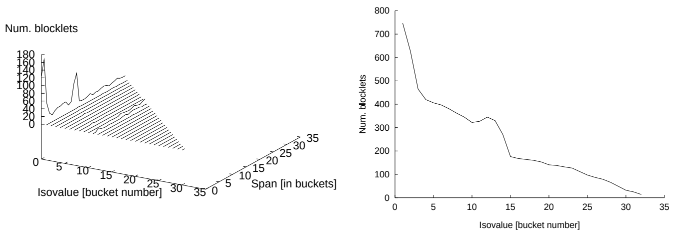
During pre-processing, blocklets are read by scanning through the data. As each blocklet is read, its minimum and maximum values are determined. From this, we find the bottom-most bucket of the histogram the blocklet spans, along with the total number of buckets that it occupies. These two values are used as indices to a position in the triangular matrix of blocklet IDs. Once this pass has been completed, the appropriate range partitions can be created.

Note that the blocklets that span any given isovalue may now be determined by visiting each matrix entry highlighted in Figure 7. By increasing the size of an initial partition while keeping it fitting into main memory, we can create range partitions that allow the largest range of isovalue queries.

Each range partition contains a list of cell blocks that span some bucket of the range. Since the number of block-



(a) A two-dimensional analytic function.



(b) The foot of the visible human male.

Figure 8: Some example triangular matrices and the resultant histograms.

lets of a certain range partition is known, we can compute the number of cell blocks required for a given range partition. By equally dividing the blocklet IDs stored at each entry of the triangular histogram matrix among all cell blocks, we create balanced cell blocks.

Some examples of number of blocklets stored in entries of the triangular matrix are shown for trial data in Figure 8.

After the preprocessing of the data, we statically assign cell blocks of a range partition to multiple processors in such a way that minimizes the spectrum difference among the processors as discussed in section 3. Each processor computes seed set for its own blocklets using the methods described in [1]. Based on the computed seed sets, processors can process multiple isocontour queries in the range using the accelerated isocontouring algorithm.

5 Experimental Results

We tested our algorithm on a Cray T3E of the Texas Advanced Computing Center(TACC). The Cray T3E is a multiple instruction multiple data (MIMD) machine which processors are toroidally–connected for message passing. The latency for message passing is approximately 100 clock cycles. Although the time required for one processing element (PE) to contact another PE may vary depending on the number of hops between the two PEs, the message passing library (MPI) hides this distance, so that data coherence cannot mimic spatial coherence (and thus minimize network traffic).

Another strength of the hardware is the large amount of memory available. Since each processing element (PE) has 128MB of main memory, there is a total of up to 7.5 GB of memory available in the largest configuration (with 60 application nodes). One obvious application for this machine is processing of large dataset. However, large data can require even more space than this, so we must allow for out-of-core visualization. During each clock cycle, the CPU can execute 2 floating point operations. Because floating point operations are the strength of the DEC Alpha it seems rational to implement an algorithm that takes real, rather than integral, scalar fields.

Figure 6 shows the results of running the code on a relatively small scalar field used for testing. Blocklets are $4 \times 4 \times 4$ vertices ($3 \times 3 \times 3$ cells) and there are only two cell blocks. Because of this, the number of blocklets in the second cell block is smaller. One aspect worth mention is that the spatial coherence of the blocklets where large portions of the blocklets in the same cell block are adjacent. This also reduces the number of seed cells required in the cell block. Figure 9 shows an isosurface of the visible human foot where each color in the final rendering highlights the contribution provided by each processor.

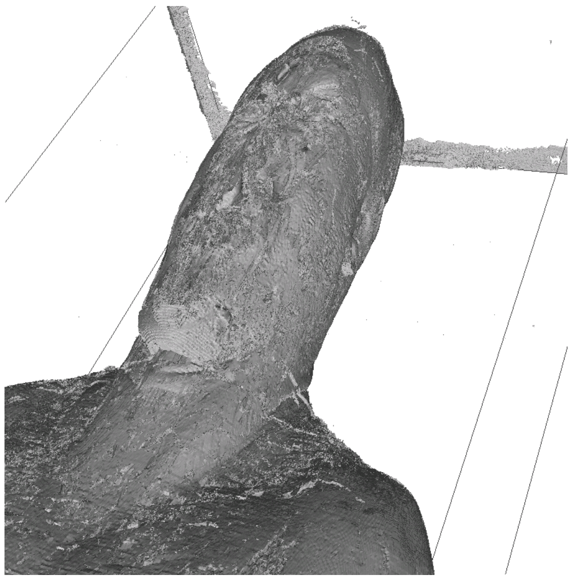


Figure 10: Isosurface with isovalue 117 (14,360,774 triangles) of the top part of the visible human.

Our intermediate size dataset is the top part of the visible human body. One large isosurface enclosing the body is shown in Figure 10. Figure 13(a-b) shows the load balance and throughput for 16,32 and 64 processors compared with the ideal case of balance (dashed lines). While the load balance still needs improvement, it is important to note that the maximum deviations from the ideal balance are due to processors that are under-used and there is no high pick. This corresponds to the histograms of each range partition shown in Figure 12 in thin solid lines compared with the ideal histogram drawn in thick dashed line. The consequence is that the total time necessary to compute an isocontour does not deviate too much from the time of ideal load balance because it is the time of the last processor that terminates the computation.

The speedup chart in Figure 14 shows the effect of the combination between parallel and out-of-core computation. Going from 16PE to 32PE and from 32PE to 64PE, the computation time is reduced by almost half because we have doubled the number of processors. The 32-processor run is slightly faster than a linear speedup. This could have several

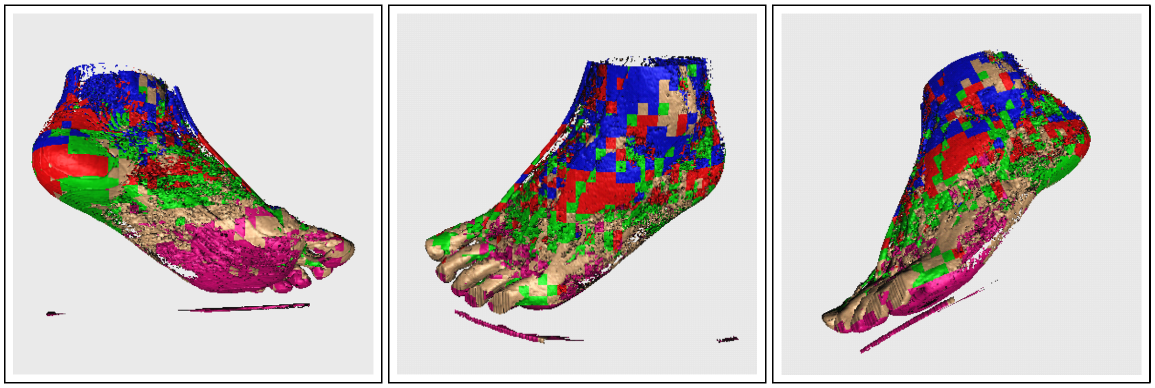


Figure 9: Isosurface of visible human male foot computed from the cryogenic image data. Color shows contribution of different processors.

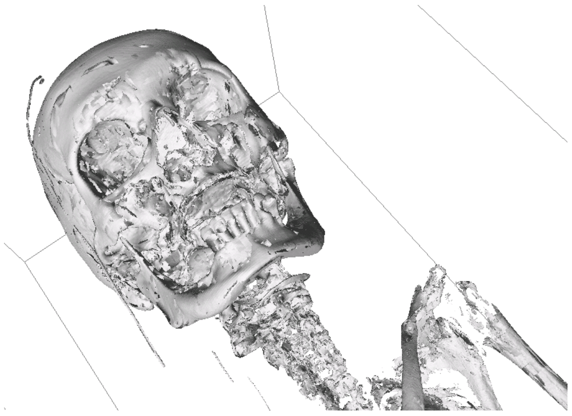
causes; first, since the program is run as a batch job, it must compete with other jobs for limited disk throughput. Also, it may compete with itself for disk throughput. As processors with fewer triangles finish contouring, they write their vertices (maintained in an AVL tree) to a file. This slows down processors still writing faces out to disk. There is a similar computational expense for seed cells that does not scale linearly with the number of processors. Running with half the number of processors does not imply that double the number of seed cells will be required because there will be more blocklets that share boundaries. Third, as the number of processors increases, the number of cell blocks allocated to each processor decreases and the slight imbalances in the load do not average out as well. This increases the variance of the output times. Fourth, the number of output vertices varies with the number of processors. Although the number of faces remains the same, when small numbers of processors are used, more blocklets are on the same processor and share output vertices. This means fewer total vertices to write than large numbers of processors but also means that more time is required to insert them because the vertex tree is larger. Finally, in addition to imbalance caused by less than ideally shaped histograms of cell blocks, some processors have one more cell block to process than others since the number of cell blocks is not an integral multiple of the number of processors. When the total number of cell blocks per processor is low (as it is for high numbers of processors), the imbalance can significantly impact the contouring time.

The analysis for the entire dataset produces similar results. Figure 15 shows a sequence of rectangles that bound the minimum and a maximum histograms of each partition (there are too many partitions to show each histogram as in Figure 12). Notice that even in this case large partitions tend not to exceed the ideal average value (small circles).

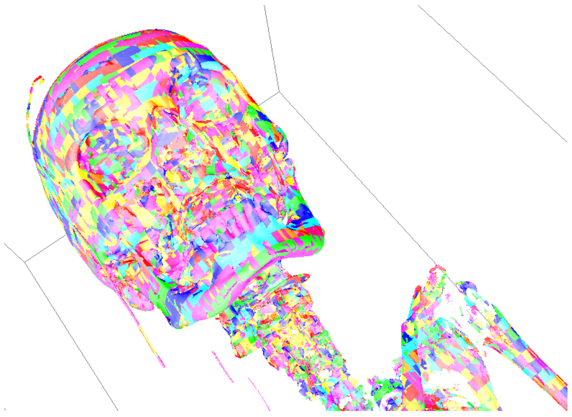
We also test our algorithm on other datasets, such as the female ct data. Portion of an isosurface is shown in Figure 11, where triangles are displayed with different color if generated by different PE in Figure 11(b).

6 Conclusion and Future Work

In this paper we have introduced a scheme for static analysis of large datasets to address simultaneously the problems of obtaining load balance in parallel computations and minimal secondary memory access in out-of-core computations. We analyzed the histogram of the entire dataset and

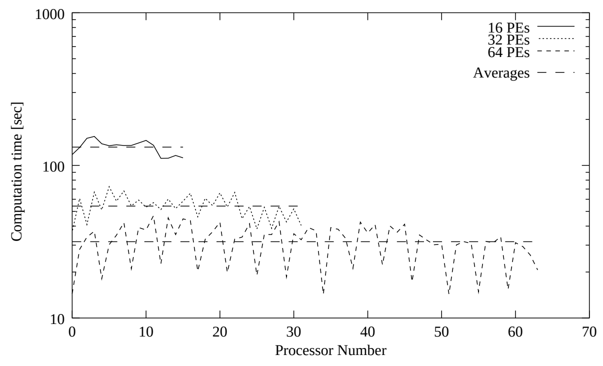


(a) shaded isosurface (isovalue 1550)

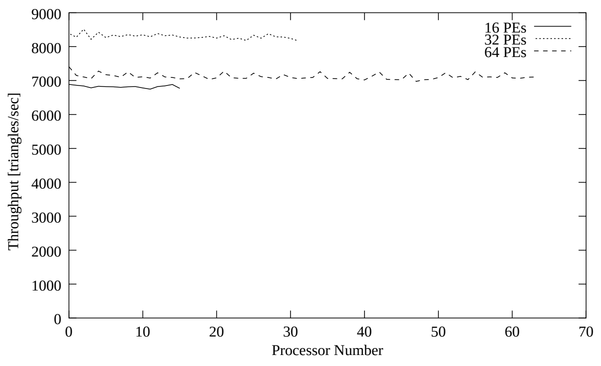


(b) isosurface colored by PEs (isovalue 1550)

Figure 11: Isosurface of female ct data at isovalue 1550. (a) Shaded isosurface (b) Different coloring of the surface corresponding to output generated by different PE.



(a) Load Balancing (isovalue 117)



(b) Throughput (isovalues 117)

Figure 13: (a) Experimental load balancing for a single isosurface computation (isovalue 117) with 8, 16 and 32 processors (solid lines) compared with the ideal cases (dashed lines). (b) Throughput (number of triangles computed per second) for multiple isosurface extractions (isovalues 117).

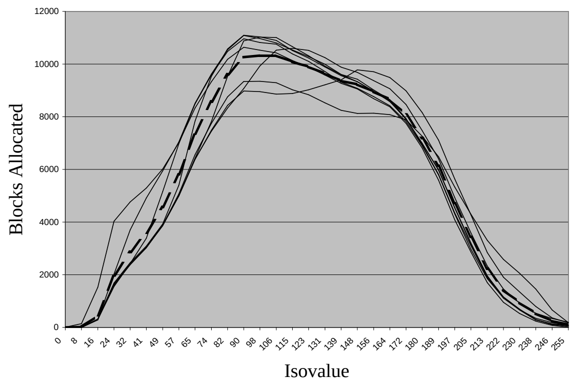


Figure 12: Histograms of the range partitions for the top part of the visible human (solid thin lines) compared with the ideal case (dashed thick line).

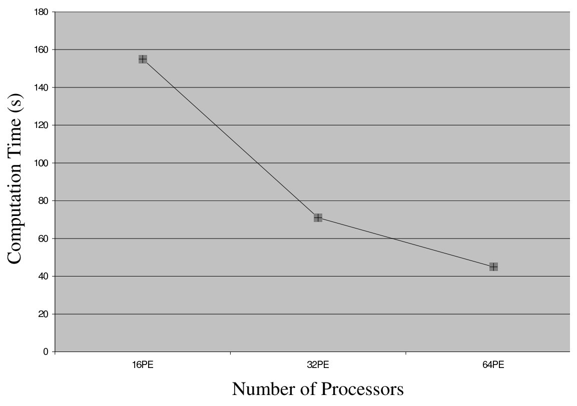


Figure 14: Speedup of the isosurface extraction (isovalue 117) for 16, 32 and 64 processors. The speedup is due both to the increased number of processors and to the reduced rate of I/O.

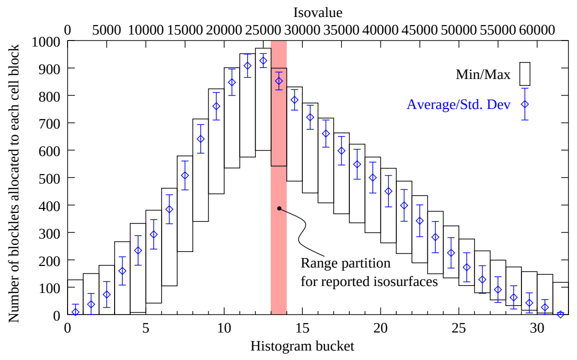


Figure 15: Envelope of the histograms of the range partitions for the entire visible human. The ideal case is marked by the small circles.

construct cell blocks that have roughly the same histograms (scaled by the number of cell blocks in the partition) across all isovalues in a given range. Even if the entire dataset is too large to fit in main memory, one range partition built in this way may be small enough. If even a single range partition is larger than the main memory, then its cell blocks will be swept through, loading as many as possible at once. This avoids loading the cell blocks that belong to range partitions that do not contribute to the computation of the current isosurface. We plan to improve this traversal by storing the cells in blocks of similar values, with cell blocks ordered by increasing value. In this manner, as few cells as possible would be traversed when a range could not be loaded into primary memory.

When the bucket size for the histogram is small and the data is still too large to fit in main memory, we can no longer be concerned with interactive exploration; our sole task becomes extracting an isosurface as efficiently as possible. Note that while this may be slow, it is still accelerated since only cells in a narrow range will be loaded. Also, since the problem is I/O bound in the out-of-core case, we can improve the performance significantly if the PEs can perform I/O in parallel.

Acknowledgments: This research is supported in part by grants from NSF-CCR-9732306, NSF-KDI-DMS-9873326, DOE-ASCI-BD-485, and NASA-NCC 2-5276.

References

[1] BAJAJ, C., PASCUCCI, V., AND SCHIKORE, D. Fast Isocontouring for Improved Interactivity. In *Proceedings of 1996 Symposium on Volume Visualization* (San Francisco, CA, 1996), pp. 39–46.

[2] BAJAJ, C., PASCUCCI, V., AND SCHIKORE, D. The contour spectrum. In *Proceedings of the 1997 IEEE Visualization Conference* (October 1997), pp. 167–173.

[3] CHIANG, Y., AND SILVA, C. T. I/O optimal isosurface extraction. In *IEEE Visualization 9́7* (Nov. 1997), R. Yagel and H. Hagen, Eds., IEEE, pp. 293–300.

[4] CHIANG, Y.-J., SILVA, C. T., AND SCHROEDER, W. J. Interactive out-of-core isosurface extraction (color plate p. 530). In *Proceedings of the 9th Annual IEEE Conference on Visualization (VIS-98)* (New York, Oct. 18–23 1998), ACM Press, pp. 167–174.

[5] CIGNONI, P., MARINO, P., MONTANI, C., PUPPO, E., AND SCOPIGNO, R. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics 3*, 2 (1997), 158–170.

[6] CROCKETT, T. Parallel rendering. Tech. rep., ICASE, 1995.

[7] CROSSNO, P., AND ANGEL, E. Isosurface extraction using particle systems. In *Visualization '97 Proceedings* (Phoenix, AZ, October 19–24 1997), R. Yagel and H. Hagen, Eds., pp. 495–498.

[8] ELLSIEPEN, P. Parallel isosurfacing in large unstructured datasets. In *Proceedings of the Fifth Eurographics Workshop on Visualization in Scientific Computing* (1994), M. Gobel, H. Muller, and B. Urban, Eds., Springer-Verlag, pp. 9–23.

[9] ELLSIEPEN, P. Parallel isosurfacing in large unstructed datasets. In *Visualization in Scientific Computing* (1995), Springer-Verlag, pp. 9–23.

[10] GALLAGHER, R. S. Span filtering: An efficient scheme for volume visualization of large finite element models. In *Visualization '91 Proceedings* (Oct. 1991), G. M. Nielson and L. Rosenblum, Eds., pp. 68–75.

[11] HANSEN, C., AND HINKER, P. Massively parallel isosurface extraction. In *Visualization '92* (September 1992).

[12] ITOH, T., AND KOYAMADA, K. Isosurface generation by using extrema graphs. In *Proceedings of Visualization '94 (Washington, DC, October 17–21, 1994)* (Oct. 1994), D. Bergeron and A. Kaufman, Eds., IEEE Computer Society, IEEE Computer Society Press, pp. 77–83.

[13] LIVNAT, Y., SHEN, H., AND JOHNSON, C. A near optimal isosurface extraction algorithm for unstructured grids. *IEEE Transactions on Visualization and Computer Graphics 2*, 1 (1996), 73–84.

[14] LORENSEN, W. E., AND CLINE, H. E. Marching cubes: A high resolition 3d surface construction algorithm. *Computer Graphics 21* (1987), 163–169. *SIGGRAPH '87 Proceedings*, M. C. Stone, ed.

[15] MIGUET, S., AND NICOD, J.-M. A load-balanced parallel implementation of the marching-cubes algorithm. Tech. Rep. 95-24, Ecole Normale Supérieure de Lyon, October 3 1995.

[16] NEUMANN, U. Comunication costs for parallel volume-rendering applications. *IEEE Computer Graphics and Applications* (July 1994), 49–58.

[17] PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P. Interactive ray tracing for isosurface rendering. In *Visualization '98* (October 1998).

[18] SHEN, H., HANSEN, C., LIVNAT, Y., AND JOHNSON, C. Isosurfacing in span space with utmost efficiency (issue). In *Visualization '96* (1996), pp. 287–294.

[19] SHEN, H., AND JOHNSON, C. Sweeping simplices: A fast iso-surface extraction algorithm for unstructured grids. In *Visualization '95 Proceedings* (Oct. 1995), G. M. Nielson and D. Silver, Eds., pp. 143–150.

[20] VAN KREVELD, M. Efficient methods for isoline extraction from a digital elevation model based on triangulated irregular networks. *To appear, International Journal of Geographical Information Systems* (1996). Also appeared as Technical Report UU-CS-1994-21, University of Utrecht, the Netherlands.

[21] WILHELMS, J., AND VAN GELDER, A. Octrees for faster isosurface generation. *ACM Transactions on Graphics 11*, 3 (1992), 201–227.