

Isosurface Extraction SIMD Architectures

Charles D. Hansen Paul Hinker
Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, New Mexico 87545
hansen@acl.lanl.gov hinker@acl.lanl.gov

Abstract

We describe our experiences with the investigation of parallel methods for faster isosurface generation on SIMD machines. A sequential version of a well known isosurfacing algorithm is algorithmically enhanced for a particular type of SIMD architecture. The SIMD implementation takes full advantage of the inherent data parallelism in the algorithm and experiments have proven the implementation to excel in terms of scalability. Having such a parallel tool, interactivity is substantially enhanced. This provides an easy to use isosurfacing back-end for scientists wishing to explore 3D scalar and vector fields.

1 Introduction

One of the most common methods for the visualization of 3D scalar fields is through the reconstruction of constant valued surfaces. These fields can be scalar components of vector fields, a scalar computed from the vector components, 3D medical imaging data, seismic data or a plethora of other types of data[1, 2, 3]. Typically, this process treats the 3D scalar field as gridded, possibly non-uniform, point samples and for a given constant value, interpolates intersections between neighboring pairs of these samples. Connecting these intersections will approximate a surface at some given constant value.

Visualizing these isosurfaces interactively can give the scientists a great deal of information about the underlying structure contained within the field. Interactive viewing of these surfaces enhances the interpretation of this structure. Unfortunately, the size of these fields generated from supercomputer models is usually very large. This limits the interactivity of isosurfacing algorithms due to the amount of time spent generating the surface of constant value. In this paper, we describe our experiences with a SIMD solution to this problem.

Isosurface generation can be accelerated through the use of table lookup[1]. Using this method, known as Marching Cubes, the vertexes of each voxel¹ are compared against the contouring value. If they are greater, a corresponding bit is set in a mask representing that voxel. This mask is used as an index into a table describing which edges contain the intersection.

¹a voxel, volume element, is composed of 8 point samples forming a cube

Although much faster than the brute-force method, one still needs to interpolate the points of intersection on the corresponding voxels. This interpolation is typically performed many times and therefore slows the process down.

Researchers have looked at hierarchical data structures to aid in addressing this problem. One such data structure is the octree[4]. Researchers have used octrees in medical imaging to avoid transparent spatial regions[5]. Others have addressed faster isosurface generation through the use of octrees[3]. This method hierarchically subdivides the 3D scalar field noting the upper and lower values contained within the region. This speeds the isosurface generation procedure by only looking for voxels within pertinent regions.

The assumption that the entire 3D field is resident on the machine generating the isosurface. In medical imaging, this assumption is valid since the scanning device is not usually the same as the visualization device and the data is usually visualized in a postprocessing setting. The transfer, possibly through a network, of the sampled data is necessary and typical in this scenario. However, in scientific computing the simulation models are running on a high-performance computer and the size of the fields (up to 1G byte per time step) impedes the transfer of data to the visualization device. This limits the interactivity of isosurfacing when monitoring running models. One approach to this problem is to generate the isosurface on the same machine which is running the model and either render the geometry locally or send the geometry to a high speed rendering device such as an SGI 380/VGX.

We have chosen to investigate parallel methods for faster isosurface generation. This paper describes the results we have obtained through experiments with a SIMD parallel version of marching cubes [1]. We examine the SIMD method, why we chose the particular algorithmic approach and present results obtained utilizing a 64K CM-2. We then discuss the conclusions of our findings.

2 A SIMD Approach

The architecture targeted for the SIMD version of the marching cubes algorithm is a 64K processor Thinking Machines Connection Machine (CM-2). Although the marching cubes algorithm maps well to the hardware, some tuning was required to achieve the expected performance from the code.

Even though there are 65536 physical processors, each processor can simulate many virtual processors (vp). This allows us to assign a vp to each voxel(1) and, in concept, concurrently determine which edges are intersected by a constant valued isosurface. Using this intersection information, it is possible to construct the bit-mapped index into marching cubes tables and construct the polygon(s) contained in each voxel.

The entire algorithm breaks into a few discrete steps.

- Create bit-mapped index
- Interpolate edge intersections & gradients
- Construct triangles
- Calculate normals

2.1 A More Detailed Description of the SIMD Algorithm

The first step is creating a bit-mapped index into the marching cubes lookup tables. In our implementation of the algorithm, the data points of the volume are located at the lower-left-forward corner of each voxel. To complete the description of each voxel, we need point samples (vp values) for the other seven corners of the voxel. At this point, a choice must be made between storing all eight point samples 'in-processor' or communicating that information as needed. An additional consideration is the fact that if corner information is stored in-processor, then interpolated edge and gradient information would also need to be stored in-processor. The memory requirements between the two methods differ by a factor of almost 3 to 1 in favor of doing the communications.

On the surface, doing communications seems to be the correct way to go. There are additional considerations, however. Because of the nature of the data sets being dealt with, a 3 to 1 memory usage difference is not really significant. The CM-2 requires that any axis be of integral power of two length. This means that the next volume larger than 32x32x32 voxels is a volume of 64x32x32 voxels. Even if the actual volume is 33x32x32 voxels in size so, chances are, if the memory is insufficient for the algorithm where everything is stored in-processor, it will be insufficient for the communication intensive algorithm as well.

Also, even though all communications, in practice, is nearest neighbor communications (the fastest type on the CM-2) the C-Star compiler is not optimized enough to recognize that fact. This means that triangle construction is done using general (via router) communications which is an order of magnitude slower. So, each vp communicates with its neighbors and assigns values to an eight position parallel array V[0:7] that describes the point sample values at each of its eight vertices.

Determining the bit-mapped index for the table lookups is straight forward since we have point sample data for all eight vertices in-processor. The index itself is constructed by simply setting the bits corresponding to the vertices which are greater than or equal to the value being surfaced.

Figure 1: cube numbered vertices and edges

Once the bit-mapped index is created, we calculate the gradient information along the x, y and z axis to use in our shading model when the time comes to render our surface. The gradient can be linearly interpolated at the point of intersection. We estimate the gradient vector at the surface of interest by first estimating the gradient vectors at the cube vertices using the operator :

$$\nabla f(x_i) = \nabla f(x_i, y_j, z_k)$$

Next, we calculate edge intersections along the three coordinate axes by linearly interpolating edge intersections for all three coordinate axes. To handle non-uniform grids, this interpolation will use voxel coordinate information instead of assuming unit cube voxel sizes. Gradient values at surface intersection points are calculated in much the same way during this step. This only entails a single interpolation for each of the three intersections for each of the coordinate axes. In the SIMD architecture, each vp knows 'where' in the compute space it resides. In other words a vp can be, effectively, asked 'What is your X-Axis position?'. This is accomplished with the pcoord() (C-Star) function or the MY-NEWS-COORDINATE (C-Paris) function.

```
Delta = IsoValue - V[0]
```

```
For (I = 0 : I 3 ; I++)
```

```
  For (J = 0 : J 3 ; J++)
```

```
    Edge[I][J] = pcoord(J);
```

```
Edge[0][0] = (Delta / (V[1] - V[0])) + pcoord(0)
:: X-Axis
```

$$\text{Edge}[1][1] = (\text{Delta} / (\text{V}[3] - \text{V}[0])) + \text{pcoord}(1)$$

:: Y-Axis

$$\text{Edge}[2][2] = (\text{Delta} / (\text{V}[4] - \text{V}[0])) + \text{pcoord}(2)$$

:: Z-Axis

Some of the edges calculated are not useful but because of the lock-step nature of the SIMD hardware, all vp's must wait until the last vp is finished. So, the overhead of choosing and eliminating useless edge calculations turns out being more expensive than just doing the interpolation for every edge.

Now we have intersection and gradient information for the twelve edges which make up each voxel. By doing NEWS communication with neighboring voxels, we pick up the nine edges interpolated by the neighboring vp's.

It's necessary for each vp, that contains a portion of the surface, to have access to the lookup table when constructing polygons. There are two widely used forms for this table. One form has a unique entry for each possible value of the bit-mapped index (i.e. 256 entries). The other form reduces the table size by applying rotations and symmetry to 14 base triangle configurations[1].

We make use of a full sized lookup table because the reduced table method divides any given index into categories. It is either one of the 14 base cases or it requires rotation, reflection or both. The lock-step nature of the CM-2 hardware makes this an expensive proposition since all processors are given the same instruction whether they are active or not. Either vp's doing base, rotation or reflective construction are handled while all other vp's sit idle. Using the full table means that all indices fall into one category and, therefore, are handled all at once.

It would be prohibitively expensive to store the entire lookup table in each vp's memory space. For example, the lookup table requires 3328 bytes of storage. Memory for each physical processor is divided evenly between the vp's associated with that physical processor. Each physical processor has 256K of memory. A vp ratio (the number of vp being emulated by each physical processor) of 2 means that each vp has 128K of memory with which to work. Storing the entire lookup table on each vp is not a problem when working with small vp ratios (i.e. small volumes). A medium sized volume of 128 x 128 x 128 requires a vp ratio of 64. Storing the table on each vp means that the lookup table would use 81.25processor memory (3328 bytes * 64 = 208K. It also means that even though the CM-2 is equipped with 8 GB of RAM, we could not surface a volume of greater than 128³ cells because the memory required for the table alone would be greater than 8 GB.

One solution would be to store the table on the CM front-end machine as a scalar array and broadcast the table entries to the CM as needed during the polygon construction stage of the algorithm. Unfortunately, there are two penalties to having the table in scalar form on the front-end. First, the communication time is significant compared with the runtime of the algorithm. Also, since the table is a scalar array,

PEs	Volume Sz	Indexing	Edges	Construct	Normals
16K	32x32x32	4.2	6.6	41.4	9.4
16K	64x64x64	18.7	30.6	322.3	60.9
16K	128x128x128	98.8	196.7	2550.3	472.5
32K	32x32x32	2.8	4.1	25.6	5.5
32K	64x64x64	10.8	16.8	169.5	31.2
32K	128x128x128	55.0	102.8	1280.4	237.1
64K	64x64x64	6.5	9.9	87.8	16.5
64K	128x128x128	31.3	54.8	640.0	119.4
64K	256x256x256	175.5	375.2	5088.7	864.0

Table 1: Results of Runs on the CM-2 in milliseconds

it cannot be accessed by the CM hardware in a parallel manner. This means a large number of vp's are idle during much of the polygon construction stage. It's necessary to loop over all possible index values (256) and activate the corresponding vp's to do their polygon construction.

A more reasonable solution for the problem is using the aref32_shared & aset32_shared instructions provided by the CM-2 hardware.

A CM-2 is made up of some large number of serial-bit processors (4K, 8K, 16K, 32K, 64K) these processors (along with routing hardware, Weitek floating point math chips and memory) are grouped into what are called Sprint Nodes. Each Sprint Node has a Weitek math chip, 32 bit processors, routing hardware and (in our CM-2) 32 M-bits of RAM. The aref32_shared & aset32_shared instructions allow us to use part of the RAM on the Sprint Node as a shared memory accessible by any physical processor on the Sprint Node and any virtual processor associated with those physical processors. Storing the lookup table in this manner requires a constant amount of memory per physical processor (i.e. $\text{tablesize}/32$). This allows us to store the full sized table in CM memory and access it in a parallel manner.

The last step in the process is to calculate normals for each triangle constructed. We have modified the lookup table so that all triangles are constructed in a *clockwise traversal* method. This allows us to generate normals quickly without having to determine proper normal direction.

3 SIMD Results

The times given in the table 1 were gathered using a Sun 4/490 front-end and the number of processing elements (PEs) specified in the first column. All times given are in milliseconds. Each timing is the average of three runs using different isosurface values. Changing the value to be surfaced had little effect on execution time. This is as expected since the time spent in each virtual processor (vp) remains constant regardless of the number of polygons within its voxel.

A close look at the timings shows that doubling the number of PEs reduces the time needed to generate the surface by roughly half while multiplying the number of voxels in the volume by eight roughly increases the execution time by that same amount. As the vp ratio (number of virtual processors being emu-

lated by each physical processor) increases, efficiency is increased. This explains why the speedup (with respect to vp ratio) seems to be slightly better than linear. The speed up is slightly less than linear when more PEs are applied to the same data set. This can be explained using the same argument. Since the vp ratio decreases when more PEs are applied to the same data set, PE efficiency suffers since more PEs are idle for more of the time.

Some readers may think the times shown in ref:fig:simd are slower than would be expected. First, the data sets we are using, from a CFD origin, contain a large number of polygons. For example, the 128x128x128 volume generates over 155,000 triangles per surface. The 256x256x256 volume over 1.24 million triangles. The algorithm outlined here will generate on the order of 170,000 triangles per second. This is close to the rendering limit of disjoint Goraud shaded on the fastest commercially available hardware (SGI VGX with MultiBuffer). Thus, within current hardware limits, this is nearly optimal for interactivity.

4 Conclusion and Future Work

As the above results show, the scalability of the marching cubes isosurface extraction algorithm is nearly optimal for the data parallel architecture of the CM-2. With the size of the massive data sets currently computed, this is an issue that is becoming increasingly important. Our results are based on isosurfacing without using spatially hierarchical data structures. It is unclear whether such an approach would improve performance of this particular implementation due to the lock step nature of the SIMD architecture. This is clearly a direction for further research. Another issue not addressed this specifically addressed by this paper is the rendering of the generated polygons. As previously mentioned, one of the motivations for generating the isosurface on the same machine where the raw data is produced is to reduce the required network bandwidth to interactive levels. For a 256x256x256 volume of floating point data, the raw data requires 530Mbits per time step. Considering that dynamic simulations contain hundreds of time-steps, this is obviously too much raw data to transport in the typical visualization process. If 100K polygons (triangles) are extracted, the data shipped over the network is reduced to 28Mbits. While this is a reduction of almost 19 times, it is still twice maximal ethernet bandwidth. However, we can utilize the CMIO bus to VME adapter to accomplish the network transport. Another solution would be to make use of the HIPPI channel via a HIPPI-VME adapter. If the polygon count increases an order of magnitude, rendering the polygons on the CM-2 becomes the best solution.

Our current area of focus is in attempting to somehow reduce the number of triangles generated by this code. One method would be the merging of coplanar polygons. This met with resistance on several fronts since part of the method is stubbornly serial in nature (retransversal of merged polygons). Also, very complex polygons are generated when many polygons

are merged (polygons with multiple holes, non-unique segments, etc.) And finally, there is a performance penalty when trying to render complex polygons (as opposed to triangles or quads) on z-buffered hardware such as an SGI VGX.

In the paper, we have described a technique for the implementation of isosurface extraction on a data flow SIMD architecture. We have shown that near linear speedups, and in some cases superlinear speedups, are possible on a real-world, well known algorithm.

5 Conclusions

As the above results demonstrate, the scalability of isosurface extraction is much greater for the data parallel architectures such as the CM-2. With the size of the massive data sets currently computed, this issue becomes more important. Our results are based on isosurfacing without using spatially hierarchical data structures. As noted previously, these have been useful to other research in obtaining speedups. With the MIMD version, spatial decomposition might improve the performance even further. This is clearly an area of future research. Conversely with the SIMD version, spatial decomposition would not improve the performance since all processors execute the same instruction in a SIMD fashion. One issue that we have not addressed in this paper is the rendering of the polygons. On the SGI 380/VGX, it is clearly preferable to utilize the graphics pipeline. However on the CM-2, the delay in extracting the polygonal information from the CM-2 memory might present problems. One solution would be to render the polygons on the CM-2. Another would be to utilize a CMIO-VME connection. Still another would be to utilize the HIPPI channel via a HIPPI-VME adapter.

In the paper, we have described a technique for the implementation of isosurface extraction on both MIMD and SIMD architecture. We have shown that the speedup is nearly linear on the SIMD machine while on the MIMD version, the speedup factor decreases once the processor count is over four. We have shown that by attempting to maintain the cache hit ratio on the MIMD architecture, greater performance can be achieved.

Acknowledgements

This work was partially funded by DOE High Performance Computing Grant KC0701, LANL LDRD 91-Distributed-Visualization, LANL Computing Division, and NSF STC

References

- [1] W. Lorensen and H Cline. A high resolution 3d surface construction algorithm. In *Computer Graphics*, volume 21, pages 163–169, 1987.
- [2] W. E. Johnson et. al. Distributed scientific video movie-making. In *Proceedings of Suprtcomputing Conference 1988*, pages 156–162, 1988.
- [3] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. In *SIGGRAPH Workshop on Volume Visualization*, pages 129–147, 1990.

- [4] D. Meagher. Geometric modeling using octree encoding. In *Computer Graphics*, volume 19, pages 129–147, 1982.
- [5] Mark Levoy. Efficient ray tracing of volume data. *ACM Transactions of Computer Graphics*, 9(3), July 1990.