The Australian National University

Computer Science Technical Report

August 1992 Technical Report TR-CS-92-10

A Fast Parallel Marching-Cubes Implementation on the Fujitsu AP1000

Paul Mackerras

Department of Computer Science
The Faculties

Computer Sciences Laboratory
Research School of Physical Sciences and Engineering

Joint Computer Science Technical Report Series

A Fast Parallel Marching-Cubes Implementation on the Fujitsu AP1000

Paul Mackerras
Department of Computer Science
Australian National University

August 1992

Abstract

Parallel computers hold the promise of enabling interactive visualization of very large data sets. Fulfilling this promise depends on the development of parallel algorithms and implementations which can efficiently utilize the power of a parallel computer. Fortunately, many visualization algorithms involve performing independent computations on a large collection of data items, making them particularly suitable for parallelization.

This report describes a high-performance implementation of the Marching Cubes isosurface algorithm on the Fujitsu AP1000, based on a fast serial Marching Cubes implementation. On a 128-processor AP1000, our implementation can generate an isosurface for a volume of reasonable size (e.g. 2.6 million data points) in typically less than 0.5 seconds (depending on the number of polygons generated).

The Fujitsu AP1000 is an experimental large-scale MIMD (multiple-instruction, multiple data) parallel computer, composed of between 64 and 1024 processing cells connected by three high bandwidth, low latency communications networks. Each processing cell is a SPARC processor with 16MB of memory. The cell processors do not share memory.

Our experience indicates that the Marching Cubes algorithm parallelizes well; in fact the speedup we obtain is actually greater than the number of processors (presumably due to cache effects). However, it is necessary to perform any further processing of the generated surface (such as rendering, or evaluation of connected volumes) in parallel if massive slowdowns are to be avoided.

1 Introduction

One standard method of visualizing 3D scalar field data is to generate and display isosurfaces—surfaces where the field value is constant. The 'Marching Cubes' algorithm [5] generates a polygonal description of an isosurface for data sampled on a rectangular grid. The algorithm considers each unit cube defined by a $2 \times 2 \times 2$ grid of sample points. If some of the samples at the corners of the cube are above the isosurface value and some are below, then the isosurface passes through the cube. For all such cubes, the algorithm uses linear interpolation to find the intersections of the surface with the edges of the cube, and then connects the intersections to

form polygons. These polygons are in general non-planar, but can be converted into triangles for rendering. In addition, the algorithm optionally estimates the gradient of the scalar field data at each intersection, and uses this as the surface normal vector at that point on the surface. This gives a smooth, rather than a faceted appearance to a curved surface.

In this paper, the term *cube* refers to a region of volume defined by a $2 \times 2 \times 2$ grid of sample points. An *edge* is the line connecting two adjacent sample points, and a *face* is the area bounded by s 2×2 grid of sample points. An *intersection* is the point on an edge where the isosurface crosses it. The values of the sample points are denoted $V(i, j, k)$, where i, j and k increase in the X, Y and Z directions respectively; $0 \leq i < N_X$, $0 \leq j < N_Y$, and $0 \leq k < N_Z$, where N_X, N_Y and N_Z are the dimensions of the volume data set. The cube at (i, j, k) is the cube defined by samples $(i + l, j + m, k + n)$, for $l, m, n \in \{0, 1\}$. The value of the isosurface is denoted by T. The sign of x relative to T is defined as the sign of $x - T$.

In order to determine how to connect the intersections into polygons, the Marching Cubes algorithm determines a *cube index* for each cube. The cube index is an 8-bit number; bit $l + 2m + 4n$ of the cube index at (i, j, k) is 1 iff $V(i + l, j + m, k + n) > T$. Note that the isosurface is unchanged if each sample is replaced by $T - V(i, j, k) + \epsilon$, for arbitrarily small positive ϵ, which inverts each bit of the cube index.[1] The 256 values of the cube index can thus be reduced to 128. In fact, these 128 cases can be reduced to 14 major cases by considering rotations and reflections of the cube; Lorensen and Cline [5] enumerate these 14 cases and specify a topology for connecting the intersections into polygons for each case. Our implementation does not use the 14 major cases.

An ambiguity arises when the isosurface intersects all four edges adjacent to a face, since the intersections can be connected more than one way (see figure 1). Such a face has two diagonally-opposite samples above T and two below T. The original Marching Cubes algorithm [5] chose a fixed, arbitrary topology for each major case, which can give rise to holes and other flaws in the surface. Other researchers have proposed various strategies in which additional information is used to direct the choice of topology[6][7]. Wyvill *et al.*[7] use the value at the center of the face, approximated by the mean of the sample values at the four corners of the face. The intersections are connected so as to cut off the corners which have the opposite sign to the center value (relative to the isovalue). Wilhelms and Van Gelder [6] consider a range of disambiguation schemes and suggest two new schemes. One of these schemes estimates the value at the center of the face using the field gradient values at the sample points, and then applies the same rule as Wyvill *et al.* The other involves the use of quadratic interpolation. Both methods give good results on their example functions, but are somewhat computationally expensive.

In our implementation, we ensure that the topology of the surface is consistent with the topology of the isosurface of a trilinearly-interpolated field. This ensures that the surface will be complete and topologically consistent. Our disambiguation scheme compares the coordinates of intersections on opposite edges of the face; each intersection should be connected to the adjacent

[1] Note that ϵ can be zero unless some of the samples are equal to T; its only purpose is to make sure that all bits of the cube index are inverted. Since it is arbitrarily small, it makes no significant difference to the position of the isosurface.

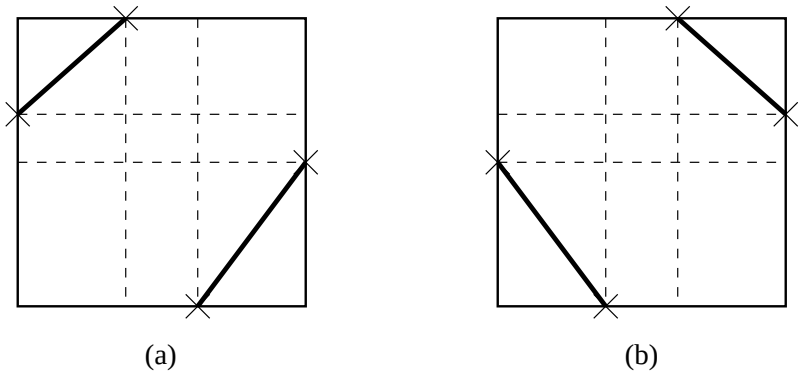

Figure 1: Disambiguation by comparing intersection coordinates

face which it is closest to, as shown in figure 1. The reasoning behind this is as follows. Assume that the face is in the X-Y plane, and that the field is trilinearly interpolated between the sample points (and therefore bilinearly within the face). The intersection of the isosurface with the face will then be a hyperbola of the form $(x-a)(y-b) = c$. If $c \neq 0$, the hyperbola never crosses the line $x = a$ or the line $y = b$, and for an ambiguous face, it can be shown that both lines intersect the face. Therefore the topology is one of the two options shown in figure 1, depending on the sign of c. The case $c = 0$ is a singularity which can be resolved to either of the two options in figure 1 by an arbitrarily small perturbation of the sample values.

The gradient at an intersection is estimated by linear interpolation from the gradient values at the adjacent sample points, which are calculated by a central difference formula ($G_X(i,j,k) = (V(i+1,j,k) - V(i-1,j,k))/2$, etc.) At the edges of the volume, there is no adjacent sample point, so the sample itself is used instead of the unavailable adjacent point (e.g. $G_X(0,j,k) = V(1,j,k) - V(0,j,k)$).

Our parallel implementation of the Marching Cubes algorithm is based on an efficient serial implementation. The volume is partitioned into contiguous blocks. Each processor is allocated one or more blocks, and runs the serial Marching Cubes code (essentially unchanged) on each block, producing a list of polygons. The complete surface is obtained simply by concatenating these lists of polygons.

2 The Fujitsu AP1000

The AP1000 [2][3][4] is an experimental large-scale MIMD parallel computer developed by Fujitsu Laboratories, Japan. Configurations range from 64 to 1024 individual processors or "cells", connected by three separate high-bandwidth communications networks: the B-net, T-net and S-net. The cells do not share memory. The AP1000 is connected to and controlled by a host computer which is typically a Sun SPARCServer.

Each cell consists of a SPARC processor running at 25MHz with 16MB of RAM (four-way interleaved, with ECC), 128kB of direct-mapped cache memory, floating-point unit, a message

controller, and interfaces to the three communications networks. The cells have no address translation hardware, but they do have a memory-protection table (MPT) which provides access and caching control for 4KB pages. The message controller provides DMA facilities for sending and receiving messages.

The B-net is a 32-bit, 50MB/sec broadcast network which connects all cells and the host, and is used for communication between the host and the cells. Using the B-net, the host can transmit data to one cell or to all cells simultaneously, and each cell can transmit data to the host. The B-net also supports scatter/gather operations.

The T-net provides cell-to-cell communication. It is arranged as a two-dimensional torus in which each cell has links to its four neighbours in a rectangular grid. The T-net is controlled by a Routing Controller (RTC) chip in each cell; its bandwidth is 25MB/sec on each link. Wormhole routing is used, with a structured buffer pool in each RTC to avoid deadlock. Each RTC will forward messages on toward their destinations without requiring any action by the cell CPU.

The S-net supports synchronization and status checking. It carries 40 signals which are ANDed together over all cells: all cells output a value for each signal, and receive the AND of the values output by all cells. To achieve synchronization, each cell sets a particular output to 1 and then waits until it sees the corresponding input at a 1. The S-net also allows the cells to synchronize with the host.

Programs for the AP1000 are written in either C or FORTRAN. Library calls are used for communication over the networks described above. Run-time control is provided in the cells by CellOS, the cell operating system, and on the host by the CAREN (Cellular Array Runtime ENvironment) program. CAREN provides facilities for run-time monitoring of cell activity, performance measurement, error logging and debugging. Symbolic debugging of cell tasks is provided through the use of GDB.

The AP1000 at the Australian National University has 128 cells. The cells have no I/O capability at present; all I/O is performed on the host. In the context of an isosurface program, this means that the host has to read the volume data and send it to the cells, then (if required) receive the isosurface representation from the cells and store it on disk. Similarly, an image calculated on the cells has to be transmitted to the host. Since the host does not have a color frame buffer, the image is usually then sent to a color workstation to be displayed. A distributed frame buffer will be installed on the cells within the next year.

3 Serial implementation

The potential speed of a parallel machine can only be fully realized if the algorithm being executed by each individual processor is as efficient as possible. With this in mind, we have designed an efficient serial implementation of the Marching Cubes algorithm, which is executed in parallel on the cells on disjoint sets of cubes.

Our implementation considers X-Y planes of the volume in order of increasing Z coordinate. It keeps information on two adjacent planes in memory at once; for each sample point, this information consists of a possible intersection on each of the three edges emanating from the

point (in the positive X, Y and Z directions), the estimated field gradient at the point (if it has been calculated) and a *face index* for the X-Y face defined by the sample points at $(i+l, j+m, k)$, for $l, m \in \{0, 1\}$. The face index is used in calculating the cube index for the two cubes which include this face; bit $l + 2m$ is 1 iff $V(i + l, j + m, k) > T$. Note that each possible intersection is uniquely identified with one sample point.

Central to our implementation are three procedures:

calc_xy uses an X-Y plane of data to calculate the face index for each X-Y face and the intersections along the X and Y edges within the plane. The intersections (i.e. their coordinates and field gradient values) are stored in a list—each intersection is computed only once. The field gradient values are calculated only for those sample points which are adjacent to an intersection; this requires access to the planes of data above and below the plane being considered. A flag is set for these sample points to avoid calculating the gradient more than once.

calc_z uses two adjacent planes of data, together with the information calculated by calc_xy, to calculate the intersections on the Z edges between the two planes. Calculating the gradients for these intersections may require access to the planes of data above and below the two planes being considered. Our implementation thus requires four planes of sample values in memory at once (two if gradient values are not required).

march uses the information derived by calc_xy and calc_z for two adjacent planes, which define one X-Y plane of cubes. For each cube, it calculates the cube index from the face indices of the lower and upper faces, calculates a *disambiguation index* if the cube contains any ambiguous faces, and then generates zero or more polygons. The polygons are selected from a list, based on the cube index and the disambiguation index.

The memory requirements of our implementation are for two complete planes of the derived data described above (which occupies 32 bytes per sample point) and up to four planes of the original volume data. Intersections are added to a list as they are discovered and referred to by either a pointer or an index from then on; the position and gradient for each intersection are only calculated once.

Rather than reducing each of the 256 cube cases to one of the 14 major cases, our implementation handles each of the cases separately (in fact the same code handles a case and its inverse). This eliminates the step of rotating and/or reflecting the polygons. The cube cases are handled by a C switch statement. For the unambiguous cube cases, the corresponding case of the switch statement merely contains code to output zero or more polygons from a list. For the ambiguous cases, the code first calculates a disambiguation index by comparing the coordinates of intersections on opposite sides of the ambiguous faces. This disambiguation index is then used in a switch statement to output one or more polygons from the list. Each polygon is specified in this list as a list of cube edges; the intersections on these cube edges become the vertices of the polygon in the order given.

A notable feature of our implementation is the use of a program to generate the switch statement and polygon list described above. This program analyses each of the 256 cube cases, finding which faces are ambiguous and enumerating all of the possible topologies by forming sets of connected cube vertices and tracing the polygons bounding the sets. This program then generates two files: one containing a list of all of the distinct polygons found, and another containing the body of the switch statement described in the previous paragraph. Although this only needs to be done once, it is a tedious and error-prone task for a human; using a program gives a higher degree of confidence that the case tables and associated polygons are correct. This approach also allows the case table to be fully enumerated for speed: the set of cases amounts to over 1500 lines of automatically generated C code.

4 Parallel implementation

Parallelizing the Marching Cubes algorithm on a MIMD parallel machine is essentially quite straightforward: simply partition the cubes between the cell processors, execute the serial Marching Cubes code on each cell processor, and combine the results. In our implementation, each X-Y plane of data of dimension $N_X \times N_Y$ is considered to consist of $(N_X - 1) \times (N_Y - 1)$ faces. These faces are divided into rectangular blocks which are each assigned to one processor. Each processor processes the same blocks on each X-Y plane, thus the blocks are extended in the Z direction to form 'pillars' with $N_Z - 1$ cubes in the Z direction.

Execution of the program proceeds in three steps:

1. The host processor broadcasts the volume data to the cells, plane by plane. Each cell processor saves in memory those samples from each plane which it requires for processing the blocks which have been assigned to it, converting them to floating-point if necessary.

2. The host processor broadcasts the isovalue, and all cell processors execute the serial Marching Cubes code on each pillar in turn. The cell processors then synchronize.

3. If required, the cell processors convert the triangle coordinates and gradients to an ASCII representation and send them to the host processor for storage.

Since each cube is defined by a $2 \times 2 \times 2$ grid of sample points, the cell processors store for each pillar slightly more sample points than cubes—one more sample point than cubes in each dimension. If gradients are required, the volume of sample points is extended by one more sample point in the positive and negative axis directions (except at the boundaries of the volume). Consequently, the sets of sample points stored for adjacent blocks overlap by up to three rows or columns.

Because of the overlap of sample points, some intersections will be calculated more than once—those which are on an edge between two sample points, both of which are in common between two adjacent pillars. Intersections which are on the common surface between two pillars will be computed twice; intersections on the Z edges where four pillars meet will be computed four times. Gradient values (if required) will also be computed multiple times. It would be possible to

replace these redundant computations with communication, but the communication cost would be likely to be greater than the computational cost for any reasonably simple implementation, because the computation of intersections and gradient values are quite fast operations. Section 6.1 discusses this in more detail.

Since cubes which are not intersected by the volume can be processed much more quickly than those which are, there can be a considerable variation in the time required to process the pillars. This is particularly so if the volume contains a region of interest near the center, with very little of the isosurface occurring near the boundaries of the volume. If each cell processor only handles one pillar, the load imbalance can typically be as much as 50%. Here 'load imbalance' is defined as the proportion of the total execution time which cell processors spend on average waiting for the slowest cell.

One solution to this problem is to divide the volume into several times as many pillars as there are processors, and assign the pillars to the processors in such a fashion that each processor gets some pillars near the edge of the volume and some near the center. Our approach achieves this by assigning the pillars to the processors in an interleaved fashion. The volume is divided along the X and Y axes into $n_X C_X$ and $n_Y C_Y$ pieces respectively, where the cell processor array is of dimension $C_X \times C_Y$, and n_X and n_Y are the degrees of interleaving in the X and Y directions respectively. Cell processor (i, j) then processes pillars $(i + lC_X, j + mC_Y)$, for $0 \leq l < n_X$, $0 \leq m < n_Y$.

A consequence of subdividing the volume more finely is that the amount of redundant computation increases. In the limit, when each pillar has only 2×2 samples in each X-Y plane, all intersections on X and Y edges would be computed twice and all intersections on Z edges would be computed four times (except for those at the boundaries of the volume). Thus a trade-off arises between load balance and the total amount of computation performed: subdividing more finely improves the load balance but increases the total computational load. Thus, as n_X and n_Y are increased, the total time taken to perform the computation first decreases and then increases. The results shown in figure 2 demonstrate this. We expect to obtain the best results when the blocks are approximately square, since a square block maximizes the ratio of area to perimeter (the redundant computations occur on the perimeter). For the 8×16 cell configuration of our AP1000, using $n_X = 2n_Y$ will give approximately square blocks. We would probably obtain even better results if we regarded the processor array as being 3-dimensional, instead of 2-dimensional, and partitioned the data in the Z direction as well as the X and Y directions.

Figure 2 shows some measured results for a 128-processor AP1000 (in an 8×16 cell configuration) on an example volume of $256 \times 256 \times 40$ samples, generating 46266 triangles. The height of each column shows the time taken for the cell processors to perform the isosurface operation, including gradient calculations, (step 2 above, excluding steps 1 and 3) and synchronize. The darker portion of each column shows the average time per processor taken to compute the isosurface (excluding synchronization time). The lighter portion shows the load imbalance, measured as the average time spent waiting for synchronization. The minimum time taken to synchronize was 0.55ms in each case. The results were quite consistent over several runs. We

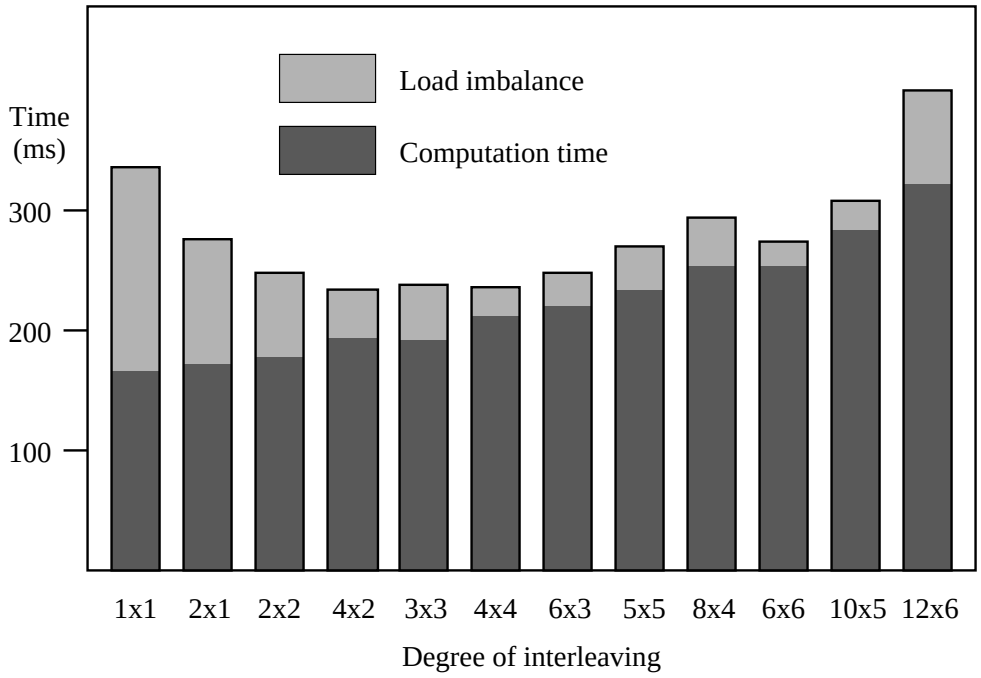

Figure 2: Measured execution times as a function of interleaving

have found empirically that values for n_X and n_Y of around 2 to 4 give the shortest total time for typical volumes.

The fastest case ($n_X \times n_Y = 4 \times 2$) took 0.233 seconds to process the 2.5 million voxels in the data set. This represents a rate of 10.7 million voxels/second. Results obtained with varying isovalues and different data sets indicate that this implementation can generate up to 2.8 million triangles/second when the number of triangles generated is large.

The time taken for a single cell processor to perform the isosurface operation was 30.1 seconds, giving a speedup of 133 for the fastest case. This is greater than the number of processors due, we believe, to cache effects: the complete machine has a total of 16MB of cache memory, enabling it to keep more of the volume data in cache than a single cell processor can.

5 Z-buffer renderer

The isosurface program described in the previous section is capable of generating millions of triangles in a few seconds. Sending these to the AP1000 host processor for rendering or storage can take up to 100 times longer. Clearly it is necessary to implement as much as possible of the further processing of the triangles on the cell processors. To this end, we have implemented a simple parallel Z-buffer triangle renderer on the cell processors. The rendered image still has to be transmitted to the host processor for display, but this is often a much smaller amount of data than the triangles.

The basic Z-buffer rendering algorithm is well-known: see [1] for details. Briefly, the algo-

rithm first colors each triangle vertex according to the lighting model used and transforms its coordinates into screen coordinates according to the viewpoint. Each triangle is then converted into a set of pixels to be written into the frame buffer. The frame buffer is called a Z-buffer because each pixel has associated with it a depth (Z) value as well as a color. A new value is written into a pixel in the Z-buffer only if the new depth value is less than the existing depth value for the pixel. In this way the algorithm allows closer objects to obscure more distant objects. In our implementation, the color for each pixel in a triangle is obtained by Gouraud shading, i.e., it is linearly interpolated from the colors of the triangle vertices.

5.1 Data and work distribution

Two standard work distribution approaches for rendering algorithms are *image-space subdivision*, where each processor renders a small portion of the complete image, and *object-space subdivision*, where each processor has responsibility for rendering a fraction of the objects to be rendered.

The Marching Cubes program described in the previous section generates a list of triangles which is distributed over the cells; thus an object-space subdivision seems natural. With an object-space subdivision approach, each cell renders its geometry data (triangles only in our case) into a full-sized Z-buffer. These Z-buffers are then merged to obtain the final image.

An image-space subdivision approach eliminates the Z-buffer merge phase, and the communication which that involves, at the expense of the communication involved in ensuring that each cell has access to all of the objects which appear in its portion of the Z-buffer. Three methods which could be used to arrange this are:

1. Store all of the object data in the memory of each cell (which limits the scene complexity which can be handled),

2. Perform a visibility analysis and object redistribution every time the viewpoint is changed, or

3. Use a distributed virtual-memory scheme to make objects available to the cell as required.

We have implemented a hybrid image-space/object-space subdivision approach, which dynamically partitions the processors into groups. Each group collectively has a single complete Z-buffer and renders a subset of the objects. That is, we use object-space subdivision between groups and image-space subdivision between the processors within a group. All of the processors in a given group store in memory all of the objects which that group is rendering, and each processor renders a fraction of the Z-buffer.

Merging the Z-buffers is done with a bitonic merge procedure. Assume for the present that each group consists of a single cell processor, which has a complete Z-buffer in memory. The merge procedure proceeds in $L = \lg(N)$ steps for N processors. The cell processors are viewed as being connected in a hypercube in which a cell's neighbor on dimension $n, 0 \leq n < L$ is the cell whose cell ID number differs only in bit n. In step n of the merge procedure, each cell exchanges half of its remaining Z-buffer with its neighbour on dimension n, and merges the

half received with the half not sent. Thus the size of the Z-buffer is halved at each step, until after the last step, each processor has $1/N$ of the final merged Z-buffer. Following this, the image is gathered by the host processor for display and/or storage. Note that step n of this merge procedure partitions the processors into hypercubes of dimension $n+1$ (containing 2^{n+1} processors). Each of these hypercubes stores a single complete Z-buffer, distributed over the processors in the hypercube.

One possible optimization of this procedure is for each cell processor to maintain information (e.g. a bounding rectangle) indicating which parts of the Z-buffer contain anything other than background pixels, and to use this information to reduce the amount of pixel data which has to be sent at each step.

The procedure which partitions the processors into groups involves merging geometry data between processors, following a similar pattern to the Z-buffer merge procedure. The number of processors in a group is always a power of 2; the processors in a group of 2^k processors form a hypercube of dimension k, and the group has a single complete Z-buffer, distributed over the processors in the group. Thus, if a processor is in a group of 2^k processors, it simply omits the first k steps of the Z-buffer merge procedure.

The partitioning procedure proceeds in $\lg(N)$ steps. In step k, $0 \leq k < L$, each cell processor is already in a group of 2^k processors, and has in memory all of the geometry data (triangles) generated by all of the cell processors in the group. Each cell processor communicates with its neighbour on dimension k and decides whether to merge its geometry data with its neighbour or not. The decision to merge is a mutual one: both processors have to be willing to merge for the geometry merging process to continue. In more detail, each step proceeds as follows:

1. Each processor sends a message to its neighbour on dimension k, informing it how much geometry data it has, and then receives the corresponding message from its neighbour.

2. Each processor decides whether it is profitable to merge (this is discussed below), and whether it has sufficient memory for the extra geometry data from its neighbour.

3. Each processor sends a message to its neighbour indicating whether it is willing to merge, and receives the corresponding message from its neighbour.

4. If the processor and its neighbour are both willing to merge, then they both send and receive their geometry data to/from their neighbour.

If a given processor cannot merge in step k, it still participates in the message protocol for steps $k+1 \ldots L-1$, but at each step, refuses to merge.

In fact it is quite possible that some but not all of the processors in a hypercube of dimension k will consider themselves to be in a group of 2^k processors. For example, consider a 4-cell machine, and suppose cells 0 and 1 merge in step 0, as do cells 2 and 3. In step 1, it is possible that cells 0 and 2 might merge, whereas cells 1 and 3 might not. Thus cells 0 and 2 consider themselves to be in a group of 4 processors, whereas cells 1 and 3 consider themselves to be in a group of 2 processors.

This does not, however, give rise to incorrect behaviour. A cell in a group of 2^k processors has the geometry data from all of those cells (even if some others of those cells do not have the geometry data from this cell), and is rendering the fraction of the full image which it would have after k Z-buffer merge steps. Provided that this cell's neighbours on dimensions $k \ldots L-1$ are willing to participate in Z-buffer merging, it will finish up with the correct result. This is assured because each cell refuses to merge geometry data on a higher dimension if it has not merged on some lower dimension.

The geometry merging phase uses a heuristic to evaluate whether it is profitable to merge. This heuristic is based on a threshold value of the ratio (size of geometry data) / (size of Z buffer). If this ratio is larger than the threshold, it is not considered profitable to merge. The 'size of geometry data' term is the total number of bytes of geometry data held by a cell and its neighbour, and the 'size of Z buffer' term is the number of bytes required for the current Z buffer. If the cell and its neighbour do decide to merge their geometry data, then the 'size of geometry data' term increases (approximately doubles) and the 'size of Z buffer' term is halved. Preliminary measurements indicate that threshold values of around 0.1 give the fastest rendering in our implementation.

6 Discussion

6.1 Efficiency issues

The main issues involved in developing an efficient parallel implementation of the Marching Cubes algorithm are listed below. Several of the issues are antagonistic, leading to trade-offs in the design. Most of these issues are encountered more generally in parallel programming.

1. Data distribution—What is the best way to divide the volume data between the processors?

   We want to maximize the total volume size which can be handled by the machine. This requires minimizing the overlap between data stored on different processors. However, some overlap is required by the nature of the Marching Cubes algorithm, and more is required to obtain a reasonable load balance. The amount of redundant storage is largest relative to the volume data for small volumes and high degrees of interleaving.

   The manner in which the data is distributed can affect the amount of computation required, as figure 2 shows. Since we perform redundant computation at the boundaries of each pillar, we expect to obtain the best results when the pillars become nearly cubical, minimizing the surface area to volume ratio. Possibly our implementation should regard the cell processor array as being 3-dimensional, instead of 2-dimensional, and partition the data in the Z direction as well as the X and Y directions.

2. Load balancing—How do we ensure that we do not have some processors sitting idle while others do a disproportionate share of the work?

   We have quantified the load imbalance as the proportion of the total computation time which the cell processors spend on average waiting for the slowest cell. Reducing this load

imbalance requires either (a) making sure that the jobs assigned to the cell processors are of approximately equal difficulty, or (b) arranging that each cell processor can perform several jobs, so that those with the easy jobs do a greater number of jobs. Our Marching Cubes implementation uses interleaving to achieve option (a), at the expense of increasing the total amount of computation required, and slightly decreasing the maximum size of volume which can be handled (the latter has not proved to be a problem). The results shown above in figure 2 show that a modest degree of interleaving is worth while.

3. Communication vs. computation—Where several processors require the same computed value, should it be computed by all, or computed by one and transmitted to the others?

   The resolution of this issue depends on whether it is faster for a given processor to compute the value, or to wait for another processor to compute it and send it. If the first processor has other useful work to do, this depends on how long it takes to compute the value compared with the time taken for a message transfer. If several values need to be sent, they can be combined into one message, thus amortizing the message overhead over them all. If the values are distributed throughout memory, the cost of packing them into contiguous memory in the sender and unpacking them in the receiver must be accounted for.

   In our implementation, intersections on the surface of a pillar are calculated independently by up to four cell processors. Within a given block, the redundant calculations are: (a) the X and Z intersections in the top row of the block (except for blocks at the top of the plane), and (b) the Y and Z intersections in the rightmost column of the block (except for blocks at the right edge of the plane). In principle, the cell processor responsible for a block could exchange messages with adjacent processors to avoid the redundant computation. We do not believe this to be cost-effective, however, for the following reasons:

   (a) Given that the time to calculate an intersection is much less than the time taken to send and receive a message, any cost-effective scheme would have to combine information about many intersections into one message (which would itself take a significant amount of time) and would be likely to be very complex.

   (b) Many blocks will have no intersections in the top row or right-most column. These blocks would incur the message overhead for no useful result, since it takes very little time to determine that an edge has no intersection.

   (c) The data for a column are not in contiguous memory, which complicates sending or receiving a column considerably.

   (d) Because each cell processor does several blocks in an interleaved fashion, the blocks are not processed in linear order across the plane. This means that the data for the last row and column of a block may not be available from the adjacent processors until some time after the cell processor has finished the rest of the block.

   (e) The number of redundant computations per block decreases as the size of the block decreases (even though the total number increases). If two message transfers are

required per block, the relative overhead of using messages will increase faster as the block size decreases than does the overhead of the redundant computation. Other schemes requiring fewer messages are possible, but would increase the complexity of the algorithm substantially, since they would require a drastic rearrangement of the order in which values are obtained.

(f) Eliminating the redundant gradient computations would be quite complex, because the gradient values are only computed when required (i.e. at sample points which are adjacent to an intersection) and one processor would not know which gradient values an adjacent processor requires without either duplicating some of its computation, or receiving a message from the adjacent processor requesting the gradient values it requires. Since computation of the gradient values is a fast operation (two floating-point operations), it is more efficient for each processor to compute the gradient values it requires.

4. Bottlenecks—How do we avoid the massive slowdowns which arise when all of the data have to pass through one processor?

   In our Marching Cubes implementation, bottlenecks arise when I/O is performed via the host. In practice we find that loading the volume data from the host can take around 10 times longer than generating an isosurface, and storing triangles in ASCII form on the host can take 100 times longer. The latter can be avoided by performing all further processing of the isosurface (e.g. rendering or extraction of connected volumes) on the cell processors. Rendering on the cells can introduce another bottleneck if the rendered image has to be sent to the host for display. Both this and the bottleneck in loading the data could be avoided on a machine which has disk storage and a distributed frame buffer connected to the cell processors.

6.2 Interactivity

Our implementation on a 128-processor AP1000 provides a response time of around 4–6 seconds on volumes of around 1–10 million samples, from the time when the isovalue is changed to the time when the complete image is displayed. A significant proportion of this (3 seconds) is due to the delay in getting the image from the cells to a color workstation connected to the host via Ethernet. We hope to reduce the total time to less than 1 second when we have a distributed frame buffer connected to the cell processors.

The current response time is fast enough to enable interactive exploration of the data in a manner which is not possible using a standard serial workstation. Nevertheless, it would be useful to reduce the response time to 0.1 seconds or less. Our success in obtaining a more than linear speedup gives hope that future machines, with more and/or faster processors, will achieve this.

7 Conclusions

The simple parallelization scheme described here has been shown to utilize effectively a 128-processor parallel machine, in terms of both time and space. That is, with a modest degree of interleaving, our implementation achieves a speedup approximately equal to the number of processors, while distributing the volume data between the cell processors with little overlap. We expect our implementation to work well on larger machines, although for a fixed size of volume, the efficiency will drop somewhat as the number of processors increases. In addition, we expect that similar implementations of the Marching Cubes algorithm will work well on other types of parallel computers, such as shared-memory MIMD machines or SIMD machines. Thus, as parallel computers become more readily available, cheaper and more powerful, isosurface generation and display will become a useful tool for the interactive exploration of extremely large volume data sets.

References

[1] J. A. Foley, A. van Dam, S. K. Feiner and J. F. Hughes, "Computer Graphics: Principles and Practice", Second Edition, Addison-Wesley, Reading, Massachusetts, 1990.

[2] H. Ishihata, T. Horie, S. Inano, T. Shimizu and S. Kato, "An Architecture of Highly Parallel Computer AP1000", *Proceedings of IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing* (May 1991), 13–16.

[3] M. Ishii, G. Goto and Y. Hatano, "Cellular Array Processor CAP and it Application to Computer Graphics", *Fujitsu Scientific Technical Journal* **23**, 397–390, 1987.

[4] M. Ishii, H. Sato, K. Murakami, M. Ikesawa and H. Ishihata, "Cellular Array Processor and its Applications", *Journal of VLSI Signal Processing* **1**, 57–67, 1989.

[5] W. E. Lorensen and H. E. Cline, "Marching cubes: a high resolution 3D surface construction algorithm", *ACM Computer Graphics* **21**, 4 (July 1987), 163–169.

[6] J. Wilhelms and A. Van Gelder, "Topological Considerations in Isosurface Generation" (Extended Abstract), *ACM Computer Graphics* **24**, 5 (November 1990), 79–86.

[7] G. Wyvill, C. McPheeters and B. Wyvill, "Data structure for *soft* objects", *The Visual Computer* **2**(4), 227–234, August 1986.