Technical Report ITL-95-2
February 1995

US Army Corps of Engineers
Waterways Experiment Station

An Investigation of the Parallel Implementation of Two Volume Visualization Techniques on the nCUBE 2

by John E. West, Michael M. Stephens, Louis H. Turcotte

Approved For Public Release; Distribution Is Unlimited

19950424 035

DTIC ELECTE APR 25 1995 B

DTIC QUALITY INSPECTED 5

The contents of this report are not to be used for advertising, publication, or promotional purposes. Citation of trade names does not constitute an official endorsement or approval of the use of such commercial products.

PRINTED ON RECYCLED PAPER

Technical Report ITL-95-2
February 1995

An Investigation of the Parallel Implementation of Two Volume Visualization Techniques on the nCUBE 2

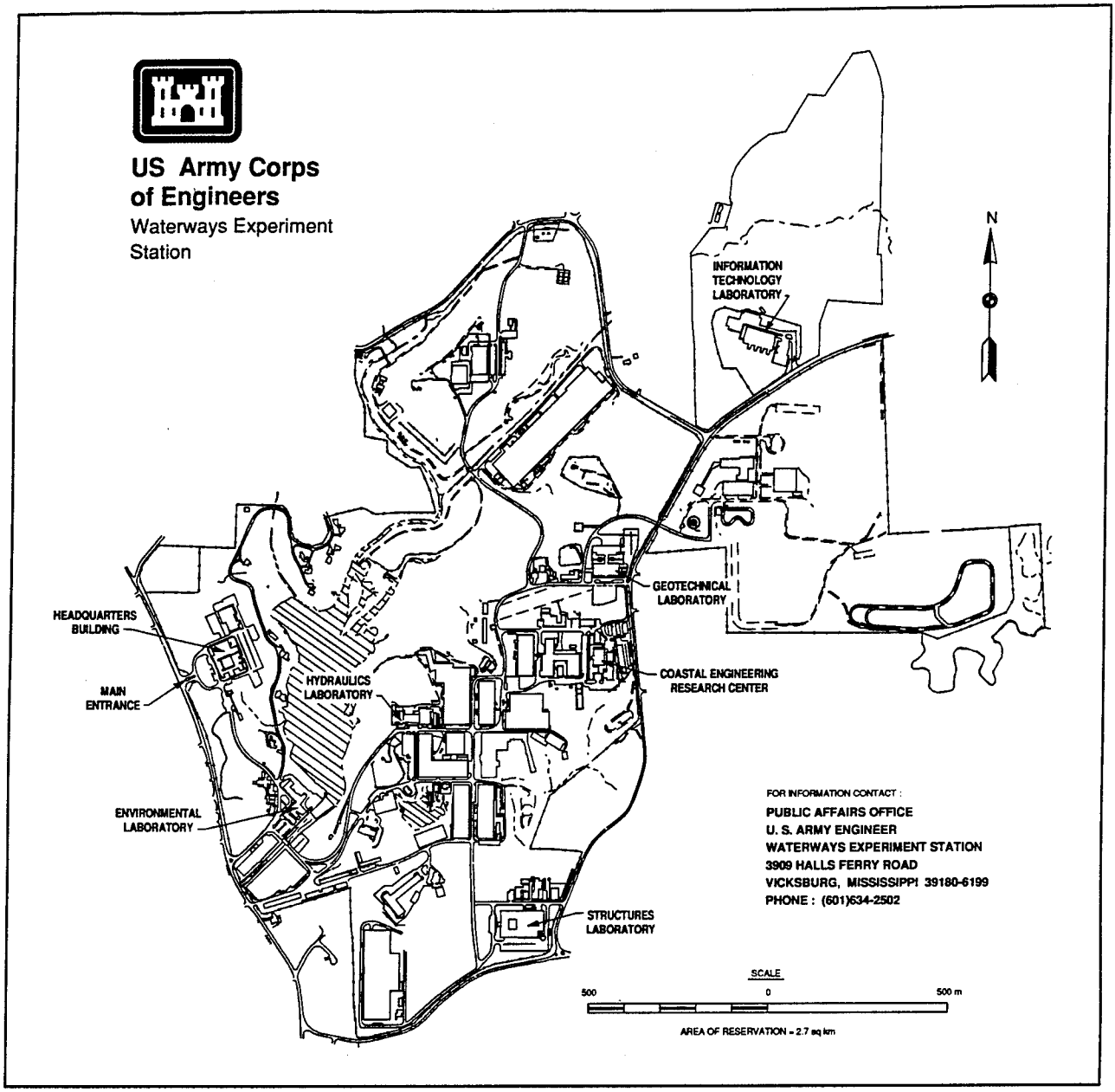
by John E. West, Michael M. Stephens, Louis H. Turcotte

U.S. Army Corps of Engineers
Waterways Experiment Station
3909 Halls Ferry Road
Vicksburg, MS 39180-6199

Final report

Approved for public release; distribution is unlimited

Prepared for U.S. Army Corps of Engineers



Waterways Experiment Station Cataloging-in-Publication Data

West, John E.

An investigation of the parallel implementation of two volume visualization techniques on the nCUBE 2 / by John E. West, Michael M. Stephens, Louis H. Turcotte ; prepared for U.S. Army Corps of Engineers.

61 p. : ill. ; 28 cm. -- (Technical report ; ITL-95-2)

Includes bibliographic references.

1. Visual programming (Computer science) 2. Parallel processing (Computer science) 3. Computer input-output equipment. 4. Electronic digital computers. I. Stephens, Michael M. II. Turcotte, Louis H. III. United States. Army. Corps of Engineers. IV. U.S. Army Engineer Waterways Experiment Station. V. Information Technology Laboratory (US Army Corps of Engineers, Waterways Experiment Station) VI. Title. VII. Series: Technical report (U.S. Army Engineer Waterways Experiment Station) ; ITL-95-2.

TA7 W34 no.ITL-95-2

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

Contents

List of Figures

List of Tables

Preface

This report presents the results of the first stage of an investigation into the adaptation of volume visualization techniques to parallel distributed memory architectures.

This work was performed by Mr. John E. West, Scientific Visualization Center, Information Technology Laboratory (ITL), U.S. Army Engineer Waterways Experiment Station (WES) Vicksburg, MS, Dr. Michael M. Stephens, Army High Performance Computing Research Center, and Dr. Louis Turcotte, Mississippi State University/NSF Engineering Research Center for Computational Field Simulation. The work was under the direction of Dr. N. Radhakrishnan, Director, ITL.

During publication of this report, Dr. Robert W. Whalin was Director of WES. COL Bruce K. Howard, EN, was Commander.

[0]The contents of this report are not to be used for advertising, publications, or promotional purposes. Citation of trade names does not constitute an official endorsement or approval of the use of such commercial products.

Chapter 1

Introduction

As computational power and numerical sophistication have increased over the years, so has the need for techniques which aid researchers in assimilating information from increasingly complex and voluminous sets of data. Advances in computational power, numerical methods, and physical measurement techniques have all increased the size and dimension of datasets which physicists, geologists, medical professionals, and computational engineers have to sift through in interpreting the results of their investigations. For example, Uselton [20] describes the problem of modeling an F-18 using computational fluid dynamics techniques, a model which resulted in four grids with a total of 1.6 million cells, each cell having eight double precision values. This yields a solution data file of over 100 megabytes for a single time slice; a time varying solution would require hundreds of such data files. Clearly, the challenge is not simply to produce more data, but to gain insight from that data.

A wide range of techniques have been developed over the years to enable scientists to visually analyze these *volume datasets.* A volume dataset is a collection of scalar or vector data (or both) in which each data point is associated with a location in $\Re^3$. Predominant techniques currently in use for volume visualizaiton include: cutting planes; surface extraction techniques, such as the marching cubes algorithm [15]; and direct volume rendering techniques such as ray tracing [7, 13, 17, 19] and splatting [21]. The salient features of each of these methods will be discussed below. While both isosurface and cutting plane techniques have the advantages of speed and utilization of previously explored and well-optimized graphics techniques, they reduce the amount of information which is visualized to a small subset of the total information. Direct volume rendering algorithms seek to remedy this by displaying the entire volume. These methods should be thought of as complementary tools, each revealing structures and relationships to be explored with the other techniques [20].

There are several traditional drawbacks to the application of these techniques to large volume datasets. In the case of numerical simulations, the data is typically computed on a large vector or parallel machine which is frequently remote from the user. The data is then downloaded to a single local workstation for visualization. However, the process of moving the data to a workstation may be impractical for large datasets, and generating meaningful visualizations once the dataset is available may be very time consuming. This is especially true for direct volume rendering, where renderings of large datasets can take hours. It has been noted by several researchers that interactive exploration of datasets tremendously increases the speed with which scientists can glean information from their data. But, in light of the size of the datasets involved and the limited memory and computational resources of commonly used workstations, this is usually only possible at the expense of image accuracy and quality, or for certain classes of data.

Examination of many of volume visualization algorithms reveals inherent parallelism which can be exploited on parallel computers; in particular, distributed memory parallel machines. Additionally, this class of parallel machine has advantages over sequential machines for volume visualization in that large datasets can be distributed over the processors and operated on concurrently, thereby reducing the computational time and memory requirement per processor. These parallel machines may be the high-end, single-box, massively parallel machines specifically bought for large scale computation and typified by Thinking Machines' CM-5, or networks of existing workstations which are combined into virtual parallel machines by software such as Linda [22], PVM [2], p4 [4], and others [18].

This report addresses several of the issues surrounding the development of volume visualization tools for distributed memory parallel machines. First, an introduction to several common algorithms used in volume visualization is presented, highlighting the parallel implementation of two of those algorithms - marching cubes and splatting. Issues pertaining to parallel rendering, such as data distribution and processor control are discussed and timing results for a range of test datasets are presented and analyzed for trends using a variety of performance metrics.

Chapter 2

Volume Visualization

This section introduces some of the basic techniques currently used in visualization of volume datasets. As discussed earlier, a volume dataset contains a set of scalar or vector data values each associated with a location in $\Re^3$. Examples of volume datasets include the stacked planes of CT or MR scans found in medical research, or the solution to three-dimensional partial differential equations, *e.g.* pressure and velocity values in the field surrounding an airfoil. Techniques for data visualization in $\Re^2$, such as contour plots or Gouraud-shaded, pseudo-color polygon maps are not sufficient on their own to facilitate rapid exploration and understanding of volume data by researchers, as they don't explicitly reveal internal structure and relationships.

There has been a wealth of algorithm development in volume visualization in recent years. Westover [21] identifies the algorithms produced in this area in several different categories. Blinn [3], Levoy [13], and Sabella [17] describe methods which map screen pixels onto the dataset being rendered by sending rays from the image into the dataset. This is frequently referred to as *backward* or *image space* volume rendering. Frieder [10], Lenz [12], Drebin [7], and Westover [21] describe methods which map the dataset onto screen pixels; techniques commonly referred to as *forward* or *object space* volume rendering. Lorensen [15] and Upson [19] describe various methods of extracting surfaces from volume datasets and displaying these surfaces. Examined in a broader context, these approaches can be categorized into two general classes: volume rendering and surface extraction.

The implementation of these algorithms, regardless of class, is dependent upon the structure, or *grid*, which defines the relationship between values in the dataset. Therefore, before studying the techniques themselves, the standard structures used to organize volumetric datasets are introduced.

2.1 Volume Dataset Grids

Each data sample in a volume dataset is usually associated with a location in $\Re^3$ by a grid. This grid describes how each data point is placed relative to the other data points, and gives its absolute location in space. These grids come in several types; the most common varieties are *structured* and *unstructured* grids. In the text which follows, the notation (i, j, k) denotes an ordered triple specifying a node location in the grid, while the ordered triple (x, y, z) specifies the location in Cartesian space of a node (i, j, k) in that grid.

A *structured grid* is one in which neighbor information is implicit in the grid ordering. For example, the left neighbor of node (i, j, k) is at node $(i - 1, j, k)$ in the grid. Structured grids come in two forms: regular rectilinear and curvilinear. A regular rectilinear structured grid is one in which the individual cells of the dataset are at least parallelepipeds, and typically are cubes.

These grids are advantageous in terms of memory usage in that, given the origin and extents of the grid, as well as the spacing between cells in each direction, the (x, y, z) location of a data sample can be readily calculated from its (i, j, k) location in the grid. This obviates the need to store the location of each point in memory, reducing memory requirements by as much as 75%. A curvilinear structured grid, on the other hand, is one in which each grid cell, or *voxel*, is a general hexahedron. For these grids it is usually necessary to explicitly store the (x, y, z) location of each data point with its value. Regular rectilinear grids are often found in medical datasets, while structured curvilinear grids are typically found in numerical solutions to partial differential equations involving complex geometry.

An *unstructured grid* is one in which the neighbor information must be stored explicitly in a connectivity table for each node in the grid. This increases the memory requirements for storing the grid and associated data values, and also increases the complexity of the algorithms which access and traverse the grid. Three-dimensional unstructured grids can use either tetrahedral or hexahedral elements of varying order (linear, cubic, etc.), and are typically found in the numerical simulation of partial differential equations using the finite element method where the geometry of interest is especially complex.

The data values stored on these grids are either *vector* or *scalar* values, or both. Vector values have a magnitude and direction associated with each grid location; vector data are typically found in the measurement or numerical approximation of flow fields. Scalar data simply have a magnitude associated with each grid sample; scalar datasets include the classic CT and MR medical datasets.

2.2 Cutting Planes

A cutting plane is formed by intersecting a plane, at arbitrary orientation, with the volume being studied. Data values from the vertices of voxels intersected by the cutting plane are used to interpolate the data value on the plane. This point on the plane is then assigned a color from the color map depending upon this data value. Once this is done for every point of interest on the plane (the level of sampling of the dataset onto the plane is usually controllable by the user), the points are connected and drawn as shaded polygons.

This technique is very fast, and can be used to get an understanding of the data in a localized area in the volume quickly. However, it suffers from the disadvantage that it only displays a small subset of the available information at one time. This problem can be alleviated somewhat by the use of multiple cutting planes, but this has limited usefulness in that too many cutting planes can actually obscure information in the dataset being studied. However, this technique can be used effectively to quickly identify regions for further study with other methods, such as isosurface and direct volume rendering techniques.

2.3 Isosurfaces

Isosurfaces highlight information contained in datasets by building surface representations of boundaries between values in the datasets. The marching cubes algorithm [15] creates a polygonal surface by classifying each voxel in the dataset. First, a voxel is formed by logically connecting eight vertices from the dataset; four from each of two adjacent slices as shown in Figure 2.1. After the user selects a value for which an isosurface is to be created, the *isovalue*, the algorithm processes each voxel, classifying its vertices as less than or greater than the isovalue. Voxels which have all vertices in the same state (i.e., either all greater than or all less than isovalue) contribute no polygons to

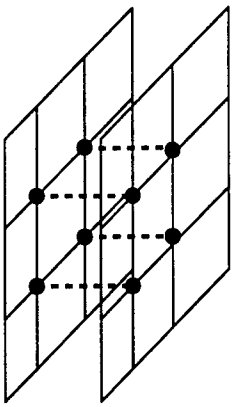


Figure 2.1: Vertices connected to form a voxel.

the isosurface. Voxels which have some vertex values less than the isovalue, and some greater than the isovalue, intersect with the isosurface, and so contribute polygons. Since each voxel has eight vertices, and each vertex is either inside or outside the surface to be created, there are $2^8 = 256$ possible combinations of vertex classifications for voxels on the isosurface.

As each vertex of a voxel is compared with the isovalue, an index between 0 and 255 is generated. A lookup table is then used to generate the triangles necessary to describe the surface/voxel intersection denoted by a particular index. The isosurface value is used in a linear interpolation to determine the location of each triangle's vertices on the edges of the voxel. Once every voxel in the dataset has been processed, a complete isosurface for a given isovalue has been generated, and it is ready for display. An additional step, polygon shading, can be inserted before display if so desired. This step may be desirable if, for example, the isosurface is to be displayed on a low-end graphics machine without hardware support for lighting models. In this shading step, a normal vector for each polygon is approximated by the gradient vector at each vertex of the voxel; the three normals and three triangle vertices are then sent for display. The display unit then scan converts each triangle, and smoothly grades the intensity across the triangle based upon the normals [15]. The chart in Figure 2.2 illustrates the flow of control for the marching cubes algorithm.

The marching cubes algorithm generates a polygonal surface in $\Re^3$ which describes a transition in the dataset between two values; this surface could represent the change from soft tissue to bone in a medical dataset, an isospeed surface in a fluid flow simulation, or a transition between oil and water in a geological survey. In all cases, the resolution of the surface generated by this technique depends upon the resolution of the underlying dataset. As the number of voxels increases in a given area, so will the number of triangles used to represent a transition. Another commonly used isosurface algorithm is *dividing cubes* [6], which generates a point-based surface at the pixel resolution of the display device, eliminating the scan conversion step of the previous algorithm. This method is limited by the resolution of the display rather than the resolution of the dataset, and is useful for datasets which are so dense that marching cubes yields polygons which are close to, or less than, the pixel size.

2.4 Direct Volume Rendering

A major drawback of surface extraction algorithms (both isosurface and cutting planes) is that they display only a subset of the total data in the volume being studied. This problem can be ameliorated

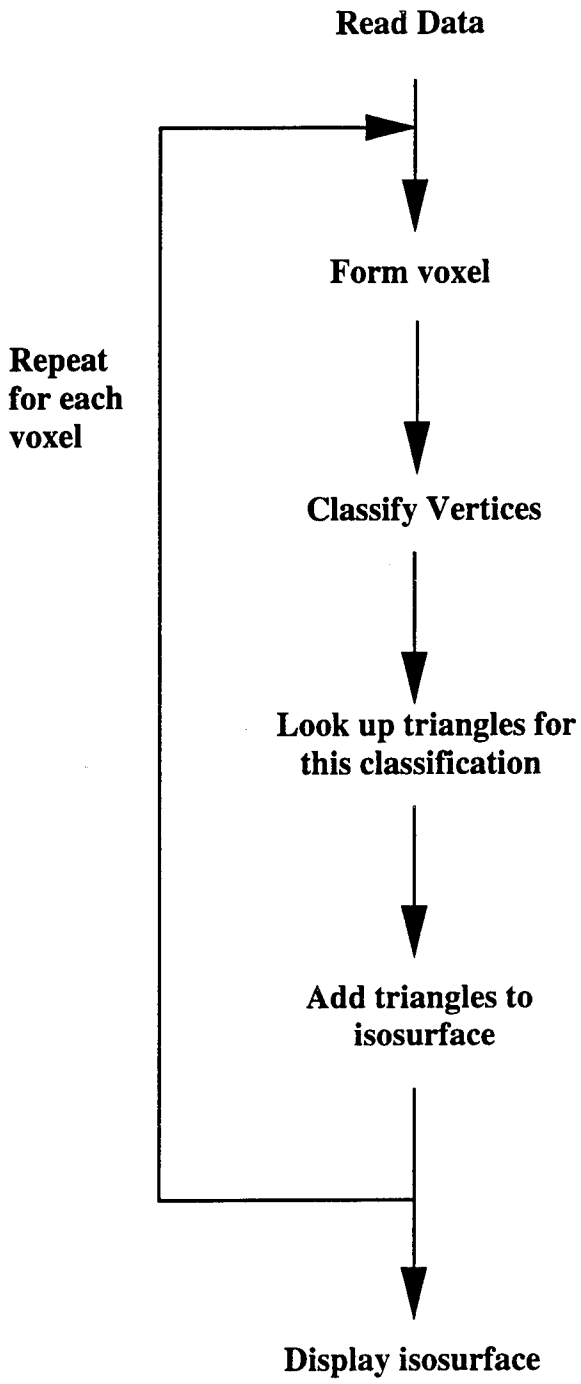


Figure 2.2: Flow diagram for the marching cubes algorithm.

by the use of multiple, semi-transparent surfaces displayed simultaneously and registered with one another, but this solution is effective for only a few such surfaces. Volume rendering, also called *direct* volume rendering, refers to a class of techniques which allow the visualization of the entire data volume rather than a subset of the volume. The image is typically created, or *rendered*, by sampling the data in the volume onto an image plane and then mapping the data values to varying colors and opacities. The two broad categories of technique, image-space and object-space methods, are differentiated by whether they project data onto the image or the image onto the data. The final effect can be like a set of transparent, colored gels which allow internal structure in a dataset to be discerned at the same time as it allows the relationship between features in different locations in the data volume to be studied. The opacity and color maps can usually be edited by the user to highlight aspects of the data under study. Two common direct volume rendering techniques are discussed below: *ray casting* and *splatting*.

2.4.1 Ray Casting

Ray casting is closely related to ray tracing, a technique which is very popular in computer graphics for creating realistic scenes. In ray casting, rays are sent along the path from the eye to a pixel and pass through portions of the dataset, without generating the reflective and refractive rays found in ray tracing. After the ray encounters the volume, the data in the volume is sampled along the length of the ray to determine the contribution of this sample path to the current pixel's opacity and color according to a transfer function. This new color and opacity information is then accumulated into the image pixel, and the next sample is processed. This continues until the ray finally exits the volume, or the pixel becomes opaque.

The process of sampling the data volume along the path of a ray as it passes through that volume and compositing the samples into the final image is essentially integration along the ray. Integration along the ray can be accomplished using the trapezoidal rule as follows

$$\int ray = \sum_{i=0}^{i=n} \frac{Sample_i + Sample_{i+1}}{2}(D_{sample}) \tag{2.1}$$

where $Sample_i$ and $Sample_{i+1}$ are the values at two sample locations along the ray, and D_{sample} is the distance between those samples [11]. This method weights a sample contribution before it is summed into the final pixel value by the distance between the samples. A side effect of this weighting is that contributions from smaller cells can disappear from the final image. This is often undesirable, especially in the numerical solution of partial differential equations, where the small cells are often chosen with great care to achieve more accurate information in areas of special interest or high gradient. A technique which effectively ignores these cells robs the researcher of vital information. This problem may be avoided by choosing D_{sample} to be, for example, 2.0. In this way, all contributions are weighted evenly.

As discussed earlier, there are two versions of the basic ray casting algorithm: image-space and object-space methods. In image-space methods, rays are sent from the viewer's location through each pixel in the image and then into the dataset. This technique has the advantage of producing fewer image artifacts as picture resolution changes for a given dataset density. In object-space ray casting, rays are sent from the data to the viewer location through the image plane. In this case, care must be taken to ensure that rays are sent from the dataset such that all pixels in the image have a contribution. Because each sample in the dataset must be visited, object-space ray casting (and, indeed, object-space volume rendering methods in general) is often slower than its image-space counterpart.

There is another step which may be added into the rendering pipeline that computes shading and visibility information for each pixel in the image so that the final rendered image actually resembles the object being rendered (e.g., a skull) rather than a set of transparent gels. However, it is often the case that this is undesirable in volume visualization of general datasets, as information in the interior of the volume is obscured. Shading is not considered in this implementation.

2.4.2 Splatting

Splatting is an object-space volume rendering technique which reconstructs the sampled data volume on the image plane using a *reconstruction kernel.* The central idea in this technique is the use of convolution: the datum at each sample point in the volume being rendered is spread over multiple pixels in the image plane by the reconstruction kernel. The number of pixels in the final image which the reconstruction kernel covers is called the *footprint.* The reconstruction kernel is essentially a two-dimensional table, or grid, with a precomputed function value, or weight, determined by the *kernel function* at each grid point.

The method proceeds by projecting a sample to the image plane, and then centering the reconstruction kernel table over this location in screen space. Next, the renderer maps this data value into a color and an opacity. Finally, the pixels covered by the kernel are identified, and the data value of the original sample is multiplied by the kernel table entry which is closest in screen space, summing this new value in the image pixel in the process. This process is illustrated in Figure 2.4. Because the kernel function values and kernel footprint remain constant for a given projection, all of the values needed for the convolution can be precomputed, except for a screen space offset, and stored in the kernel table. This dramatically reduces the execution time. As in ray casting, shading functions and complex lighting models can also be applied to provide a more realistic-looking rendering, however for the aforementioned reasons we have chosen not to implement shading at this time. The chart in Figure 2.3 illustrates the flow of control for the splatting algorithm.

There are several points to note about this algorithm. First, it is possible to implement adaptive image reconstruction by varying the footprint size from very small, producing a highly pixellated but quickly rendered image, to the "proper" size, which produces a pleasing and informative final image. The ambiguous wording in the preceding statement is intentional, and points to the subjective choice of footprint size. The footprint size is typically expressed as a function of the inter-voxel spacing in the dataset. A footprint size which produces a good image for one dataset may be too big, producing a blurry image, or too small, producing a pixellated image, for another dataset. Also, the footprint size is dependent upon the number of pixels in the image to be rendered. A general rule-of-thumb which the authors have found is that footprints having a radius of 150 to 170 percent of the inter-voxel spacing work well for most datasets - however, this choice is subjective and varies with the dataset resolution and application.

In addition to the radius of the reconstruction kernel, there are two other factors which will affect the final image. The first is the resolution of the kernel itself; that is, the number of points in the grid used to discretize the footprint function. This choice entails a tradeoff between the memory space needed to store the kernel table and the resultant artifacts in the generated image. A kernel which is too coarse, say 5x5 entries, can result in an approximation to the circular footprint which is actually square. Denser kernels produce successively smoother footprints, at the cost of increased memory. The resolution of a coarse kernel can be improved by interpolating pixel values based upon footprint function values from kernel points surrounding the pixel being processed. However, this increases execution time, when compared to the nearest-neighbor sampling which can be used in higher density tables, and can produce interpolation artifacts. The second factor which will

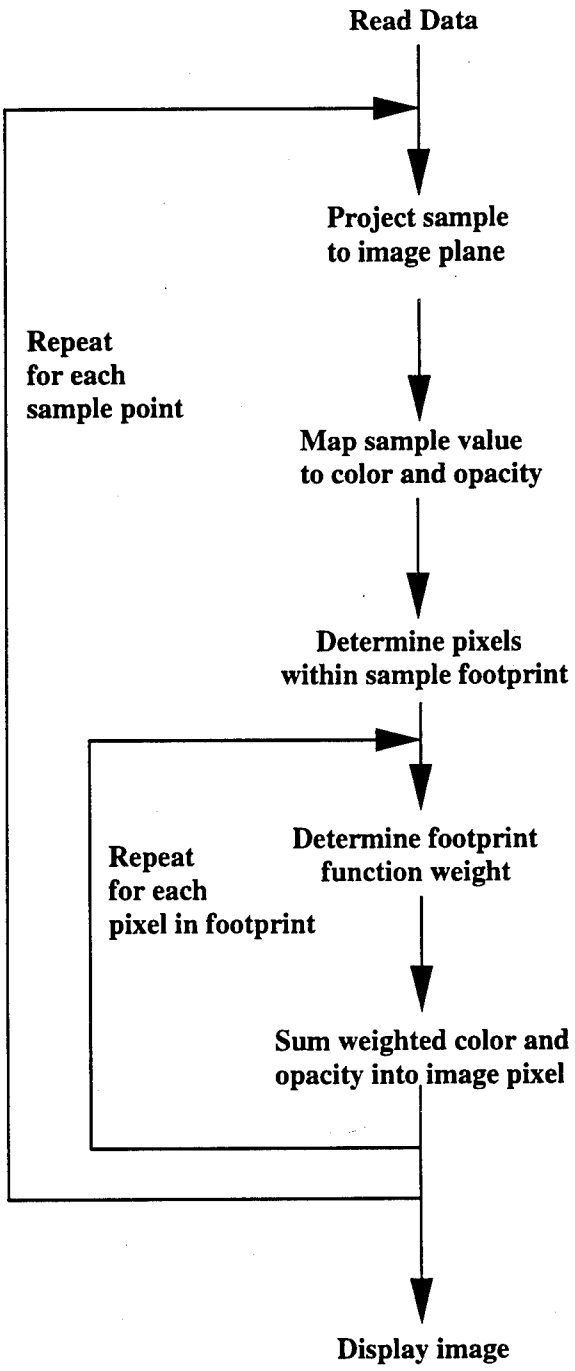


Figure 2.3: Flow diagram for the splatting algorithm.

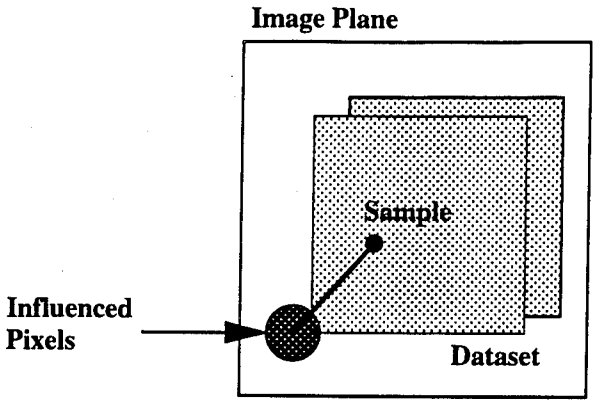


Figure 2.4: Splatting.

affect the final image produced by this technique is the kernel function itself. The kernel function is used to determine the weights of each location in the kernel table. The authors have found that a Gaussian function works well; other functions which produce reasonable images are the cone and bilinear functions [21]. Although some functions are undoubtably more reasonable than others, theoretically almost any function could be used.

Chapter 3

Parallel Processing

Many volume visualization techniques have inherent parallel features which can be exploited on distributed memory parallel machines. In this chapter, a discussion of and motivation for the use of these architectures in volume visualization is first presented, followed by a discussion of such central issues as data distribution and task allocation. Finally, an introduction to metrics commonly used in assessing the performance of parallel programs is given. The discussions which follow are introductory; the reader is referred to Lewis, *et al.* [14] and Fox, *et al.* [9] for a more complete discussion of parallel architectures.

3.1 Parallel Architectures

Parallel machines are classified according to the amount of synchronization required to execute programs, the processor interconnection network, and the memory hierarchy. Two of the most common parallel architectures are MIMD and SIMD. A MIMD machine is a Multiple Instruction Multiple Data architecture, meaning that every processor in the machine can be executing its own code with its own data. A SIMD machine is a Single Instruction Multiple Data architecture in which every processor executes a single instruction in lock step with the other processors in the machine, each operating on its own data. Examples of MIMD machines include networks of workstation *clusters*, the Intel iPSC/860, nCUBE, and Thinking Machines' CM-5; SIMD machines include Thinking Machines' CM-2 and MASPAR systems.

Given a collection of processors in a parallel machine, these processors must be connected by some type of interconnection network. This interconnection network specifies which paths are available for processors to communicate data and control information to one another. Common interconnection network topologies include hypercubes, rings, meshes, fat trees, and toroids. The nCUBE 2 and iPSC/860 are both hypercube machines.

Perhaps the most important characteristic of parallel machines is the organization of memory; this organization is classified as being one of three types: *distributed*, *shared*, or *hybrid*. In distributed memory machines, every processor has its own local memory which cannot be directly accessed by any other processor in the machine. In order for one processor to access data owned by another processor, the second processor must send that data to the processor which needs it by means of a *message*. Thus, distributed memory machines are often called explicit *message-passing* machines. The alternative to this approach is the *shared memory* architecture. In this class of machine, each processor may have a local cache, but there is one central bank of memory to which all processors have access. For this architecture, every processor has access to all of the data

without the need for message passing, but special care must be taken to ensure that updates to memory occur in the proper order by all processors (i.e., a processor shouldn't write a new value to a variable until all processors which need the old value have accessed it). The third type of memory organization, hybrid, is a mix of shared and distributed memory, with each processor sharing a common data space in addition to its own local private memory.

Machines which support a large number of processors are predominantly distributed memory machines. These machines can have upwards of 64K processors (Thinking Machines' CM-5) and are called *highly parallel* or *scalable* machines. Shared memory machines, like Silicon Graphics' Challenge series of machines, typically support a relatively small number of processors (currently, up to 36).

Thus far, the examples for the various machine types which have been given have been single-box parallel machines built specifically for parallel computation. However, there is another growing class of parallel machine: the virtual parallel machine, or cluster. These clusters are typically formed from groups of workstations or vector computers linked together over a network with special software that enables them to function as a distributed memory parallel machine. There are several software environments which will allow individual machines to be grouped together in this way; among them MPI, Linda, PVM, p4, and others (Turcotte provides a description of these software environments and others in [18]). This approach is becoming increasingly popular for several reasons. First, the ability to use existing workstations in a new way is always attractive to research institutions who often find it difficult to justify the purchase of expensive, dedicated, parallel machines. Second, facilities with dedicated parallel machines may have such a high demand that cycles are at a premium. Clusters of workstations offer a way to develop and test parallel codes without stretching these often limited dedicated resources. With the advent of unified APIs like the Message-Passing Interface (MPI) [8], which presents a unified message passing environment on all platforms, code developed on a network of workstations using MPI can be ported directly to a parallel machine with relatively minor changes in the code. This will make parallel computing even more feasible as codes developed in MPI will run on a wide range of platforms, eliminating the need for expensive code ports, and extending code life cycles. Also, the existence of MPI will encourage parallel code developers to move away from *ad hoc* implementations written for a specific machine and a specific application to develop more general, longer lasting and well-engineered codes applying to a wider range of applications and architectures.

3.2 Issues in Parallel Program Design

This section presents an overview of some of the general issues involved in designing a system for performing parallel volume visualization on MIMD architectures. First, the different types of parallelism found in algorithms is discussed, followed by brief discussions of the issues surrounding parallel I/O, data distribution, and program synchronization.

3.2.1 Parallel Algorithms

The foundation of parallel computing is the idea that the work being accomplished by the computer can be performed by a group of processors operating simultaneously on independent portions of the codes and/or data, rather than a single processor doing the work sequentially. However for algorithms to work efficiently in this fashion, they must contain a significant degree-of-parallelism; that is, the tasks, as well as the data used for these tasks, must be distinct and separable. Not

surprisingly there are varying degrees of task/data independence that can be exploited by the "divide-and-conquer" approach to problem solving upon which parallel computers are based. Algorithms which possess a high degree of data independence should be very efficient when executing on parallel machines. The degree of parallelism in an algorithm can be broadly, and somewhat subjectively, categorized as either *coarse grain* or *fine grain* parallelism.

Coarse grain parallelism breaks the task and data into large chunks, and the algorithms which exploit this parallelism fall into one of two types: task-oriented or outer-loop oriented methods. In the first type, the computations performed, and the data used, by any one processor usually differ from the computations and data on other processors. Eventually the results from these independent tasks are brought together for further processing. In the second method, work and data are allocated to individual processors such that all processors are doing similar computations but on their own independent portion of data. In either case, as long as the algorithm allows for complete data independence, coarse grain parallelism scales. However, in almost all cases, *some* data must be shared among processors, resulting in processors becoming dependent upon one another for data. This dependence results in less than ideal scaling, as simply moving the data between processors incurs communication and synchronization overhead during which computational cycles are lost while processors wait for data. Fortunately there are many algorithms which possess a sufficiently high degree of data independence for coarse grain parallelism to be effective. By its very nature coarse grain parallelism maps very well to MIMD computers.

Fine grain parallelism differs from coarse grain parallelism in the amount of data that is independent. In coarse grain parallelism, entire vectors and arrays, or at least large portions of these, are algorithmically independent. In fine grain parallelism, the atomic entities are much smaller portions of the data being manipulated, leading to use of the term "inner loop" parallelism. In general, exploiting fine grain parallelism manually is tedious, and may cost more in time than the value of the benefits derived. For this reason, this level of program parallelism is often left to exploitation by compilers.

3.2.2 Parallel I/O

Parallel machines offer varying degrees of support for I/O, which effects algorithmic design. On some machines, only one special processor can participate in I/O functions. On these machines, reading in a data file requires the designation of that particular node to read the data file and send portions of the data out to the other processors, as shown in Figure 3.1. The alternative situation is one in which each processor has I/O functionality. In this case, multiple processors can read and distribute data simultaneously, or each processor can read its own data independently of the others. The single-reader approach to file I/O is advantageous if the parallel machine does not have a local file system, or if all of the data to be read is in a single file. Having several processors reading the same file, or a set of data files in remote locations, may load the network to the point that it takes significantly longer with several processors than with a single processor reading and sending data out. On the nCUBE used in the present work, each of the nodes (64 in this case) has I/O functionality. Additionally, the nCUBE used in this study has six SCSI disks which are mounted local to its 64 processors, three on each of two separate SCSI controllers. In this situation, it is often advantageous to *stripe* input data files for reading and distribution by several "reader" processes, since the files are on different disks (minimizing controller conflicts) and local to the processors (eliminating network load concerns); note that this file striping differs from traditional striped file systems managed transparently to the user by an operating system. Striping an input file simply means breaking the original file into several smaller portions, each containing part of the original

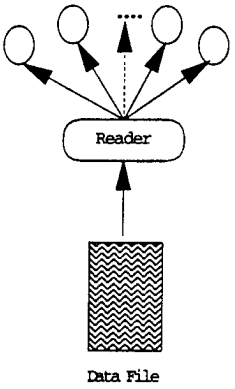


Figure 3.1: Single reader process reading and distributing data.

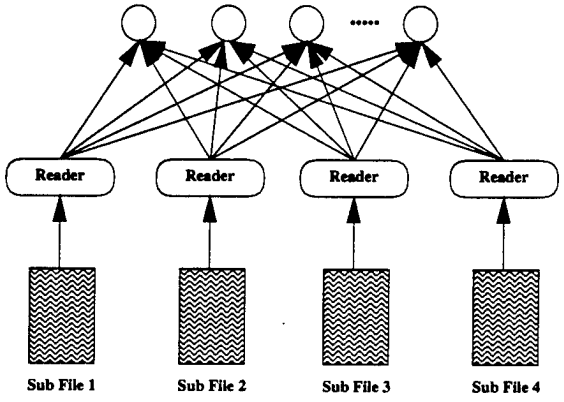


Figure 3.2: Four reader processors reading and distributing data.

data. The data is commonly apportioned in either a round-robin fashion among the new data files, or in a simple linear division of the file into n parts. Because the nCUBE used by the authors has six different local disks, this implementation supports up to six processors which simultaneously read and distribute the data from six striped files. Figure 3.2 illustrates this concept with the original file split into four sub-files and distributed to the processors by each of four readers.

The file is not striped into more than six units for several reasons. First, having up to 64 files (one for each processor in the machine) on only six disks would generate contention as the two SCSI controllers try to manage all the data requests. Second, it would be necessary to have a set of striped files for any of the processor configurations which could possibly access the data; i.e., if the data was striped into sixteen units and one decided to run a job on 32 processors, the original file would have to be re-striped, or several processors would have to access the same file at the same time, once again generating SCSI conflicts. On large data files (upwards of 128^3 data points), the authors have seen a roughly linear speedup in data read time; that is, striping the input file into six units yields a speedup of six, despite the fact that there are only two SCSI controllers.

3.2.3 Data Distribution

Whether the data is input from the keyboard, generated by another program, or read from a file, on a distributed memory machine there remains the problem of how that data is to be mapped

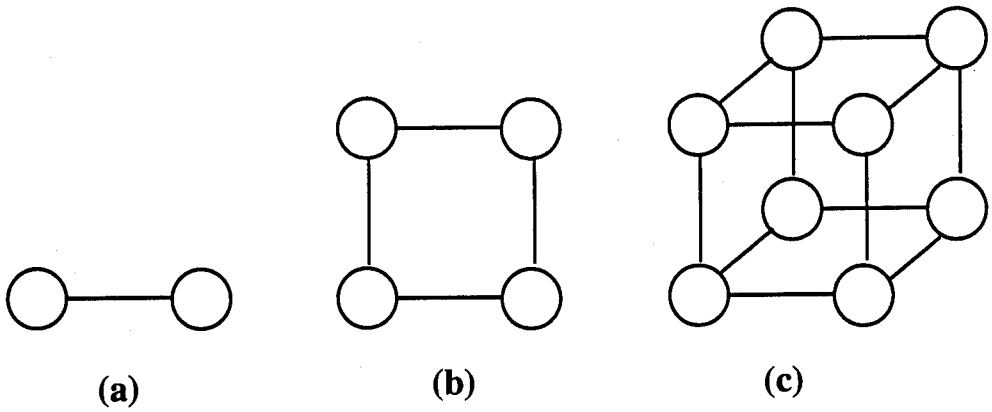


Figure 3.3: Cartesian process topologies from (a) two, (b) four, and (c) eight processors.

onto the processors. The essence of parallel programming lies in dividing a problem into smaller problems which can be worked on independently, each sub-result contributing to the final answer. Based upon this statement, then, it follows that the data needs to be distributed to the processors in such a way that each receives sufficient data to perform their tasks, and such that each processor has the same amount of work to do. This last condition results in what is called a *load balanced* computation. Although this may sound simple, balancing work load is not, in general, a trivial task. In some applications, it is common for conditions to be such that one processor begins to perform many more computations on its data than other processors as the computation progresses; in this case, a computation which begins in a balanced state may become unbalanced, causing inefficiency and wasted cycles as the less burdened processors wait on the others to finish. This situation can often be remedied by supporting a protocol that determines when such imbalances arise and re-allocates tasks or data to remedy the situation. There is a large and on-going body of research into the area of data distribution, or *domain decomposition*, for different classes of algorithms which is far too complex to summarize in this report. We will focus on the data distribution techniques used in the present work; the reader is referred to Fox, *et al.* [9] for a more complete treatment of domain decomposition issues.

In the current system, the data to be visualized is mapped from the three-dimensional volumetric grid to the processors by creating a *virtual topology*. A virtual topology is simply a way to organize processors which more closely suits the task at hand, rather than always being confined by the interconnection topology of the machine. The nCUBE used in the present work has a hypercube connection network, but the data being visualized is organized on a grid in $\Re^3$. Mapping the data from a grid to a hypercube can in theory be done, but it is more convenient to organize the processors themselves into a grid resembling the data and then perform the data distribution. MPI supports creating Cartesian topologies with a simple set of calls, eliminating the need for the user to perform the mapping manually. Figure 3.3 shows the creation of one-, two-, and three-dimensional Cartesian topologies from sets of two, four, and eights processors respectively.

The data is distributed among the processors as subcubes of data which are then mapped onto the three-dimensional Cartesian processor topology (if there are fewer than eight processors, MPI creates a Cartesian topology in the appropriate lower dimension - the data distribution algorithms extend directly to the lower dimensions) using a linear distribution. That is, if there are p processors in the x dimension corresponding to q rows in the dataset, processor 0 gets rows 0 to $q\text{mod}(x)$-1, processor 1 gets rows $q\text{mod}(x)$ to $2q\text{mod}(x)$, and so on. If the number of rows in the dataset, q, is not

evenly divisible by the number of rows in the Cartesian topology, x, the remainder rows can either be distributed evenly among as many processors as needed (it is easy to show that no more than $n-1$ processors with have an extra row, column, or depth) in a round robin fashion, or all remainder rows are simply assigned to one processor. In general, the former is desirable as it minimizes the resultant load imbalance. In the present work both options have been tested, and the authors have not found a significant performance difference between the two. This indicates that, in this case, the resulting load imbalance is not significant. For example, consider a 387^3 dataset distributed onto 64 processors with the processors arranged into a 4x4x4 Cartesian topology. Considering only the first dimension, three of the row processors would each get 96 rows from the datasets, with the final processor receiving 99 rows - a load imbalance of only roughly 3%.

3.2.4 Synchronization

Synchronization in a parallel program is necessary when it needs to be assured that all of the processors in the system have reached a certain point before continuing. For example, if they all read in a subset of data, and the program is to calculate the maximum over all processors' data, it might be necessary to first synchronize after the processors get their data to ensure that the maximum computation includes valid data on all processors. Synchronization generally causes wasted cycles as processors which have advanced in the computation wait for others to catch up. It is usually desirable to perform computations in as asynchronous manner as possible, with each processor proceeding at its own speed.

3.3 Performance Metrics

In quantifying the performance of a parallel application, it is necessary to have metrics by which performance can be gauged. The most common measures used to analyze a parallel application are *wall clock* time, *speedup*, and *efficiency*. Wall clock time, also referred to as execution time, is simply the time it takes for the application to run from start to finish; execution time on n processors is denoted T_n. A program's speedup is the ratio of execution time on a single processor, T_1, to T_n

$$S_n = \frac{T_1}{T_n} \,. \tag{3.1}$$

The efficiency of a program is also an important measure, and is defined as

$$\eta_n = \frac{S_n}{n} \,. \tag{3.2}$$

Efficiency measures how well a task is utilizing the processors available for a given speedup. The ideal speedup is n on n processors, although it is common to have less than ideal speedup and possible, although uncommon, to have a speedup greater than n. If the speedup remains close to n as more processors are added to a problem, the algorithm is *scalable*, and has *linear* speedup. An algorithm which has a speedup greater than n is *super-linear*. Super-linear speedup is most commonly caused by hardware considerations such as an increased number of cache hits for certain processor configurations.

Chapter 4

Parallel Volume Visualization

In this section, the need for parallel volume visualization techniques is discussed, and then the parallel implementation of the two algorithms studied for the present work, marching cubes and splatting, is presented.

4.1 Motivation

Much effort has been directed in the years since volume visualization techniques have been developed in enhancing the speed of the algorithms. This interest in increased speed is motivated both by the rapidly increasing size of datasets which scientists wish to study and by the need for interactivity in data exploration, as discussed above. The bulk of this work, however, has been conducted for uniprocessor machines or specialized rendering architectures. With the advent and growing popularity of parallel machines and systems for connecting existing workstations into virtual parallel machines, a growing body of research is being conducted into processing datasets for visualization on this class of machine.

Distributed memory parallel computers offer several features which can be leveraged in improving the speed of volume visualization techniques. First, the original dataset is spread among the nodes of the parallel machine, reducing the memory demands which even small volume datasets place on single processor systems. For example, a parallel machine having 128 nodes with eight megabytes of memory per node can store up to 1024 megabytes of data. If the grid is regular and rectilinear so that the (x, y, z) locations are not stored, this translates to a volume with roughly 640x640x640 single precision (4 byte) floating point samples. In many cases this will allow the visualization of datasets which historically have been too large to manipulate easily (if at all) on single processor systems. Second, some of the algorithms can be structured so that each processor creates a sub-image of its local data (whether its a subset of triangles in an isosurface, or a volume rendering of a portion of the final image) independently of the other processes, with little interprocessor communication. The user does not achieve a speedup of n on n processors by the addition of the composition step in which the sub-visualizations must be assembled from each processor for display and by communication overhead. Nonetheless, significant improvements in execution time may still be achieved.

In this regard, object-space methods for volume rendering are extremely important. Although the sequential implementation of an object-space rendering method is almost always slower than an image-space method, the object-space methods are often inherently parallel, and significant speedup is possible with relatively simple parallel implementations [21]. While it is possible to successfully

parallelize image-space methods [16], the algorithms for accomplishing this are generally more complex and difficult to develop than their object-space counterparts.

Third, although the images created by volume rendering techniques are often very effective, especially when used in conjunction with other techniques, the real power of the technique becomes evident when the volume is set in motion so that users can rapidly understand how the features in the dataset relate to one another. However, the user rarely achieves interaction with the dataset because a new image must be created for each new viewpoint as the user moves the dataset, and each image is very time consuming to generate. There have been numerous attempts at optimizing these techniques to improve performance for certain types of datasets or architectures, but these techniques usually involve sacrificing image quality and accuracy for speed, a tradeoff which is not particularly desirable in scientific visualization. Successful parallel implementations of these techniques can lead to much improved display rates.

4.2 Parallel Marching Cubes

Marching cubes is a classic example of an algorithm with a large amount of coarse-grained parallelism. As discussed previously, the algorithm steps through a volume dataset on a voxel-by-voxel basis, classifying each voxel with an eight bit index. The value of each voxel's index is determined by subsequently classifying each of the eight vertices of the voxel under examination into one of two different states: inside of the isosurface or outside of the isosurface. Voxels having indices indicating that the surface passes through them are further processed, generating a set of triangles from a lookup table based upon the voxel index which describe how the surface intersects this voxel. These triangles are then added to the array of triangles which form this surface, and the next voxel is processed.

This algorithm is inherently parallel in that voxels may be processed independently of one another. The data is first read in using the multiple-reader model previously discussed, and then distributed by these readers (up to six for this disk configuration) onto the three-dimensional Cartesian topology of processors in contiguous sub-cubes. When the domain decomposition is computed, the number of rows, columns, and depths of the original dataset for which each processor is directly responsible is computed. It should be noted that the marching cubes algorithm will require one extra set of data beyond the boundary of a sub-cube in order to ensure that the final surface is free of gaps when it is assembled for display.

At this point the processors synchronize, awaiting determination of the isosurface for which a surface is to be generated. Once this value is obtained (it is sent from the distributed user-interface running on the SGI host machine - discussed later when the interface is described) each processor generates the portion of the isosurface which intersects its subset of the data volume. After generation is complete, all of the processors send their portions of the isosurface to the host for display, and then await the next isovalue. During the isosurface generation phase, there is no need for processors to communicate with one another, which greatly enhances the speed and efficiency of this implementation. However, the isosurface composition phase, during which all processors send the triangles which they have generated to the host, is a significant communications bottleneck. The process is entirely sequential; therefore, while the amount of data which each processor must process in creating an isosurface for a given dataset decreases as the number of processors used increases, the total amount of time spent in communication with the host by all processors may actually increase. Future work will include the investigation of ways to increase the efficiency of this stage of isosurface generation.

4.3 Parallel Splatting

The splatting algorithm parallelizes in much the same way as the marching cubes algorithm. In the sequential algorithm, each sample point in the dataset is first projected onto the image, and then the sample value is multiplied by the weights in the footprint function and summed into all of the pixels within that particular sample's footprint. As before, multiple readers read and distribute the striped data file to all of the processors participating in the computation. They then synchronize, awaiting the location of the viewpoint before beginning the splatting process. Once the location of the viewer and the image plane are known, each processor can splat every sample contained in its sub-cube of the original data volume (in this case there are no data boundary values to maintain) into its local copy of the final image independently of all other processors. Once every processor has finished splatting its samples, all of the sub-images are composited on a single processor, and the final image is processed and sent to the host for display. The image composition phase does require a great deal of communication as each process sends its sub-image to the compositing process; this is discussed in detail below.

As with the parallel marching cubes algorithm, there is no interprocessor communication during the splatting phase, even when the viewer's location changes. This is in contrast to most parallel ray casting implementations which typically must receive and pass rays along to neighboring processors as they enter and leave their sub-cube of data. The absence of interprocessor communication can only be guaranteed for ray casting methods if the data volume is view-aligned in memory, and each processor knows ahead of time what data it needs to produce an image. This is generally not possible for arbitrary viewer locations and interactive data exploration without the application of expensive and complex data orientation and indexing schemes, so splatting has some advantages over ray casting in this context.

4.3.1 Image Composition

The image composition phase of the above process is particularly worrisome. The nCUBE used for this work has a communication buffer whose size is determined at compile time (this is true if the parallel program is launched by the host machine, as it is in this case), and memory set aside for use by the communication buffer decreases the total memory available on each node for the program and the data needed by that program. If a node has more messages pending than space in the communication buffer, the message is simply dropped, and the program fails. During image composition, each processor must send the local image which it has generated to a single processor for composition. For example, consider the relatively small image size of 200x200 pixels and a communication buffer of 500,000 bytes. Each pixel in the image has 5 floating point values associated with it, for a total of 800,000 bytes per image (one image on each processor). If even one processor attempts to send its entire image to the root processor for composition the communication buffer will overfill and the program will fail.

Several communication schemes have been explored in an attempt to resolve this issue. The simplest scheme is a simple sequential method where the compositing processor (the node where the final image is generated and sent to the host for display) receives each processor's local image one at a time. Each processor sends only the portion of the total image contributed by its data to the root processor for composition, using a handshaking protocol for safe communication. First, rather than sending the entire image array, some of which may be unaffected by the portion of data allocated to any given processor, each processor determines which portion of the total image is effected by its local data, and forms a bounding box around these pixels. It is these pixels

which are finally sent to the root processor for composition into the final image. Depending upon the number of processors used in the generation of the visualization and on the viewpoint, each processor could send only 1/16 of the total number of pixels in the image. However, this is not sufficient to guarantee that the communication buffer will not overfill for large images and arbitrary viewpoints. To ensure that a processor will not attempt to send messages larger than the size of the communication buffer, each processor sends its local sub-image to the compositing process in appropriately-sized chunks which will not exceed the communication buffer size.

The downside of the sequential scheme is just that: *i.e.*, the process is entirely sequential. This creates a potential for wasted bandwidth, if the number of pixels to be sent is small enough that only a portion of the communication buffer will be used. However, it is difficult to avoid this problem in general (by, for example, sending the sub-images of more processors where sub-image size and communication buffer size permit), because as the image size, number of processors, or viewpoint changes the amount of unused bandwidth can vary dramatically.

The other communication schemes for the image composition phase which have been explored thus far order the processors as binary trees and pass information from the leaves to the compositing process. These types of schemes are often used in reduction operations where, for example, a value or values on each processor is to be summed together to achieve a final result known to one or more processors. This approach works well in those situations because, as in the summation example, two processors combine their portions of the sum and pass along only one message. In image composition, two sub-images sent up the tree can only be combined if they cover the same area of the final image; if they do not, combination is not possible and both sub-images are passed along to the next processor. In the general case, all sub-images will cover unique areas of the final image, and no reduction of messages in the tree will be possible. This general case can actually result in more communication and synchronization costs than with the simpler sequential design. In practice, the best performance we have achieved thus far using trees for composition is equivalent to the performance of the sequential scheme; some of the tree schemes have much worse than sequential performance. In light of this finding, the results reported in this paper are based on the sequential composition scheme; research into improving the performance of the image composition is still needed.

4.4 User Interface

The user interface is the portion of this system which ties together the visualization and the computation. It is X Windows/Motif based and uses mixed-mode programming (X for the interface and SGI's GL for displaying scenes) for graphical display of the visualizations produced on the nCUBE. This system is structured on a host/node model, which uses the nCUBE's front-end, an SGI Personal Iris, as the host or controlling process, and the nCUBE for computation of the scene to be displayed. This model was chosen because, in general, parallel architectures are not very efficient for performing the type of immediate user interaction needed for view orientation, opacity and color map editing, and manipulation of rendered data [5].

The interface runs on the nCUBE's front end SGI, which has no graphics capability; therefore DGL is used to display the final visualization over the network on another local SGI workstation. DGL is Silicon Graphics' distributed graphics library which enables a scene computed on one SGI to be displayed on a remote machine across the network. The graphical user interface (GUI) is responsible for allocating and launching jobs on the nCUBE, passing file information and visualization parameters when necessary to a root node in the processor array, and displaying the scene

information sent to it by the nCUBE's processors. All of the compute intensive visualization tasks, such as generating a volume rendering or the constituent triangles of an isosurface, are done on the parallel machine itself, which sends the scene to the GUI after it has completed computation.

Chapter 5

Results

In this section, timing results for the parallel implementations of both the splatting and marching cubes algorithms are presented, and the results are examined in the context of both speedup and efficiency metrics[1]. Timings for one processor of a two processor Silicon Graphics Onyx are also given for comparison.

5.1 Hardware Configuration

The Waterways Experiment Station (WES) nCUBE has 64 processors. Each processor has a clock speed of 25 MHz with eight megabytes of local physical memory and no virtual memory. This eight megabytes is divided between the program and all data the program will use, and the space needed to store the maximum number of messages expected to be pending at one time for each processor (the communication buffer). There is no facility for sharing memory between processors; information is transferred from node-to-node by messages sent on the nCUBE's hypercube interconnection network which has a bandwidth of 2.75 MBytes/s between neighbors. The nCUBE is connected to six local SCSI disks of 1 GBytes each which are divided among two SCSI controllers.

The nCUBE is an attached processor array connected to a Silicon Graphics Personal Iris/35S via a VME bus. The SGI has a 36 MHz R3000 processor and 16 MBytes of memory.

5.2 Test Datasets

In evaluating the performance of the algorithms on the nCUBE, test cases for several datasets over three different image sizes were conducted. Four datasets of varying sizes were used, and images of 100x100, 200x200, and 300x300 pixels were rendered on hypercube dimensions ranging from 2^0 to 2^6. Table 5.1 summarizes the sizes and contents of the datasets used. The test dataset is meaningless data used only for test purposes. The explosion datasets are derived from soil accelerometer data measured during an underground explosion. The larger explosion dataset is an interpolated version of the smaller dataset. The MR brain dataset is a magnetic resonance image of a human brain from the University of North Carolina at Chapel Hill.

[1] Appendix A of this report contains sample volume renderings from various datasets.

Dataset	Dimensions	Data
Test	10x10x10	Test dataset
Small Explosion	33x33x25	Explosion soil velocities
Big Explosion	66x66x50	Explosion soil velocities
MR Brain	128x128x109	MR scan of a human brain

Table 5.1: Datasets and dimensions used for evaluation.

n	Readers	Time
1	1	6.99
2	2	3.51
4	4	1.78
8	6	1.13
16	6	1.10
32	6	1.07
64	6	1.07

Table 5.2: I/O times (in seconds) for the 66x66x50 dataset.

5.3 I/O Performance

Table 5.2 shows the time required to read and distribute the 66x66x50 explosion data file to varying processor configurations. In this table, n is the number of processors used, *Readers* is the number of processors reading the data and also the number of files into which the original data file was striped, and *Time* is the total time for all processors to read, distribute, and receive the data. The data file is 0.9 MBytes; it is regular and rectilinear, so that the (x, y, z) locations are not stored, reducing the total size of the original file by 3/4. For each processor configuration shown in the table, the maximum number of reader processes was used. Note that this number is limited to six by the number of independent SCSI disks local to the nCUBE.

As expected, the read times decrease as the number of processors participating in the simultaneous file read increases up to six. After this point, the times still decrease slightly; this is thought to be due to decreased communication overhead as the reader processors send fewer data to each individual processor during distribution.

5.4 Parallel Marching Cubes Performance

Tables 5.3 and 5.4 summarize the run times measured for the marching cubes implementation on two representative datasets: the 10x10x10 test dataset and the 66x66x50 large explosion dataset. There are three times given for each of the processor configurations tested. The first is the time required for each processor to compute its portion of the isosurface on local data, or *local* time. The second is the time required for all processors to send this local isosurface to the host for compositing and display, the *host comm.* time; the third time is the sum of these times. Note that all times are

in seconds. The isovalue used for the test dataset is 0.87 and generates 364 triangles in the final surface; the isovalue used for the big explosion dataset is 49.99 and generates 14914 triangles in the final surface. These timings do not include the time to read and distribute the dataset, or the time to send the image over the network and display it on a local workstation using DGL.

The graphs in Figures 5.1 and 5.2 show the speedup and efficiency for the marching cubes algorithm operating locally on the subset of data residing on each processor. The performance figures for the test dataset are significantly lower than those for the explosion dataset primarily due to the relative amounts of work each dataset demands from the processors. For the small dataset, there is less data to process (and therefore less work to do), and so synchronization and other overheads are a larger percentage of the run time. For the larger dataset, there is significantly more data per node to be processed, so actual computations form a larger percentage of the run time, yielding higher processor efficiencies. The inefficiencies indicated in these graphs are also caused by the fact that, as more processors are added to the problem, the amount of triangles which must be generated on each processor also changes. Different hypercube dimensions result in different data distributions, which then potentially result in load imbalances as processors which contain a relatively small portion of the isosurface wait for processors with larger portions of the isosurface to complete generation. Note that there is also a potential for this type of load imbalance as the isovalue varies for a given dataset and processor configuration.

While this is indeed a problem, it is one largely inherent in the algorithm and the degree of severity will vary with individual datasets and processor configurations. The sequential communication in the surface composition, however, causes a huge performance penalty, illustrated in Figures 5.3 and 5.4. These figures depict the speedup and efficiency curves for the entire isosurface generation process, including both the time to generate the local isosurfaces and the time needed for each processor to send that isosurface to the host. As the number of processors increases, this communication becomes a severe bottleneck. This problem is difficult to overcome. The size of the nCUBE's communication buffer is static, and so a scheme using communication trees, or similar methods, to composite sub-isosurfaces in parallel would have to be able to monitor the amount of information already in the communication buffer in order to prevent it from overfilling; this is further complicated in that the scheme would have to be completely general due to the variable number of triangles in any given isosurface. nCUBE support for for determining the amount of information in the communication buffer has not yet been found. Therefore, if a communication tree is to be used, a producer-consumer protocol will have to be created to control the amount of information sent as messages at one time by any processor. Great care will have to be taken in the design of such a protocol to ensure that the amount of communication and synchronization overhead incurred by this protocol does not cause a new bottleneck.

5.5 Parallel Splatting Performance

Tables 5.5 to 5.16 summarize the measured times for the test runs performed. Each table represents the timing results for a single dataset and image size over the full range of processor configurations; the times given are in seconds. For each hypercube dimension, there are three times given: a splatting time, an image composition time, and a total time. The splatting time includes only the time needed for each processor to splat its portion of the dataset and create its local image. The image composition time is the total time required for all processors to send their sub-images to the root processor. The total time is the sum of splatting and composition times, in addition to the time required for the root processor to composite and process the image and send the pixels to the

	Time (s)		
n	Local	Host Comm.	Total
1	0.06	0.055	0.11
2	0.03	0.048	0.08
4	0.02	0.054	0.075
8	0.01	0.064	0.07
16	0.005	0.389	0.39
32	0.003	1.83	1.83
64	0.002	4.05	4.05

Table 5.3: Isosurface generation times (in seconds) for the 10x10x10 dataset.

	Time (s)		
n	Local	Host Comm.	Total
1	10.34	1.14	11.51
2	5.26	0.61	5.87
4	2.67	0.33	3.00
8	1.31	0.31	1.63
16	0.67	0.91	1.59
32	0.36	1.88	2.24
64	0.20	3.99	4.20

Table 5.4: Isosurface generation times (in seconds) for the 66x66x50 dataset.

	Time (s)		
n	Splat	Composition	Total
1	15.64	0.00	17.32
2	8.17	0.42	10.28
4	4.49	0.67	6.85
8	2.59	1.16	5.44
16	1.84	2.32	5.84
32	1.39	4.25	7.32
64	1.16	9.14	11.98

Table 5.5: 10x10x10 dataset, 100x100 pixels.

	Time (s)		
n	Splat	Composition	Total
1	59.46	0.00	65.78
2	30.18	1.72	38.21
4	15.72	2.76	24.78
8	8.32	4.70	19.31
16	5.36	8.52	20.18
32	3.59	14.69	24.58
64	2.70	26.22	35.58

Table 5.6: 10x10x10 dataset, 200x200 pixels.

host for display. These timings do not include the time to read and distribute the dataset, or the time to send the image over the network and display it on a local workstation using DGL.

	Time (s)		
n	Splat	Composition	Total
1	132.74	0.00	146.91
2	67.00	N/A	N/A
4	34.52	N/A	N/A
8	17.89	10.42	42.41
16	11.25	18.91	44.23
32	7.26	32.50	53.87
64	5.27	59.49	78.90

Table 5.7: 10x10x10 dataset, 300x300 pixels.

	Time (s)		
n	Splat	Composition	Total
1	42.20	0.00	43.86
2	22.09	0.20	23.94
4	11.73	0.42	13.81
8	6.43	0.57	8.65
16	3.73	1.00	6.38
32	2.31	2.96	6.92
64	1.59	5.57	8.82

Table 5.8: 33x33x25 dataset, 100x100 pixels.

	Time (s)		
n	Splat	Composition	Total
1	157.25	0.00	163.43
2	81.45	0.77	88.42
4	42.46	1.65	50.28
8	22.48	2.20	30.87
16	12.34	3.82	22.35
32	6.96	6.20	19.34
64	4.26	12.96	23.40

Table 5.9: 33x33x25 dataset, 200x200 pixels.

	Time (s)		
n	Splat	Composition	Total
1	342.35	0.00	356.15
2	176.98	2.40	186.73
4	91.58	3.72	109.14
8	48.27	4.95	66.99
16	26.16	8.58	48.53
32	14.44	13.28	41.62
64	8.59	24.15	46.52

Table 5.10: 33x33x25 dataset, 300x300 pixels.

	Time (s)		
n	Splat	Composition	Total
1	109.42	0.00	111.08
2	55.06	0.26	56.98
4	27.89	0.41	29.96
8	14.28	0.59	16.52
16	7.69	1.17	10.52
32	4.29	2.67	8.62
64	2.60	4.86	9.12

Table 5.11: 66x66x50 dataset, 100x100 pixels.

	Time (s)		
n	Splat	Composition	Total
1	345.35	0.00	351.57
2	173.25	0.83	180.29
4	87.12	1.67	95.02
8	44.05	1.97	52.24
16	23.16	3.33	32.70
32	12.37	5.20	23.79
64	6.99	10.57	23.78

Table 5.12: 66x66x50 dataset, 200x200 pixels.

	Time (s)		
n	Splat	Composition	Total
1	723.21	0.00	737.18
2	362.24	2.40	378.45
4	181.73	3.88	199.48
8	91.32	5.06	110.23
16	47.66	7.44	72.02
32	25.16	10.99	49.98
64	13.93	19.28	47.85

Table 5.13: 66x66x50 dataset, 300x300 pixels.

	Time (s)		
n	Splat	Composition	Total
1	N/A	N/A	N/A
2	N/A	N/A	N/A
4	98.91	0.77	101.34
8	50.29	0.62	52.57
16	25.50	1.16	28.33
32	13.09	2.74	17.48
64	6.89	5.68	14.24

Table 5.14: 128x128x109 dataset, 100x100 pixels.

	Time (s)		
n	Splat	Composition	Total
1	N/A	N/A	N/A
2	N/A	N/A	N/A
4	236.34	2.12	244.68
8	119.99	1.84	128.06
16	60.49	2.76	69.49
32	30.64	5.12	41.98
64	15.79	9.69	31.70

Table 5.15: 128x128x109 dataset, 200x200 pixels.

	Time (s)		
n	Splat	Composition	Total
1	N/A	N/A	N/A
2	N/A	N/A	N/A
4	N/A	N/A	N/A
8	228.20	3.72	245.75
16	114.66	6.12	134.61
32	57.91	9.16	80.90
64	29.56	17.07	60.52

Table 5.16: 128x128x109 dataset, 300x300 pixels.

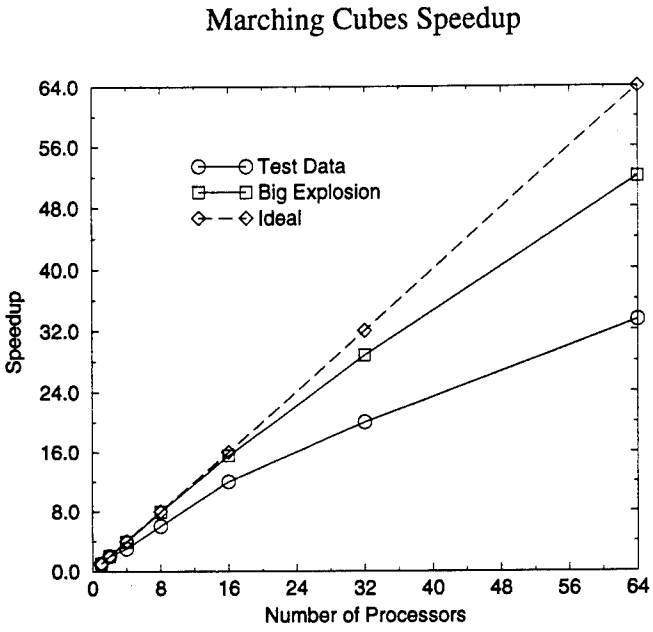


Figure 5.1: Local isosurface generation speedup.

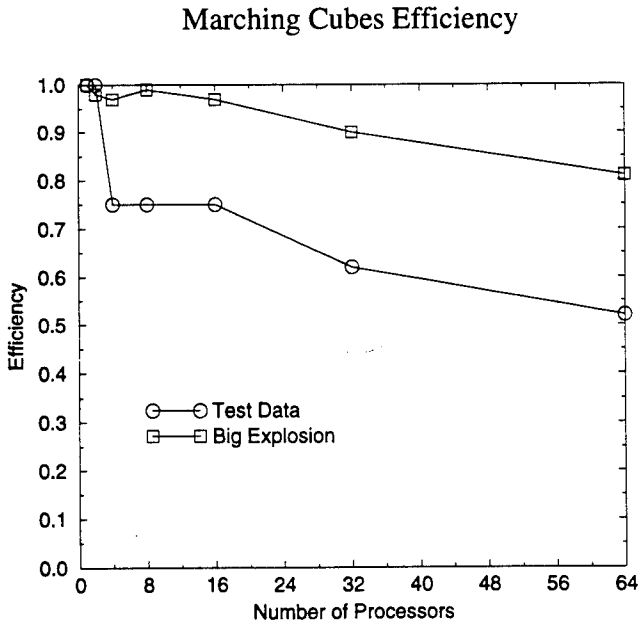


Figure 5.2: Local isosurface generation efficiency.

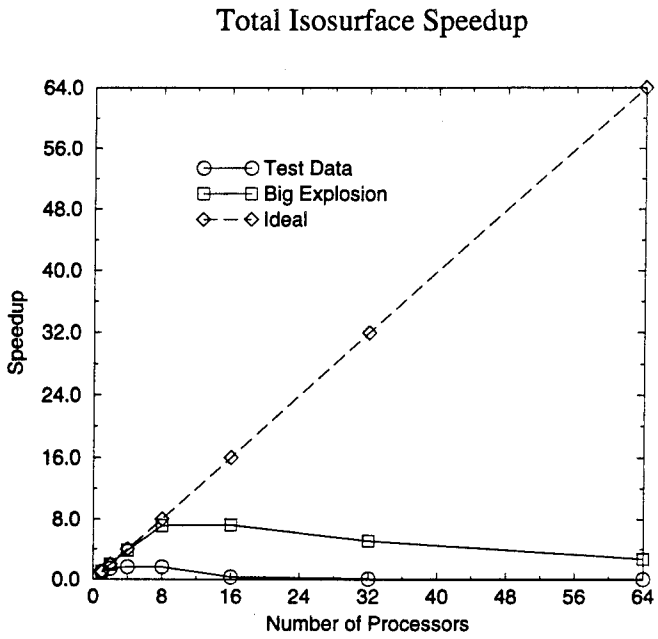


Figure 5.3: Speedup for the entire isosurface generation process.

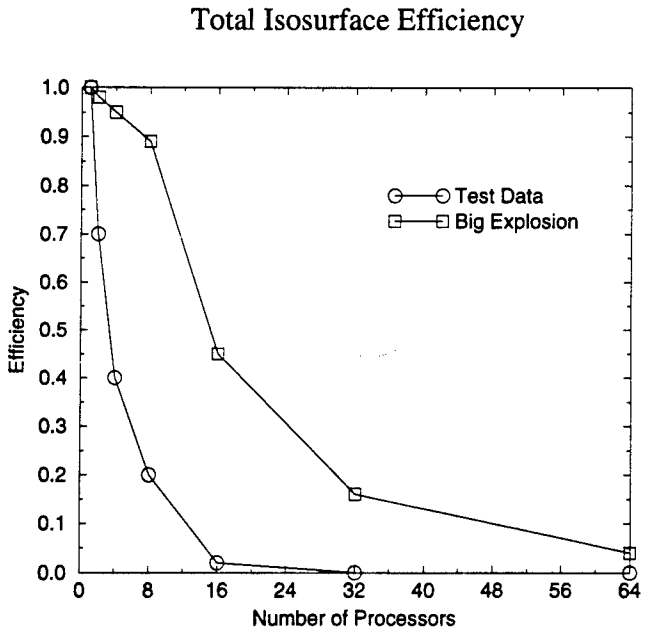


Figure 5.4: Efficiency for the entire isosurface generation process.

From these timing results, it is possible to generate speedup and efficiency graphs which illustrate the effectiveness of the implementation in terms of minimized execution time and maximized processor utilization. Speedup and efficiency graphs were generated based upon the time required for splatting sub-images, and for the total rendering time (including splatting time, composition time, and host communication time). The efficiency and speedup figures for the largest dataset, the 128x128x109 brain set, are based upon the lowest dimension hypercube on which the data could be accommodated. For example, Table 5.15 shows that the program would not run on fewer than four processors due to memory constraints. Thus, in developing the speedup and efficiency curves for this set of renderings, the speedup on 4 processors was assumed to be 4.0, and all measurements for cubes of larger dimension are referenced to this figure.

Figure 5.5 shows the speedup curves of the splatting process generating only local sub-images on all datasets for a fixed image size of 200x200 pixels. Figures 5.6 to 5.8 show the efficiency of processor utilization by this portion of the rendering process for different image sizes. Figures 5.9 to 5.11 show the total rendering times for each combination of dataset and image size. Figure 5.12 shows the speedup for the total rendering process over all datasets on a fixed image size of 200x200 pixels, and Figures 5.13 to 5.15 depict the processor efficiencies for the entire rendering process.

These graphs illustrate several key points about this algorithm. First, the splatting portion of the rendering process is very amenable to parallelization. The graphs show that as the amount of work increases, so does the efficiency of the splatting algorithm. The amount of work is increased by increasing either the size of the image to be generated, the size of the dataset being processed, or both. For example, Figure 5.6 shows that, for a fixed image size larger datasets use increasing numbers of processors more efficiently, while Figure 5.8 shows that all datasets achieve a higher efficiency as more processors are added for larger image sizes. The decreased efficiency at lower workloads is caused by overhead in the algorithm. For a small dataset, there is relatively little work to do, and the nodes spend a higher percentage of their time in synchronization and other overheads, reducing efficiency of processor utilization. As the amount of work increases, the relative portion of time which the array spends in synchronization and overhead drops dramatically, resulting in higher speedups and processor efficiency.

The total rendering process, however, does not perform as well as the splatting algorithm itself. The graphs for total rendering performance indicate that, while the splatting speedup is nearly linear on large datasets, this is not the case for the rendering process as a whole. The reduction in performance is due almost entirely to the sequential nature of the image composition. The splatting time decreases substantially as more processors are used in the computation, but the amount of communication increases as the root processor must get all other processors' sub-images. In fact, Table 5.12 shows that image composition time actually overtakes splatting time on 64 processors; similar trends are evident in the other tables. As discussed previously, research into improving the performance of the composition process is being conducted. Future research in this area will include attempts to streamline the image composition process.

Despite the less-than-ideal performance of the rendering process as a whole, it is important to remember that a speedup is still achieved; for example, the MR dataset on 64 processors has a speedup of just under eight compared to the rendering time on four processors. Practically, this means that the user will be able to interact with the dataset eight times faster on 64 processors than on four. In this type of task, total wall clock time is more important than the somewhat arbitrary efficiency metrics to the user waiting for a visualization. In certain applications, the user does not care that the processors are only being used to 50% efficiency, but only that the final image is generated in x seconds instead of Nx seconds.

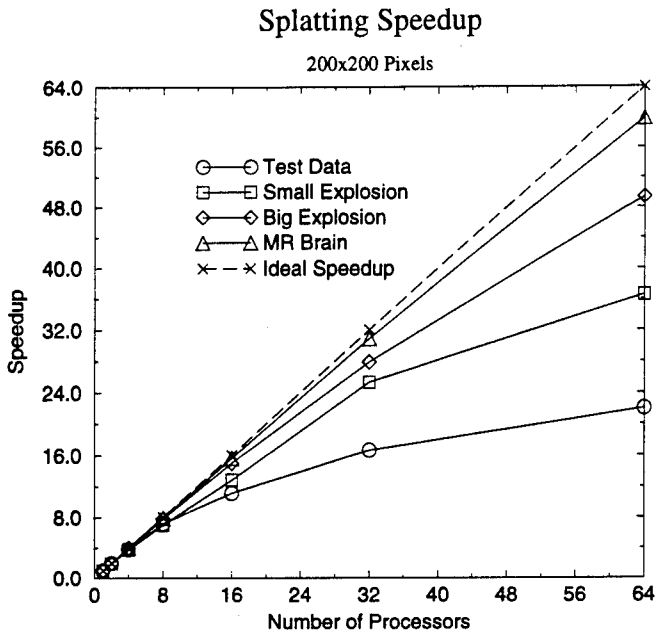


Figure 5.5: Splat speedup for all datasets, 200x200 image.

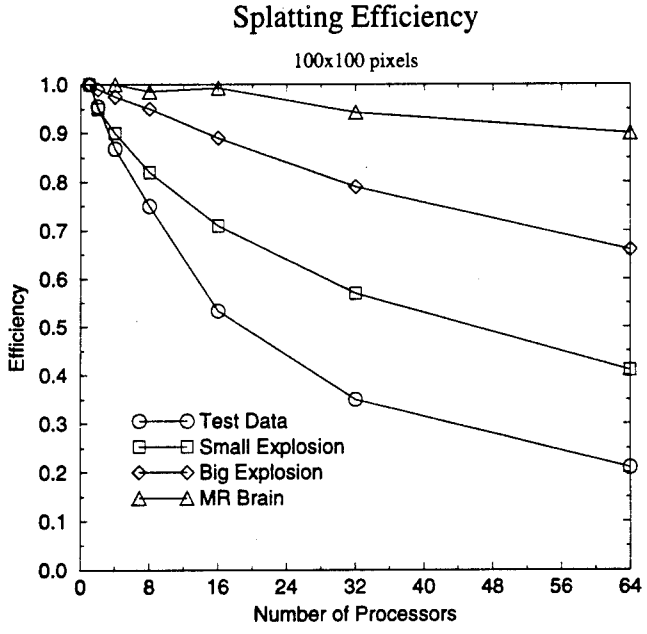


Figure 5.6: Splat efficiency for all datasets, 100x100 image.

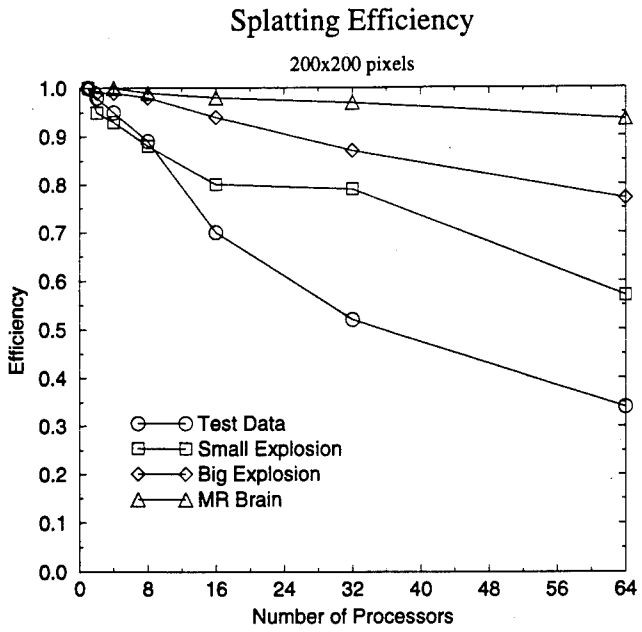


Figure 5.7: Splat efficiency for all datasets, 200x200 image.

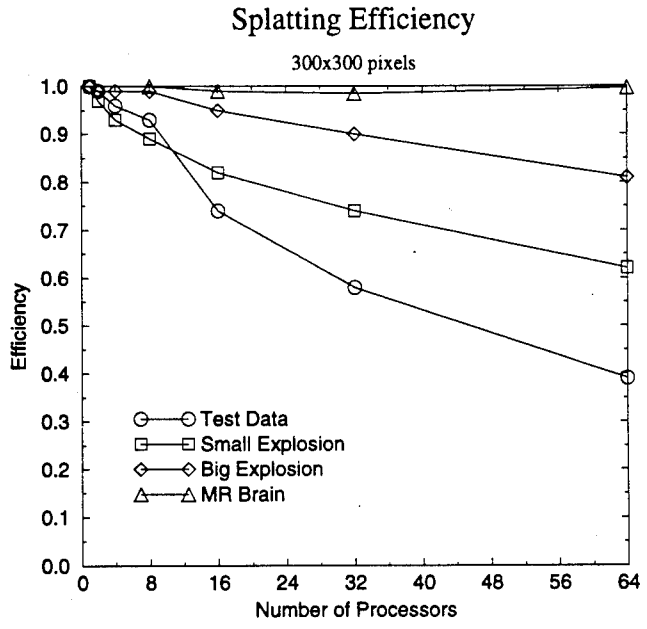


Figure 5.8: Splat efficiency for all datasets, 300x300 image.

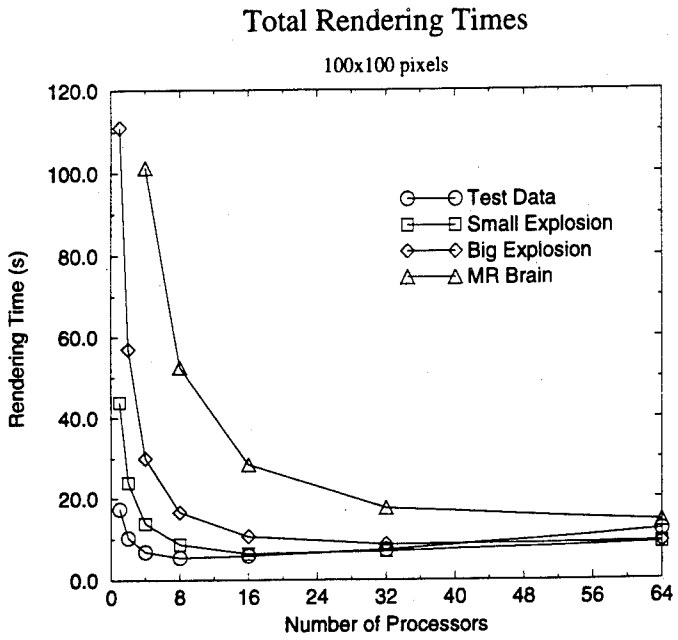


Figure 5.9: Total rendering times (in seconds), 100x100 image.

One final point to note from Figures 5.9 to 5.11 is that the rendering times appear to be converging to a minimum value; for the two smaller datasets, the test and small explosion data, the rendering times actually increase as more processors are added beyond eight and sixteen processors. This implies that this implementation of the splatting algorithm will not scale well for arbitrarily large numbers of processors on arbitrary datasets. Rather, care will have to be taken in adding processors to the computation such that the percentage of time spent in communication does not overwhelm the time spent in performing the splat itself. This is not a surprising result, given the sequential nature of the image composition algorithm, and is a classic example of the type of performance limitation given in Amdahl's law [1].

5.6 Comparison with Single Processor Performance

Timing measurements were also taken on one processor of a two processor Silicon Graphics Onyx machine with 100 MHz R4400 processors, 256 MBytes of two-way interleaved memory with RE2 graphics using a version of the code without any parallel overhead. It is important to note that this particular machine has processor technology several years more advanced than the nCUBE processors, and each of its processors is much faster than the single processors on the nCUBE. This comparison is not meant to demonstrate the superiority of the nCUBE's performance over the SGI's, but rather to give some indication of the performance of the code on a single processor machine without the parallel overhead. The single processor C code was compiled using the '-O' option for optimization. Compiler optimization improves the performance of the sequential code by a factor of 2.0. The nCUBE codes were also compiled with this option, but there was no change in performance with the compiler optimization turned on.

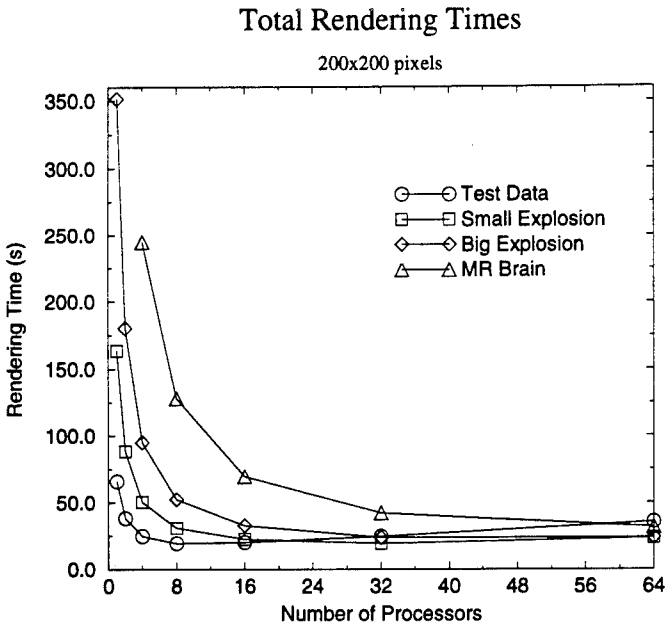


Figure 5.10: Total rendering times (in seconds), 200x200 image.

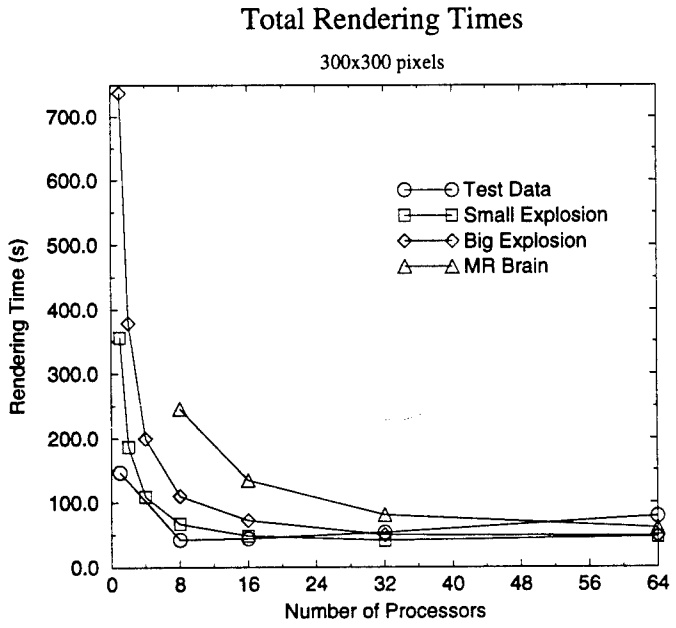


Figure 5.11: Total rendering times (in seconds), 300x300 image.

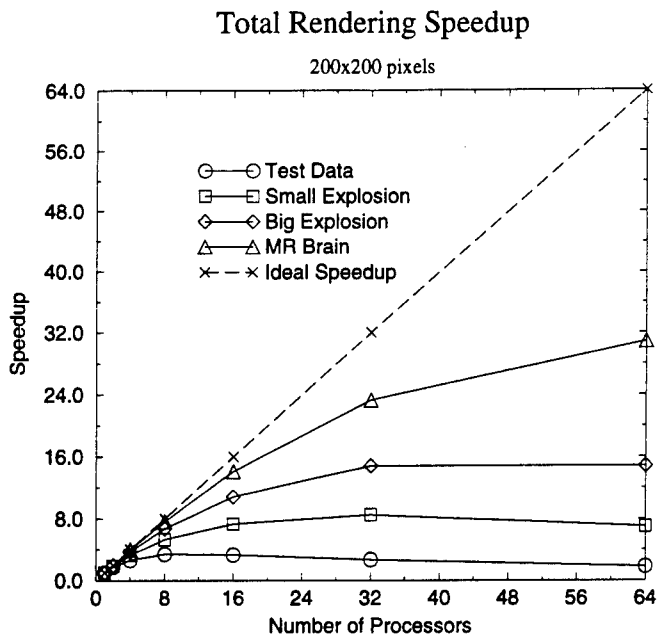


Figure 5.12: Total rendering speedup, 200x200 image.

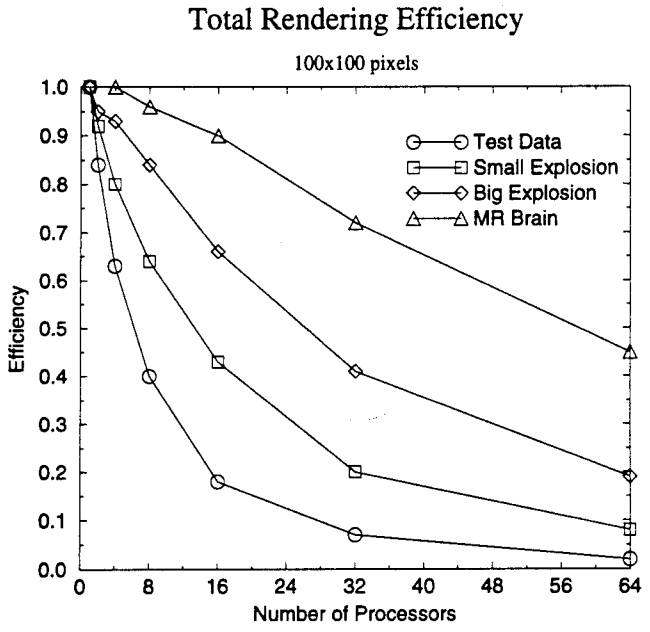


Figure 5.13: Total rendering efficiency, 100x100 image.

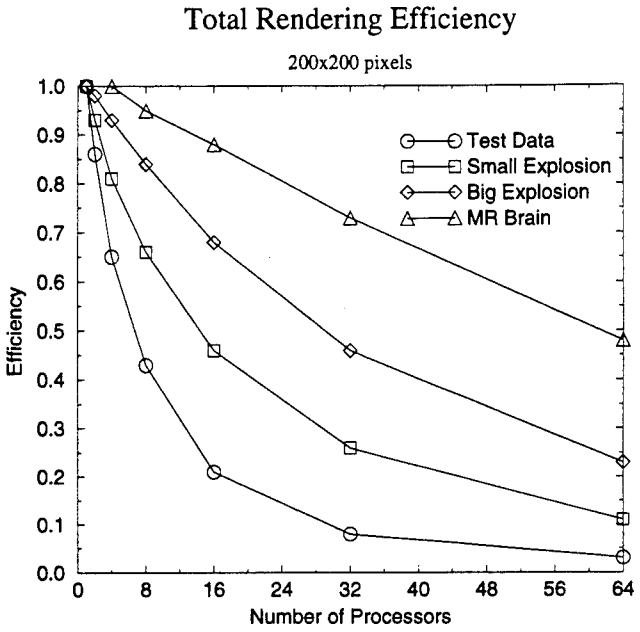


Figure 5.14: Total rendering efficiency, 200x200 image.

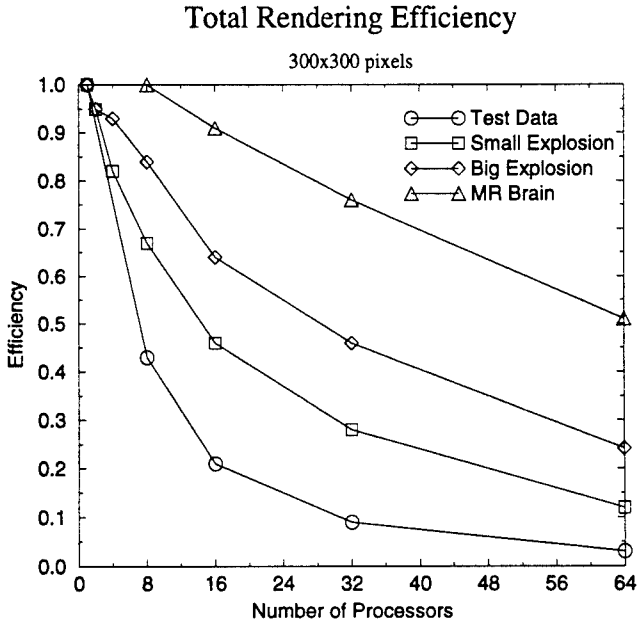


Figure 5.15: Total rendering efficiency, 300x300 image.

Dataset	Time (s)
Test	0.00
Big Explosion	0.60

Table 5.17: Onyx single processor isosurface generation times.

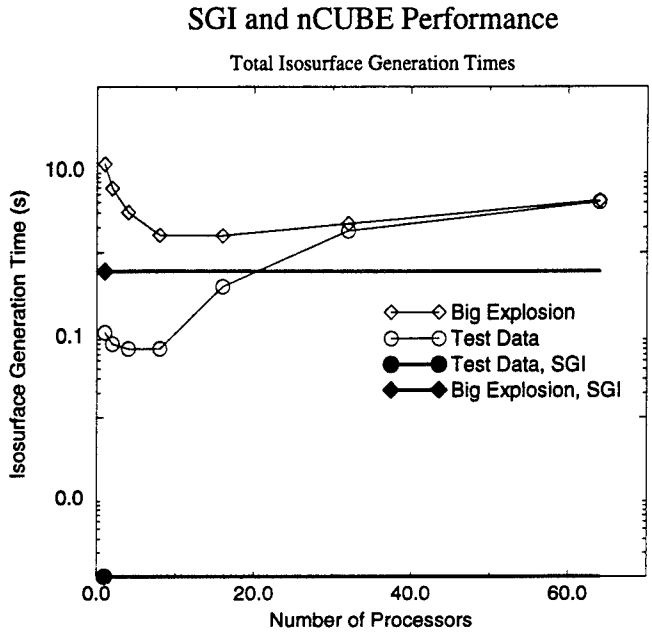


Figure 5.16: Comparison of total isosurface generation times.

5.6.1 Marching Cubes

Table 5.17 lists the times to construct isosurfaces from the 10x10x10 test dataset and the 66x66x50 large explosion datasets used in the parallel tests. As in the parallel tests, the isovalues chosen were 0.87 for the test set and 49.99 for the explosion dataset; the final isosurfaces yielded 364 and 14914 triangles each. The times given are in seconds, and don't include the time to read in the data or draw the final isosurface.

Figure 5.16 compares the uniprocessor isosurface generation times to the times to generate the same surfaces on various nCUBE processor configurations. This graph shows that, for all processor configurations on both datasets, the SGI outperforms the nCUBE. This is not surprising, given the poor performance characteristics of the parallel marching cubes algorithm discussed in Section 5.4. It may be the case that further experimentation and analysis will show that the marching cubes algorithm is simply not worth parallelizing on machines with a communication network topology like the nCUBE, because of the large sequential fraction in the algorithm.

Dataset	Time (s)
Test	5.96
Small Explosion	17.00
Big Explosion	33.24
MR Brain	73.20

Table 5.18: Onyx single processor rendering times.

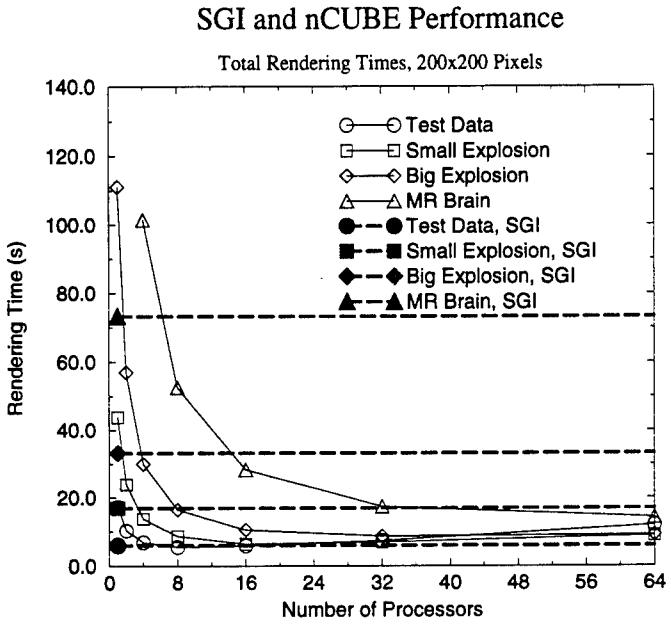


Figure 5.17: Comparison of total rendering times for splatting a 200x200 pixel image.

5.6.2 Splatting

Table 5.18 gives the times to render the four test datasets into a 200x200 pixel image on the Onyx. Figure 5.17 compares the Onyx rendering times for each dataset on a single processor to the nCUBE times on varying processor configurations. In both cases, the times given are the times for the entire rendering process, including image composition, on the parallel and single processor machines. The figures do not include the time needed to read and distribute the data, or the time to display the scene over the network.

A single processor of the Onyx has a peak performance of approximately 17 Mflops, while a single processor of the nCUBE has a peak performance of only 4.25 Mflops. Figure 5.17 indicates that the SGI outperforms a single processor of the nCUBE by roughly four to one, which is reasonable given the peak performance of each processor.

Chapter 6

Conclusions and Future Work

Two commonly used volume visualization algorithms, splatting and marching cubes, have been successfully adapted to the nCUBE 2, a distributed memory parallel MIMD machine. This investigation has yielded a working parallel scientific visualization system which demonstrates the feasibility of parallel volume visualization, and also highlights many areas for future effort. The current system is viewed as a prototype for further investigation.

There are several ways in which the implementation of the marching cubes algorithm could be improved. First, the number of triangles generated for any given isovalue will depend upon the isovalue itself and its popularity in the dataset, and the density of the dataset. For example, an isosurface which is a flat plane can be described by only two triangles without any loss of information; yet the actual number of triangles which will be generated is directly dependent upon the number of voxels intersecting the isosurface. The number of triangles involved in describing a surface is of interest for two reasons: they must be sent to the GUI for display by messages, incurring communications costs, and they must then be displayed by the GUI - the more triangles, the longer it takes to draw the surface. It may therefore be desirable to apply some measure of the surface gradient and coalesce triangles in areas of low surface curvature. This would result in surface degradation and loss of information (in all but the planar case), but the subsequent reduction in display and communication time (which is the slowest portion of the code, as illustrated by the timing results) would be more than offset - the impact of surface degradation could be minimized by the addition of user-specified parameters which would afford control over the processed surface's fidelity to the original.

Another area wherein the algorithm's runtime could be enhanced deals directly with the display device. Silicon Graphics rendering engines are optimized for certain graphics primitives. The primitive of particular interest to us is the triangular mesh, or *tmesh*. A surface described by triangles which has been put into tmesh format would render significantly faster than the same surface described as individual triangles. This would not only reduce render time, but since putting the surface in this format would eliminate any redundant triangles (of which there are a great deal), it would also reduce the amount of communication necessary between each process and the GUI. Finally, the process by which the local isosurface is composited to form the total isosurface needs to be explored to see if a new technique can be developed to allow the surfaces to be composited in parallel; this has proven to be a difficult task, given the limitations of the nCUBE communication buffer.

The are also several areas in which the parallel splatting algorithm may be enhanced. As mentioned above, research into a new method for compositing sub-images and delivery of the final

image to the host needs to be conducted to determine if a larger portion of the process can be performed in parallel. Also, if each node compressed its sub-image using a lossless compression technique before participating in the composition process, the communication costs could be significantly reduced. However, care must be taken that the time spent in compression is less than the time saved in communication.

Future work will also focus upon porting the prototype implementations of each algorithm developed thus far to other architectures, and studying how each algorithm performs given different communication and processor structures. It may be that splatting and marching cubes are particularly suited for specific architectures and communication protocols. This task should be accomplished with less difficulty than is normally encountered in porting codes between different parallel architectures by virtue of the unified interface provided by MPI.

Finally, the data visualization techniques examined in this report are scalar data techniques. An interesting research topic would be to extend the prototype system to include vector visualization techniques, adapted to the nCUBE's architecture.

Bibliography

[1] Gene Amdahl. The validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

[2] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A user's guide to PVM: Parallel virtual machine. Technical Report ORNL/TM-11826, Mathematical Sciences Section, Oak Ridge National Laboratory, September 1991.

[3] J. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. *Computer Graphics*, 16(4):21–30, July 1982. Proceedings of SIGGRAPH '82.

[4] Ralph Butler and Ewing Lusk. User's guide to the p4 programming system. Technical Report ANL-92/17, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, October 1992.

[5] J. Challinger. *Scalable Parallel Direct Volume Rendering for Nonrectilinear Computational Grids.* PhD thesis, University of California, Santa Cruz, December 1993.

[6] L. Cline, H. Lorensen, S. Ludke, C. Crawford, and B. Teeter. Two algorithms for the three-dimensional reconstruction of tomograms. In A. Kaufman, editor, *Volume Visualization*, pages 64–71. IEEE Computer Society Press, 1991.

[7] R. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics*, 22(4):65–74, August 1988. Proceedings of SIGGRAPH '88.

[8] Message Passing Interface Forum. Mpi: A message-passing interface standard. Technical Report Computer Science Department Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 5 1994. To appear in the International Journal of Supercomputing Applications, Volume 8, Number 3/4, 1994.

[9] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors.* Prentice Hall, 1988.

[10] G. Frieder, D. Gordon, and A. Reynolds. Back-to-front display of voxel-based objects. *IEEE Computer Graphics and Applications*, 5(1), January 1985.

[11] A. Globus. Gridigrator: A very fast volume renderer for 3d scalar fields defined on curvilinear grids. Technical Report Report RNR-92-001, NAS Systems Division, Applied Research Branch, NASA Ames Research Center, Mail Stop T-045-1, Moffett Field, Ca 94035, 1992.

[12] R. Lenz, B. Gudnumdsson, B. Lindskog, and P. Danielsson. Display of density volumes. *IEEE Computer Graphics and Applications*, 6(7), July 1986.

[13] M. Levoy. Volume rendering: Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(5), May 1988.

[14] T. Lewis and H. El-Rewini. *Introdution to Parallel Computing.* Prentice Hall, 1992.

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

[16] K. Ma, J. Painter, and C. Hansen. A data distributed, parallel algorithms for ray-traced volume rendering. IEEE Computer Graphics and Applications revised submission, March 1994.

[17] P. Sabella. A rendering algorithm for visualizaing 3d scalar data. *Computer Graphics*, 22(4):51–58, August 1988. Proceedings of SIGGRAPH '88.

[18] L. H. Turcotte. A survey of software environments for exploiting networked computing resources. Technical Report MSSU-EIRS-ERC-93-2, Mississippi State University/NSF Engineering Research Center for Computational Field Simulation, 1993.

[19] C. Upson and M. Keller. Vbuffer: Visible volume rendering. *Computer Graphics*, 22(4):59–64, August 1988. Proceedings of SIGGRAPH '88.

[20] S. P. Uselton. Volume rendering for computational fluid dynamics: Initial results. Technical Report Report RNR-91-026, NAS Systems Division, Applied Research Branch, NASA Ames Research Center, Mail Stop T-045-1, Moffett Field, Ca 94035, September 1991.

[21] L. Westover. Footprint evaluation for volume rendering. *Computer Graphics*, 24(4):367–376, August 1990. Proceedings of SIGGRAPH '90.

[22] Robert A. Whiteside and Jerrold S. Leichter. Linda for supercomputing on a local area network. In *Proceedings of Supercomputing 1988*, pages 192–199, Orlando, Florida, October 1988.

Appendix A

Color Images

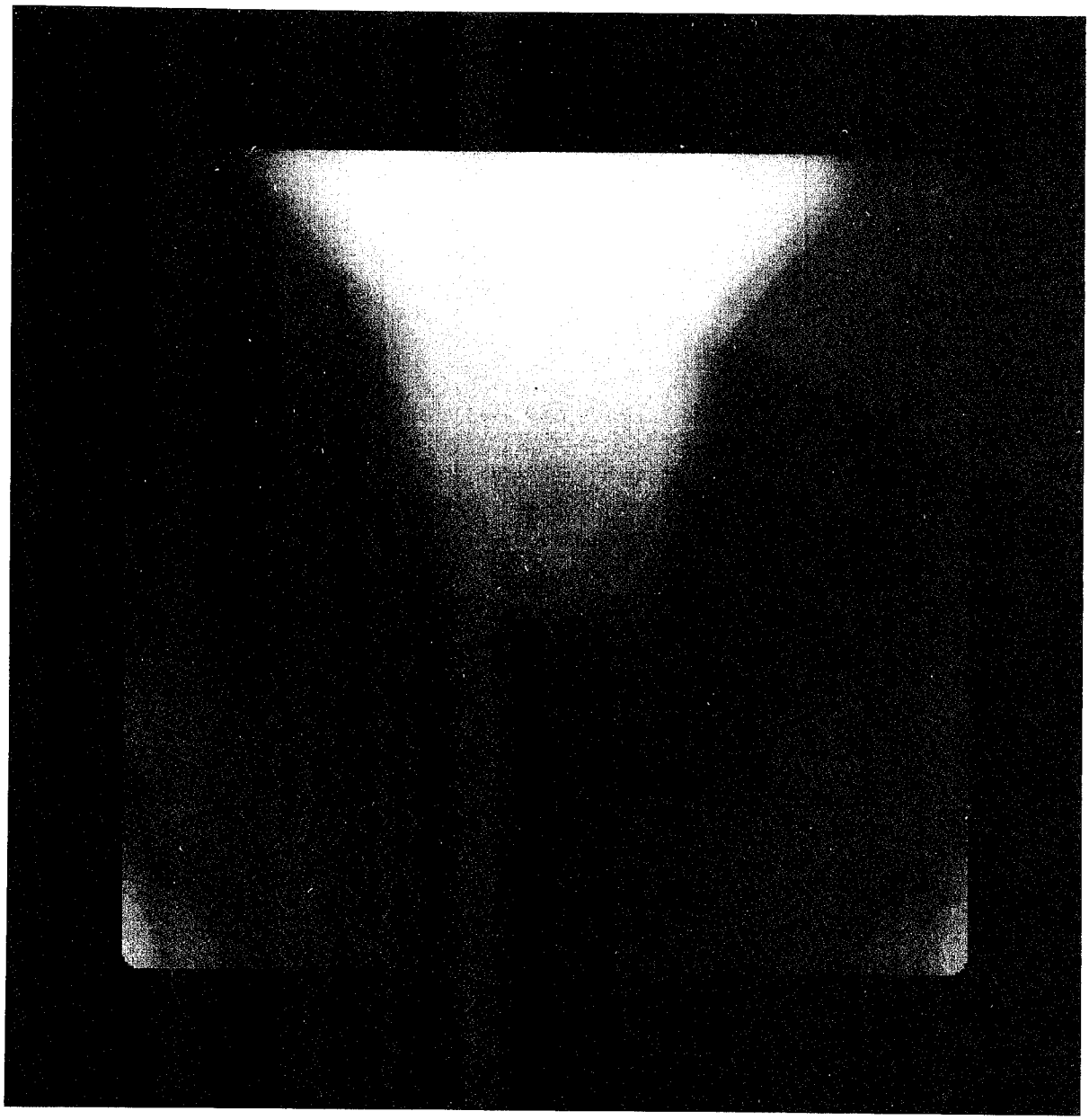


Figure A.1: 350x350 pixel splat image of the big explosion dataset.

In each of the images which follow, it is important to remember that the image reproduced in this report do not necessarily match with images produced on the screen during a rendering. There is a considerable loss of detail and variation in colors and intensity inherent in the creation of hardcopy.

This image is a volume rendering of the larger explosion dataset. The scalar values in the dataset represent the velocity of soil particles as measured with an array of accelerometers during an underground explosion. The explosion occurs at the top center of the picture. The color scale maps low velocity values to blue through purple, green, and yellow up to red.

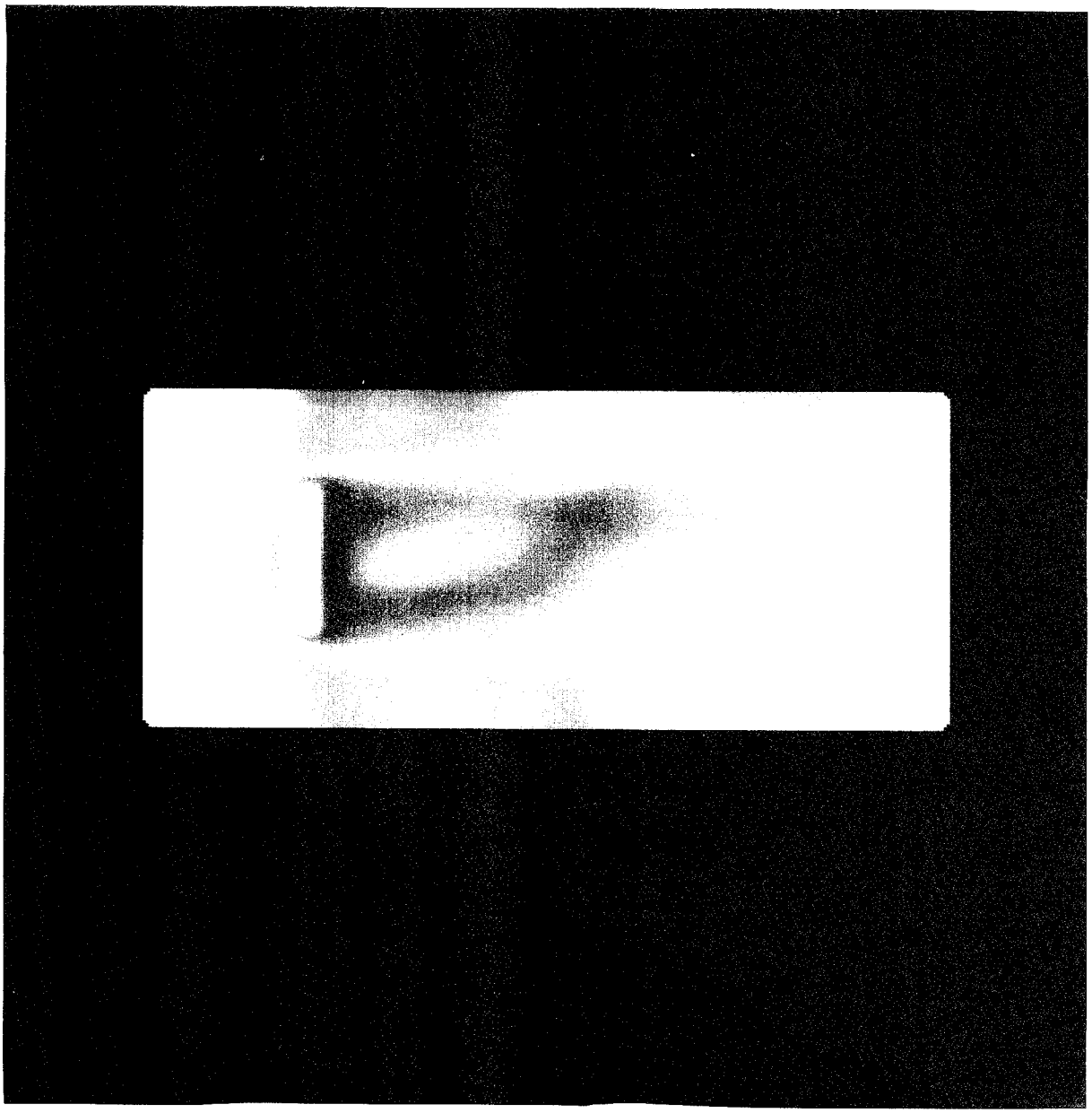


Figure A.2: 350x350 pixel splat image of fluid flow past a square plate.

This image is a volume rendering of computational fluid dynamics data. The data in the volume is fluid velocity values for flow past a square plate, shown in profile as a white rectangle in the image. The color scale maps fluid velocity from blue to green, yellow, and red, up to light pink. Fluid flows from left to right in the image.

Figure A.3: 350x350 pixel splat image of CT data for a human hip.

This image is a volume rendering of a CT dataset for a human hip. Bone is rendered as nearly opaque white material, and soft tissues are rendered as nearly transparent green material. The patient's pelvis is broken, and it is clearly evident in the image that a large fragment of bone has separated from the pelvis. Different views of this dataset reveal that the bone fragment is behind the pelvis, and is barely visible in this image below and to the right of the fracture.

Figure A.4: 350x350 pixel splat image of MR brain dataset.

This image is a volume rendering of an MR image of a human skull. After the MR scan, the skull cap and surrounding tissue were edited out of the data to reveal the brain more clearly. The patient is facing to the left, and some facial features are clearly evident in this image. An attempt was made to enhance the vascular structure in this image by finding the range of values it encompasses and mapping them to red values. Low tissue values in this image appear as grey values, and values higher than the red values used for the brain tissue map through purple to full scale blue.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1995	Final report

4. TITLE AND SUBTITLE
An Investigation of the Parallel Implementation of Two Volume Visualization Techniques on the nCUBE 2

5. FUNDING NUMBERS
Contract number
DAALO3-89-C-0038

6. AUTHOR(S)
John E. West, Michael M. Stephens, Louis H. Turcotte

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Engineer Waterways Experiment Station
3909 Halls Ferry Road
Vicksburg, MS 39180-6199

8. PERFORMING ORGANIZATION REPORT NUMBER
Technical Report
ITL-95-2

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Army Corps of Engineers
Washington, DC 20314-1000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Available from the National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report presents a divide-and-conquer approach to developing visualization software for scientific analysis of high resolution, volume datasets on distributed memory parallel computers. Two complementary methods for exploring volume datasets are discussed, isosurface generation and direct volume rendering, and the logic behind the design of parallel implementations of the original sequential algorithms is highlighted. For both algorithms, the data, once distributed, remains in place regardless of the viewer's location, and no interprocessor communication is required during scene generation. Development of the algorithms' communication structure is also discussed, with an introduction to issues surrounding parallel I/O and data distribution for these implementations. Finally, basic parallel performance metrics will be outlined, and test results indicating the performance results of the parallel implementations on an nCUBE 2 will be discussed.

14. SUBJECT TERMS
Marching cubes
Parallel computing
Scientific visualization
Splatting
Volume visualization

15. NUMBER OF PAGES
61

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED		

Destroy this report when no longer needed. Do not return it to the originator.