Exploiting Parallel Memory Hierarchies for Ray Casting Volumes

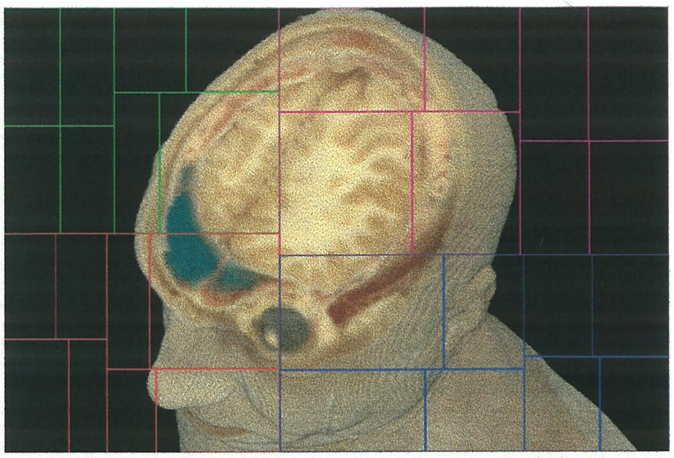
Thesis by

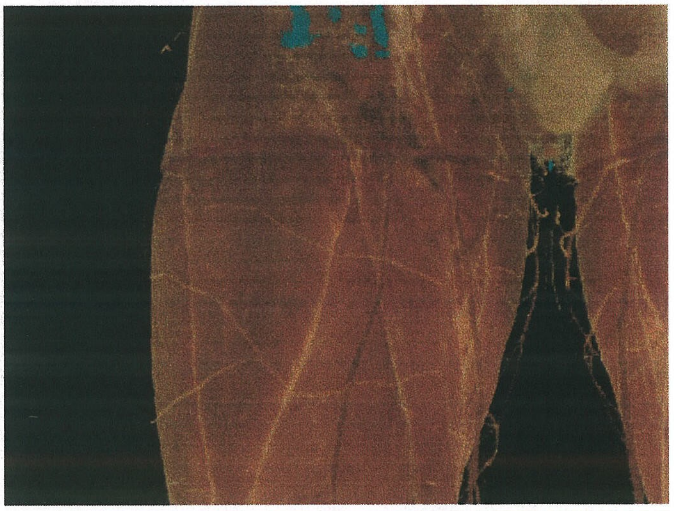
Michael E. Palmer

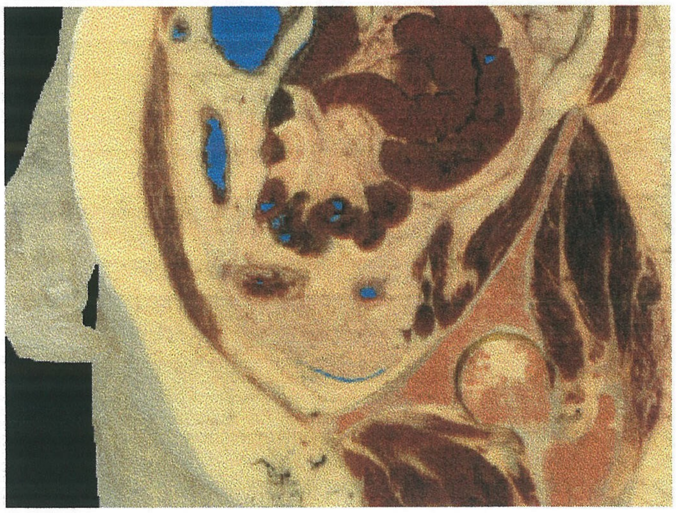
In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

1997
(Submitted April 4, 1997)


The Visible Male dataset (357 MB). Squares in this *image partition* correspond to individual processors. Colors correspond to separate nodes of a Power Challenge Array.


Manipulation of the mapping between voxel values and opacity reveals the internal musculature and vasculature of the leg of the Visible Male.


A view of the hip joint of the Visible Female dataset. In this work, we render the Visible Female at full spatial resolution: 1610x894x5192 voxels, for a total of 7.1 GB, the largest volume dataset ever rendered.

Acknowledgements

This thesis would not have been possible without the help of many people. First of all, I would like to thank my family and my fiancée, Jessica Liu, for their unconditional love and support.

I would like to thank my advisor, Stephen Taylor, for all his advice and support over the past six years. Steve is truly a man of integrity and generosity. He combined a respect for letting me follow my own path, with just enough gentle nagging to make sure I would submit my work for publication on a periodic basis.

Thanks to Brian Totty, who contributed a great deal to this thesis. He promoted my work within Silicon Graphics, and collaborated with me on several papers. Brian is also responsible for most of the performance-tuning optimizations to the ray casting kernel. Brian also contributed the tongue-in-cheek name for my experimental ray casting code: *RendAsunder*

I would like to thank Paul Woodward, Steve Anderson, and the rest of the Laboratory for Computational Science and Engineering (LCSE) at the University of Minnesota. The work I did there in the summer of 1994 was the genesis of *RendAsunder*, my experimental ray casting code. The LCSE also provided the vorticity dataset used in many of my experiments. Steve Anderson has been a good friend to me, gave me much technical advice when I was just beginning on the Silicon Graphics platform, and contributed to early versions of *RendAsunder*. Some of his code is still running in the current version.

I would like to thank Silicon Graphics for the opportunity to participate in Supercomputing '94 and '95. Silicon Graphics also gave me the opportunity to present *RendAsunder* at two other conferences. Specifically, thanks to John Brown of Silicon Graphics for taking an interest in *RendAsunder*, promoting its development, and spoiling me with ample machine time on a Power Challenge Array.

Further machine time, in particular for the experiments in chapter 6 and for the rendering of the largest volume dataset ever, the 7.1 GB Visible Female dataset, was provided by the National Center for Supercomputing Applications (NCSA) at the University of Illinois, Urbana-Champaign. The excellent operations staff at NCSA were very helpful.

Jerrell Watts' SCPlib message passing library allowed me to achieve high performance on the NCSA machines. Jerrell went above and beyond the call of duty several times with late night emergency sessions of performance tuning.

Thanks to John Thornley for discussions of my thesis and journal paper, and for advice on how to handle the whole process of putting together a Ph.D. thesis.

Thanks to H. Ross Harvey for contributing the Power Challenge "cache bashing" benchmark in Chapter 5.

Thanks to Peter Schröder for giving me access to the machines in his lab to finish up many odds and ends.

Thanks to Al Barr and Cindy Ball of the Graphics Group at Caltech for access to, and much help with, video equipment, respectively.

Thanks to Michael Ackerman of the Visible Human Project at the National Library of Medicine for providing the Visible Male and Visible Female datasets.

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) under contract number DABT63-95-C-0116, and AASERT award number N0014-93-1-0843.

Abstract

Previous work in single-processor ray casting methods for volume rendering has concentrated on algorithmic optimizations to reduce computational work. This approach leaves untapped the performance gains which are possible through efficient exploitation of the memory hierarchy.

Previous work in parallel volume rendering has concentrated on parallel partitioning, with the goals of maximizing load balance and minimizing communication between distributed nodes. This implies a simplified view of the memory hierarchy of a parallel machine, ignoring the relationship between parallel partitioning and memory hierarchy effects at all but the top level.

In this thesis, we progressively develop methods to optimize memory hierarchy performance for ray casting: 1) on a uniprocessor, using algorithmic modifications to isolate cache miss costs, specialized hardware to monitor cache misses on the bus, and a software cache simulator; 2) on the a shared-memory Power Challenge multiprocessor, examining the fundamental dependence of algorithmic design decisions regarding parallel partitioning upon memory hierarchy effects at several levels; and 3) on a distributed array of interconnected Power Challenge multiprocessors, on which we implement a logical global address space for volume blocks, and investigate the tradeoff between replication (caching) and communication of data.

The methods we develop permit us to exploit the coherence found in volume rendering to increase memory locality, and thereby increase memory system performance. This focus on the optimal exploitation of the entire memory hierarchy, from the processor cache, to the interconnection network between distributed nodes, yields faster frame rates for large (357 MB to 1 GB) datasets than have been previously cited in the literature, and allows us to efficiently render a 7.1 GB dataset, the largest ever rendered.

Our results have implications for the parallel solution of other problems which, like ray casting, require a global gather operation, use an associative operator to combine partial results, and contain coherence. We discuss implications for the design of a parallel architecture suited to solving this class of problems, specifically, that these algorithms are best served by a deep memory hierarchy.

Contents

List of Figures

Chapter 1 Introduction

1.1 Motivation

As the data sets generated by computational simulations and medical imaging grow larger, scientists are swamped by their large (multi-gigabyte) three-dimensional datasets. The most intuitive way for a scientist to understand these vast volumes of data is to explore them interactively through high-performance volume rendering. Interactivity requires certain minimum frame rates (e.g., 10 frames per second). This level of performance for rendering very large datasets is not possible on contemporary single-processor machines. Parallelism offers the promise of rendering large datasets at interactive frame rates. However, the leap to parallelism introduces a host of new algorithmic design considerations. One of the most important is the consideration of the interaction between parallel partitioning and efficient exploitation of the memory hierarchy at all levels.

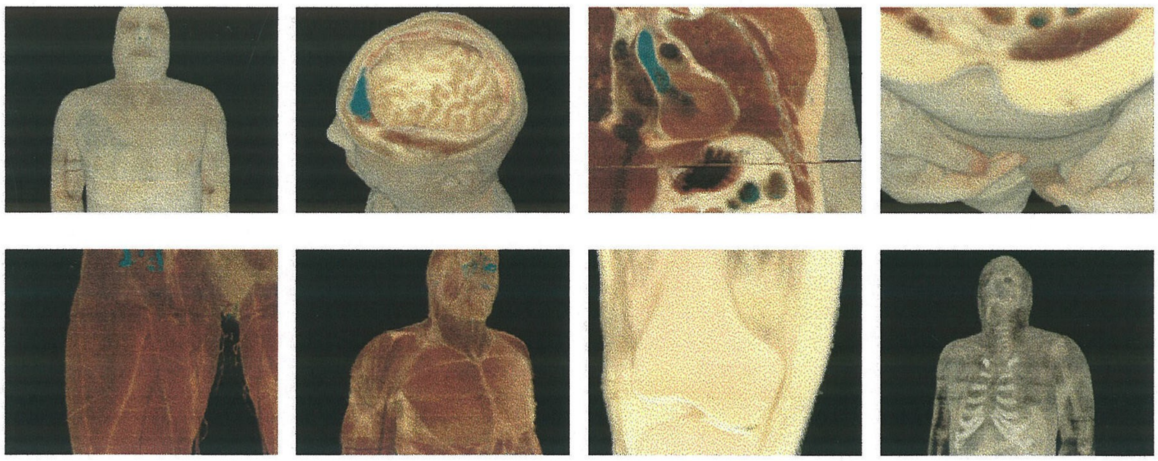


Figure 1.1: Example ray casting output, male dataset

Figure 1.1 shows example images generated by the ray casting volume rendering implementations described in this thesis using the Silicon Graphics Power Challenge.

Manipulation of the mapping between voxel value and opacity, and use of a front clipping plane, yield views of the interior muscle and bone structure of this human body dataset [Ackerman95].

1.2 Contributions of this thesis

Early research in uniprocessor ray casting volume rendering focused on algorithmic optimizations to improve performance; for example, by reducing the total number of rays that must be cast, or the number of intersections between rays and data voxels that must be calculated. However, previous work has left largely untapped the performance benefits to be gained through optimal exploitation of the memory hierarchy. The primary means to achieve this on a uniprocessor is by increasing locality of memory access through the manipulation of the order of memory accesses.

Past research in parallel volume rendering has concentrated on the partitioning problem, with the specific goals of maximizing load balance and minimizing communication between distributed processors. Past work has not, in general, considered the more complex problem of exploiting the entire memory hierarchy of the parallel machine. The focus on communication between distributed processors, to the exclusion of other levels of the memory hierarchy, implies an oversimplified model of the operation of the entire machine. A parallel computer is a machine dedicated not only to performing operations on data, but also to moving the data efficiently from the memory locations where it is stored, through many levels of a deep memory hierarchy, to the processor registers where it can be operated on. Hardware from the level of the processor cache, to the interconnection network between distributed nodes, plays a role in this task. The performance of each level of this system is extremely sensitive to algorithmic characteristics such as locality of memory access. Consideration of the efficient exploitation of all levels of a deep memory hierarchy must be reflected in algorithmic design, and fundamentally affects design decisions regarding parallel partitioning.

Although poor load balance is commonly the focus of parallel rendering work, we will show that poor performance at any level of the memory hierarchy can also thwart parallel speedup. Optimal performance at all levels of the memory hierarchy is even harder to achieve than optimal load balance, in part because memory hierarchy effects can be complex and unintuitive.

This thesis focuses on the volume rendering of volumetric data defined on regular cartesian grids by ray casting. Regular grids, because of their simplicity, are the obvious place to begin the characterization of memory hierarchy effects for ray casting. They are used commonly in a range of applications, from magnetic resonance image (MRI) to some computational fluid dynamics (CFD) work.

Our primary experimental hardware platform is the Silicon Graphics Power Challenge Array, an example of a hybrid distributed/shared memory architecture, consisting of a distributed array of up to eight Power Challenge shared-memory multiprocessors, each containing up to eighteen MIPS R8000 processors.

In this thesis, we progressively develop techniques to maximally exploit the memory hierarchy on a uniprocessor; within a shared-memory Power Challenge multiprocessor; and finally on a distributed array of interconnected Power Challenge multiprocessors.

On a single processor, we focus on the optimization of the performance of the level one (L1) and level two (L2) caches of the R8000 microprocessor. Our investigative techniques include 1) algorithmic changes to isolate the costs of cache misses; 2) direct observation of L2 cache misses using a limited-edition hardware bus-snooping board; and 3) simulation of both L1 and L2 cache misses with a software cache simulator, which can be configured to mimic the caching behavior of the L1 and L2 caches of the R8000.

On multiple processors within a single shared-memory Power Challenge node, we investigate the relationship between parallel partitioning and cache performance, studying both image-based and object-based parallel partitions. We evaluate these partitioning options in terms of achievable cache hit rates, load balance, parallel speedup, the tendency to saturate the shared bus as the number of processors increases, and the applicability of common algorithmic optimizations to ray casting.

Across multiple distributed Power Challenge Array nodes, we implement in software a system providing a logical global address space for accessing and caching blocks of volume data. We investigate both image-based, and object-based, parallel partitioning algorithms for distributed machines, evaluating them in terms of volume of data communication, attainable load balance, parallel speedup, and their performance under conditions of limited memory resources for use by the distributed block caching system.

This attention to the performance of each level of the deep memory hierarchy yields higher frame rates for large volume datasets than we have previously seen cited in the literature, both for large datasets (357 MB to 1 GB) on a single shared-memory array node, and for very large datasets (up to 7.1 GB) on multiple array nodes, which have not been previously rendered.

Our findings have implications to other parallel algorithms, which share with ray casting volume rendering the characteristics of: 1) they require a global gather operation (i.e., the entire distributed dataset contributes to a solution that will end up stored on a single node), 2) the operator which combines partial results is associative, and 3) they contain some form of coherence which can be exploited to increase memory locality. These algorithms are not scalable, because the communication they require increases with increasing numbers of nodes. However, the coherence they contain can be exploited on a machine with a sufficiently deep memory hierarchy to obtain good parallel speedup on moderate numbers of nodes.

This, in turn, has implications to the design of parallel architectures. The ideal architecture for solving these problems on moderate numbers of processors is not a distributed architecture with a slow but "scalable" interconnection network. Rather, expansion to moderate numbers of parallel processors should be achieved by creating a deeper memory hierarchy. The top level of the memory hierarchy should be implemented by connecting shared-memory multiprocessors with a high performance network that is scalable only to a low number of connected nodes. (This is unlike the "scalable" but relatively low performance HIPPI crossbar network used in the Power Challenge Array.)

Chapter 2 Background Material and Terminology

In this chapter, we define a number of terms and concepts relevant to parallel volume rendering. Some of these concepts are developed in more detail in other chapters as they become relevant. More detailed discussion of closely related papers is presented in Chapter 7, Related Work.

2.1 Volume rendering

Volume rendering is the process of rendering two-dimensional images, from a particular viewpoint, from discretized three-dimensional scalar fields. The data to be rendered consists of space-filling discretized regions called *voxels*.

2.1.1 Coordinate systems for computer graphics

We call the coordinate system of the three-dimensional dataset *object space*. In Figure 2.1, this set of coordinate axes is labeled X, Y, and Z. The coordinate system of the viewer is called *image space*, and is labeled U (to the viewer's right), V (the viewer's "up" direction), and N (opposite to the direction in which the viewer is looking). We call the discrete volume elements making up the dataset *voxels* and the discrete squares making up the image plane *pixels*.

As the viewer peers about the dataset, flying around and through it, the relationship between object space and image space changes. This relationship is specified by a coordinate transformation matrix called the *view matrix*, which maps any point in object space to a point in the viewer's image space.

Two methods of projection are commonly used in computer graphics. *Orthogonal projection* assumes that all rays projected from the eye point are parallel, making a

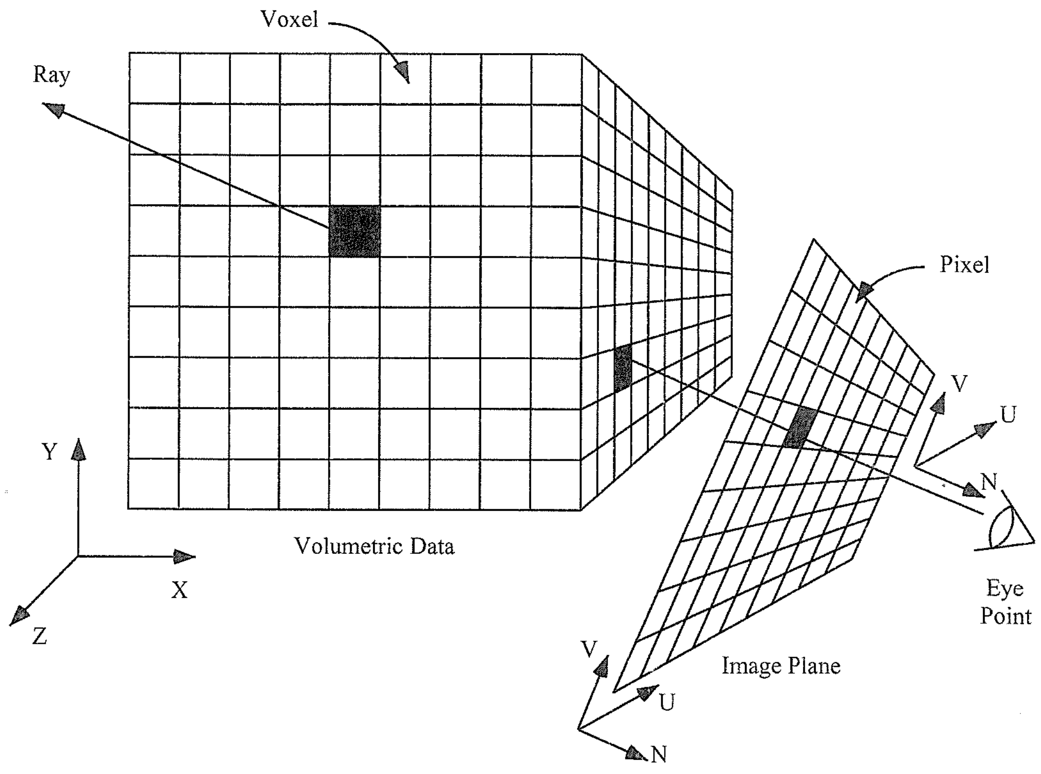


Figure 2.1: Coordinate systems in Volume Rendering

projected cube look like the one on the left of Figure 2.2. The parallel edges of a cube preserve their parallelism in this projection. In our day-to-day experience in viewing the world, we are more familiar with seeing objects in *perspective projection.* In the perspective projection, rays projected from the eye point diverge from the eye point. This projection makes all parallel edges of objects, such as the cube on the right of Figure 2.2, appear to converge. Since projection of a three-dimensional object onto a two-dimensional surface implies a loss of information, the perspective projection gives important *depth cues* to a viewer accustomed to perceiving three-dimensional objects in this way. The orthogonal projection allows certain algorithmic optimizations which increase performance, but it has become less popular in contemporary volume rendering because of the unusual appearance it lends to objects.

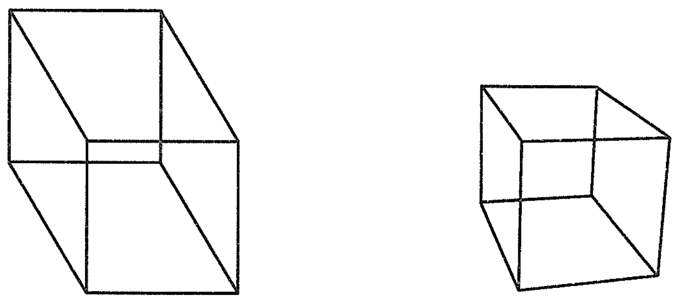


Figure 2.2: Cubes in orthogonal projection and perspective projection

Rendering using the perspective projection requires, in addition to the view matrix, a value specifying the severity of the perspective transformation. This is specified as the *field of view*, the angle that the full width of the image subtends from the eye point. This specifies the angle of divergence between a ray at one edge of the screen and a ray at the other edge.

2.1.2 Representing three-dimensional data

A continuous three-dimensional scalar field must be discretized for storage on a digital computer. A convenient means is *spatial discretization.* We can approximately represent the scalar field with a finite collection of small volume elements, giving each element a single value corresponding to the average value of the continuous scalar field within that volume element. Alternatively, we could approximate the field by storing its value at a finite set of points in space.

Instead of discretizing in space, we could alternatively take a fourier transform of the scalar field, and represent the resulting frequencies in discrete form. There are countless other possible representations. A band-limited function can be exactly represented by a finite set of discrete samples. If the input function is not band-limited, each type of representation yields characteristic deviations, or errors, compared to the original input.

We primarily consider spatial discretization in this work. A variety of geometries can be used in spatial discretization, including regular cartesian grids, rectilinear and curvilinear grids, unstructured grids (e.g., irregular tetrahedra), hierarchical grids (e.g., octrees), and multiresolution grids (i.e., regular blocks of varying resolution, which may or may not overlap).

Regular grids, for instance grids consisting of identical cubes (or other identical solids, like regular tetrahedra), are the simplest form of grid. The entire grid can be uniquely specified by a small number of parameters. For instance, a regular cubical grid can be specified by the dimensions of the grid and the size of one cubical element. There exist very efficient algorithms to find the path of a ray traversing a regular cubical grid.

Slightly more complicated are rectilinear grids. These grids are formed by stretching one or more of the axes of a regular grid. These grids can be completely specified with the same information necessary to specify a regular grid, plus information about the stretching function along each axis. Curvilinear grids are more complex in that the grid lines may curve; however, each data element still has the same number of neighbors (e.g., six in the case of cubical grid cells).

Many interesting scalar fields contain large regions in which the value of the field does not change greatly, and other regions where rapid change is concentrated. Such scalar fields can be stored relatively compactly by multiresolution methods. For example, octree subdivision consists of a hierarchy of cubes recursively divided into eight subcubes. In relatively constant regions of the scalar field, a large cube can serve to represent the entire region. If the scalar field is not constant within the region of a cube, then the cube is divided again. Another means to achieve a multiresolution subdivision of space is to define regions or blocks which each contain grids of a different resolution.

In contrast to the small number of parameters that are required to specify a regular grid, irregular grids generally require that each grid point be specified explicitly, since each grid point is placed without adherence to an overall regular structure. Such grids are commonly constructed of irregular tetrahedra. They can become quite complex in that grid cells may not all have the same number of neighbors. These grids could be constructed of elements with varying numbers of sides, although this is uncommon. Tetrahedral grids are often useful in computational fluid dynamics, since the faces of the tetrahedra can be aligned with the faces of solid surfaces impeding the flow of the fluid, reducing the error in the representation of such surfaces. It is relatively expensive to compute the path of a ray through an irregular grid; however, a small total number of properly placed irregular tetrahedra can often yield less approximation error than a much larger number of regular cubes.

In this thesis, we are primarily concerned with volume data represented on regular cartesian grids. These grids are commonplace, ray casting on them is efficient, and their simplicity makes them the obvious first subject of an investigation into the importance of memory hierarchy effects in ray casting volume rendering.

2.1.3 Physical models for volume datasets

Given an approximation of a three-dimensional scalar field in a computer's memory, there is not a single best representation of the data which will bestow maximum understanding upon a human observer. In an extreme example, the volume dataset could be dumped onto the screen as a long sequence of 0's and 1's. More usefully, we might extract some statistics from the dataset that summarize it in a table; for instance, in a fluid flow simulation, we might calculate the average velocity, maximum velocity, and the variance in the velocities of the fluid particles. However, in order to promote deeper understanding of complex three-dimensional data by scientists, we may exploit the human visual system by making an analogy to the physical world. We can model our three-dimensional dataset as a three-dimensional object with certain visual characteristics. We will choose a location in three space for the viewpoint, choose a view direction, and generate two-dimensional images of this object that properly obey the rules of perspective.

Much research in *ray tracing* [Kajiya84] has attempted to realistically model the physics of the transmission, reflection, and refraction of light, as shown in Figure 2.3. Ray tracing can reproduce an impressive array of visual and optical effects, including motion blur, depth of field, penumbras, translucency, and glossy reflections [Cook84]. However, the goal of scientific visualization is to allow scientists to understand their datasets, not to accurately model the physics of the transport of light. Complex reflective effects, for example, could cause features to appear to be where they are not present in the data. Therefore, most scientific volume visualization follows a simpler and more practically oriented physical model. Volume rendering for this purpose generally includes features that aid in the visual interpretation of the images as three-dimensional objects, and eschews others in exchange for clarity of interpretation, simplicity of implementation, and higher performance.

There are two common types of physical models for three-dimensional datasets: 1) *isosurface extraction* [Lorensen87] models, which reduce the volume data to a set of discretized surfaces approximating those at which the scalar field assumes a constant value, and 2) *direct volume rendering* models [Upson88, Sabella88, Drebin88], which treat the scalar field as a cloud of translucent material of varying opacity and color.

Isosurface extraction models reduce the volume data to a set of polygons defining surfaces at which the discretized scalar field assumes a constant value. These sets of polygons are commonly generated by an algorithm called "marching cubes" [Lorensen87]. The polygons are then typically illuminated by a light source and rendered as opaque or translucent objects, which may or may not cast shadows on each other. Self-shadowing is not generally required, since directional shading (shading

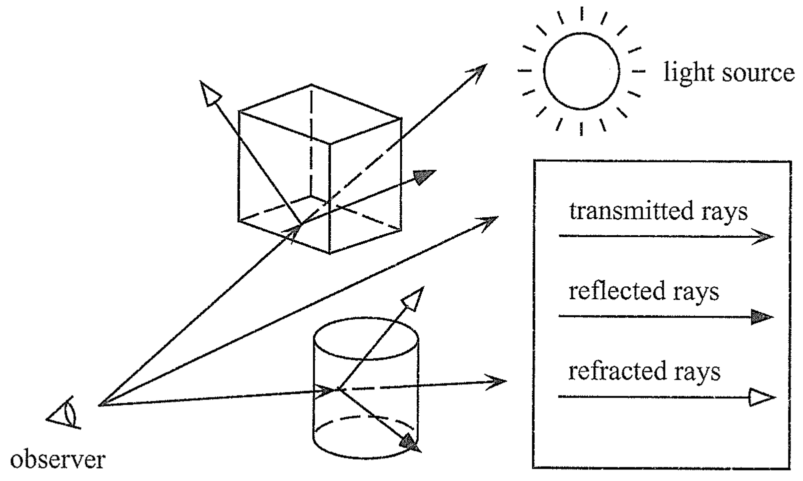


Figure 2.3: Transmission, reflection, and refraction in ray tracing

based on the surface normal of the polygons, and the direction of the light from the light source) provides sufficient visual cues. The human visual system does not effectively identify the "missing" self-shadowing. True self-shadowing is much more expensive computationally than directional-based shading.

The set of polyhedra representing the isosurface typically has much smaller storage requirements than the original dataset, and polygons can be rendered economically and quickly by specialized graphics hardware. The disadvantage of this approach is that the entire dataset is not represented in the resulting images, and that aliasing artifacts are common. For example, one may wish to extract from an MRI dataset the set of polygons lying on the surface of the bones. However, a single signal intensity does not define the bone surface (the signal varies due to bone density and due to artifacts in the MRI data). A common result is that some bones that are not actually connected may appear to be fused together. Since this method is a reduction of the dataset, information is lost, and such errors can occur.

Direct volume rendering algorithms model the data volume as a cloudy material of varying color and opacity. A mapping from the scalar voxel values to a color and opacity is defined. [Blinn82, Kajiya84, Max86] A simplified model of *light emittance* and *light transmission* is commonly used: the cloudy material both emits light and

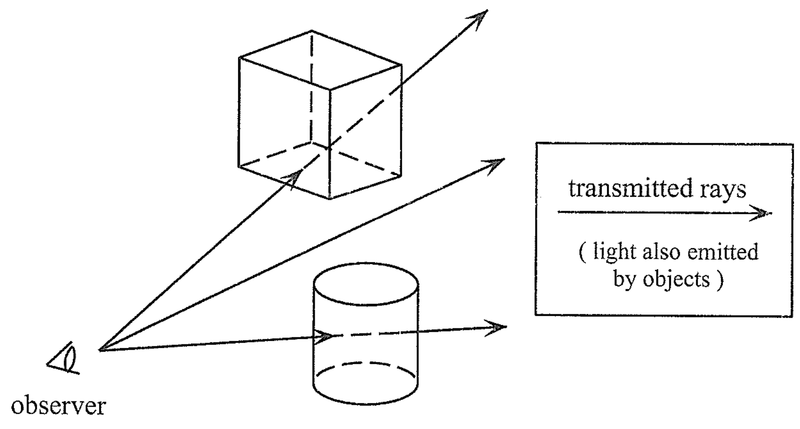


Figure 2.4: Transmission and emission in ray casting

partially occludes light passing through it [Porter84], as shown in Figure 2.4. Shadows from external light sources are generally omitted: true self-shadowing of the cloudy material is expensive, and no well-defined surface normals exist *a priori* to be used for inexpensive directional shading. (Surface reflections can be defined, at additional expense, using the gradient of the scalar field [Levoy88].) Scattering and reflection of light are omitted, since these properties can make clear understanding of the resulting images more difficult, and are in addition expensive.

2.1.4 Algorithms for direct volume rendering

There are several algorithms which can be used for direct volume rendering, i.e., used to produce results that accord with the physical model of a cloudy material of varying opacity and color. These include ray casting [Sabella88, Nieh92]; projection methods [Westover90, Shirley90, Laur91, Wilhelms91] in which individual volume elements are projected on to the image plane and composited together; and shear-warp methods, which decompose the view matrix into a series of shears and/or warps [Drebin88, Schröder91, Wittenbrink94, Lacroute94] and process slabs of the entire volume at once.

Ray casting

Figure 2.5 illustrates an algorithm known as *ray casting.* A ray originating at the eye strikes an image pixel on the display screen, and passes through objects in the data volume. The ray accumulates color and opacity depending on the voxels that it intersects. This is reverse of the direction that a light ray in our physical model would travel, but can be made to yield equivalent pixel values.

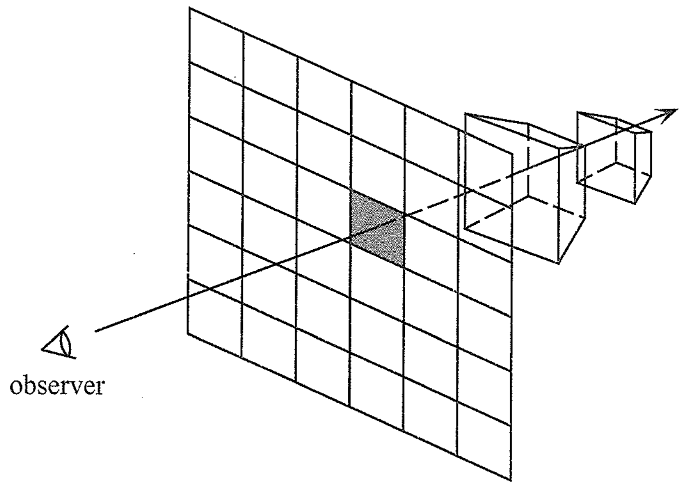


Figure 2.5: Rendering by ray casting or "image order" rendering

Models developed for the realistic simulation of light transport [Blinn82, Kajiya84, Max86] were discretized [Sabella88], simplified, and combined with the concept of the opacity channel [Porter84], to yield discrete equations for the accumulation of color and opacity by a ray passing through voxels of varying color and opacity.

Each ray has an associated four-vector of red, green, blue, and opacity, which is initialized to the zero vector. A ray intersects an ordered set of voxels V_1 to V_n; this set of voxels depends on the manner in which the ray intersects the dataset. As the ray passes through each voxel, it accumulates color and opacity from it, depending on the color and opacity of the voxel, and the length of the intersection between the ray and the voxel. Opacity is accumulated in the following manner:

$$ray_k^\alpha = ray_{k-1}^\alpha + (1 - ray_{k-1}^\alpha) * V_k^\alpha * L(V_k)$$

where ray_k^α is the total opacity accumulated for the ray. for voxels 1 through k. The voxels which comprise this path depend on the eye point and the direction of the ray. V_k^α is the opacity of voxel V_k; and $L(V_k)$ is a function of length of the line segment formed by the intersection of the ray with the voxel.[1]

Color is accumulated in a similar way:

$$ray_k^R = ray_{k-1}^R + (1 - ray_{k-1}^\alpha) * V_k^R * L(V_k)$$
$$ray_k^G = ray_{k-1}^G + (1 - ray_{k-1}^\alpha) * V_k^G * L(V_k)$$
$$ray_k^B = ray_{k-1}^B + (1 - ray_{k-1}^\alpha) * V_k^B * L(V_k)$$

where ray_k^R, ray_k^G, and ray_k^B are the red, green, and blue color values accumulated for voxels 1 through k; and V_k^R, V_k^G, and V_k^B are the color values of voxel V_k.

If we measure opacity on a scale of zero to one, the opacity of the ray asymptotically approaches one (1). As the ray becomes completely opaque, each subsequent voxel makes a smaller contribution to the color and opacity of the ray. After a ray has passed completely through the volume, the effect of a background color may also be accumulated. The final color is assigned to the image pixel associated with the ray.

In vector form, the four equations for *composition* of voxels, above, can be stated as:

$$ray_k^{\alpha,R,G,B} = ray_{k-1}^{\alpha,R,G,B} + (1 - ray_{k-1}^\alpha) * V_k^{\alpha,R,G,B} * L(V_k)$$

Researchers found that manipulation of the opacity channel could be used to antialias the images, and to delineate complex relationships among surfaces and interior structures. [Upson88, Drebin88]

[1]An exponential function of the length is more accurate to the canonical light transport integral, but a linear approximation to this is commonly used for steps of small length.

Composition is associative but not commutative

We have some flexibility in the order in which we composite the voxels. This is the property that will allow us to manipulate the order of memory accesses to influence access locality. We define the *composition operator*, C, as:

$$ray_k^{\alpha,R,G,B} = C(ray_{k-1}^{\alpha,R,G,B}, V_{k-1}^{\alpha,R,G,B})$$

This operator is used to take the value of the ray at step k, and accumulate the value of the kth voxel upon it. Note that the composition operator is associative but not commutative. That is:

$$C(C(a,b),c) = C(a, C(b,c))$$

but, in general:

$$C(a,b) \neq C(b,a))$$

Therefore, we need not composite an ordered set of voxels V_1 through V_n strictly in order. We can composite any contiguous subset of voxels without regard to whether neighboring contiguous subsets have yet been completed.[2] The results from these subsets can then be composited together to generate the results for larger contiguous subsets, until ultimately the entire set of voxels V_1 through V_n has been included.

It is this flexibility in the order of composition that will allow us to manipulate the order of memory accesses, to increase memory locality. This flexibility also makes *projection methods* possible:

[2]An assumption made here is that addition is associative. This is only approximately true in general with digital representations of floating point numbers. However, the color and opacity values remain within a small range of values (generally 0.0 to 1.0), and the error due to this approximation is negligible in practice.

Projection methods

In contrast to ray casting, which considers each pixel and calculates the data voxels it intersects, *projection methods* [Westover90, Shirley90, Laur91, Wilhelms91] consider each data element in turn, and calculate the element's contribution to the image.

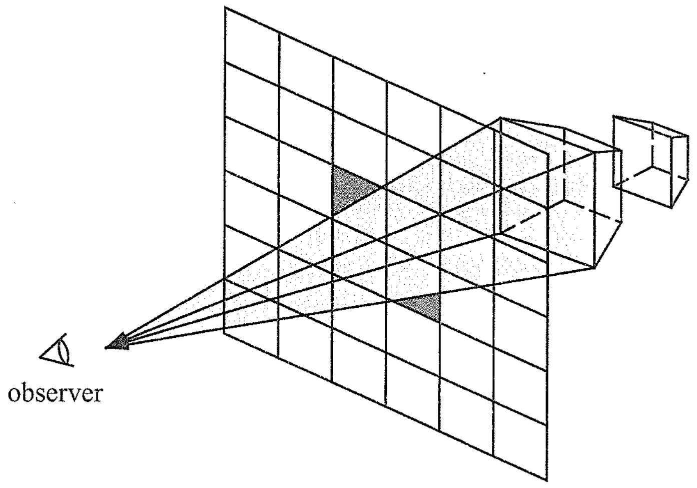


Figure 2.6: Rendering by projection or "object order" rendering

These methods are more commonly used with irregular grids, for which it is relatively expensive to do ray traversal. They are also used to render sets of three-dimensional objects that do not fill space. As shown in Figure 2.6, the objects (or irregular voxels) in the dataset are sorted in order of their distance from the eye point. Some projection operator P is applied to each object to determine its projected image on the pixels of the image plane. These two-dimensional projected images are composited together to form the final image. The same composition equations stated in the previous section are used.

Shear-warp methods [Drebin88, Schröder91, Wittenbrink94, Lacroute94] decompose the view matrix into a a sequence of computationally inexpensive shearing and/or warping operations. The operations are applied to an entire slab of data at a time, determining the projected image of the slab onto the image plane. The subimages are then composited together. This type of algorithm can be considered an optimized projection method.

Volume rendering algorithms used in this thesis

In this thesis, we will primarily use ray casting methods. However, for considerations of cache locality and parallel partitioning, we will also divide the regular dataset into blocks. We will treat these blocks semantically as objects to which we apply some projection operator P to determine their projected image, which are then composited together, as in projection methods. At the level of the individual block, the operator P which we use is ray casting. We will exploit the flexibility afforded by the associativity of the composition operator to manipulate the order of memory accesses, thereby increasing memory access locality and memory hierarchy performance.

2.1.5 Algorithmic optimizations to ray casting

Since this work focuses on ray casting for direct volume rendering, we discuss some algorithmic optimizations that are commonly made to the ray casting algorithm. Algorithmic optimization seeks to obtain higher performance, while accepting certain rendering artifacts in return.

Coherence in volume rendering

Many algorithmic optimizations to ray casting take advantage of some form of *coherence* [Sutherland74], for example:

- Object coherence

  That objects tend to be connected, bounded bodies.

- Area coherence

  That the two-dimensional projection of a three-dimensional body tends to have a connected, bounded range.

- Frame coherence

  That one frame of an animated sequence is likely to resemble the previous frame.

(e.g., in the colors assigned to given pixels; in the costs to render given units of work.)

- Coherence (locality) of memory reference
  That a memory reference to a given location is likely be temporally close to references to logically neighboring locations.

Many optimizations to ray casting take advantage of some form of coherence:

Bounding boxes

Object coherence is often exploited by placing bounding boxes around the non-empty regions of the dataset. [Kay86, Glassner84, Levoy90] These "boxes" need not necessarily be rectangular solids. Only rays which intersect non-empty volumes need to be processed. If a ray exits such a volume, it can skip to where it next enters such a volume, ignoring the intermediate voxels. This optimization exploits object coherence and the performance improvement it yields is dataset dependent. This optimization need not result in any rendering artifacts.

Five-dimensional bounding boxes (3D spatial plus a 2D solid angle) are placed around sets of rays originating from nearby points and pointing in the similar directions in [Arvo87].

Adaptive pixel subsampling

Area coherence can be exploited by casting fewer rays where the image does not contain much fine detail. For example, initially, a sampling of rays distributed randomly around the image are cast. If the final accumulated values of neighboring rays differ by more than a certain threshold, then more rays are cast in their neighborhood; otherwise, a ray is used to color more than a single pixel. A common artifact is for small objects to be missed entirely.

Data blocking

A primary focus of this thesis is improving memory hierarchy performance. One means to achieve this on at the level of the processor cache is to subdivide a large dataset into blocks that are stored contiguously in memory. If these blocks are sized appropriately relative to the size of the cache, and are processed serially, then both cache locality, and overall performance, can be greatly increased. [Palmer97a] This is a primary focus of this thesis, and these concepts are discussed in greater depth in section 2.4.

Opacity clipping

A ray's accumulated opacity value will asymptotically approach 1 at a rate that depends on the opacity of the intersected data voxels. A common algorithmic optimization known as *opacity clipping* [Levoy90] stops the further advancement of rays that have become nearly opaque. This optimization is quite dataset dependent: it works best on datasets with large opaque areas, and not at all on completely translucent datasets where rays never become completely opaque. A two- to four-fold performance improvement is possible in practice on many interesting datasets.

Re-use of information between frames

When the change in viewpoint between frames is small, temporal coherence results. This can be exploited to reduce redundant computation [Badt88]; in this thesis and in [Palmer94a, Palmer94b, Palmer95, Palmer97a, Palmer97b], we rely on this coherence in order to use workload estimates from one frame to predict workloads for the next frame.

Variable pixel density

In ray casting, there is a near-linear relationship between the number of pixels rendered (i.e., the resolution of the image) and the time to render a frame. [3] By the

[3]This relationship is true if we neglect cache effects, which are discussed in detail below.

technique of dynamic resolution reduction, it is possible to automatically choose which of these is most important to the user. One pattern of resolution reduction is shown in Figure 2.7. In the center of the screen, we maintain full resolution. Around this central area, we place concentric bands of reduced resolution. When the user stops moving, we refine the image in several steps until the entire screen is at full resolution, as shown from left to right in the figure. With this technique, we obtained [Palmer95] a typical four-fold reduction in total number of rays cast, and a corresponding four-fold frame rate increase, when it is most needed during user movement. This perhaps exploits coherence of user interest: the user will tend to place the region of the data in which they are currently most interested in the center of the screen.

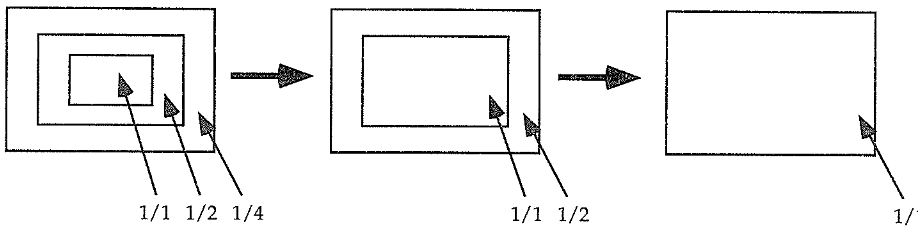


Figure 2.7: Dynamic resolution refinement for higher frame rate

Dynamic subsampling of the dataset by distance from viewpoint

Due to perspective, data voxels far from the eye appear smaller than those that are very close. At a certain distance, the projected area of a voxel is smaller than a single image pixel. When a ray reaches this distance, a more aggressively band-filtered dataset (which can be represented accurately with a smaller number of discrete samples) may be substituted for the original dataset with little visible effect on the image. [Laur91, Danskin92] As the ray travels further, still lower resolution datasets may be substituted. This has two effects: it decreases the total number of ray-voxel intersections which must be made, and improves cache hit rates if intersections are calculated in an appropriate order. It exploits the coherent and constant divergence of rays in the perspective projection.

Types of coherence exploited in this thesis

In some of the experiments described in later chapters, we use the algorithmic optimizations of opacity clipping, which exploits ray coherence; and bounding boxes, which exploits object coherence. For the purposes of load balancing, we exploit frame coherence, using load measurements for one frame to predict the likely load for the next frame. Most importantly, we focus on how to effectively maximize coherence (locality) of memory access, in order to tap significant gains in memory hierarchy performance.

2.2 Parallel architectures for ray casting

A number of researchers have constructed specialized hardware for volume rendering. A notable example is the PixelPlanes architectures [Fuchs89], however, this hardware was never distributed commercially. The first version of this architecture appeared as early as 1981. [Cabral94] used the texture mapping hardware of the Silicon Graphics RealityEngine graphics subsystem to do volume rendering (a task for which it was not explicitly designed). This implementation was quite fast (10Hz) for small volumes, but volumes were limited to the size of the memory in the graphics subsystem (16MB). This thesis is primarily concerned with algorithms that do not require a high fraction of the machine cost to be dedicated to special-purpose graphics hardware.

Volume rendering algorithms have been implemented on a wide variety of general-purpose architectures. Two early implementations on the Connection Machine CM-2 [Schröder91, Schröder92] achieved near-interactive rates for small volumes by sacrificing perspective projection. Both algorithms were specifically designed for the fine-grained SIMD architecture of the CM-2.

Early distributed-memory implementations were [Dippé84, Cleary86, Nemoto86]. [Corrie92] was the first distributed-memory implementation which used caching of volume blocks among the nodes. A large number of other distributed-memory algorithms have been described.

[Challinger91] implemented both ray casting and projection algorithms on the shared-memory BBN TC2000. [Nieh92] was a high-performance implementation on the shared-memory Stanford DASH. [Cox93] took specific advantage of the shared bus with a snooping algorithm to optimize pixel merging.

This thesis will primarily concern itself with MIMD (both shared- and distributed-memory) architectures, rather than SIMD architectures, whose prominence has waned in recent years.

2.3 Parallel partitioning for ray casting

2.3.1 The two goals of parallel partitioning

Ray casting is replete with parallelism. However, a primary concern in both shared-memory and distributed-memory MIMD implementations is selection of a suitable algorithm for partitioning the work (and, for distributed-memory architectures, the data) on a parallel machine. Parallel partitioning and load balancing algorithms have traditionally been designed to satisfy two often conflicting goals:

- Equal balance of workload among processors
- Minimization of communication among distributed nodes

2.3.2 Algorithms for parallel partitioning and load balance

Finding such algorithms suitable for ray casting volume rendering is nontrivial. There exists a wide variety of such algorithms, which can be classified by partition type and by the method used for load balancing.

Image partitioning versus object partitioning

In ray casting, work units can be created by subdividing the two-dimensional display screen, assigning sets of rays to each processor; or by subdividing the three-dimensional volume database, assigning sets of voxels to each processor.

Image partitioning [Challinger91, Nieh92, Corrie92, Whitman93, Palmer94b] methods partition the two-dimensional image plane, and assign subdomains (e.g., scanlines or rectangles) to the processors. The appeal of image partitioning is that the disposition of a pixel can be solely determined by the owning processor, without the composition of contributions from other processors. This allows global *opacity clipping*, the early termination of rays that become opaque before completely traversing the dataset. This algorithmic optimization yields performance gains which are dataset dependent but sometimes substantial. The primary disadvantage of image partitioning is lack of locality in accessing the three-dimensional volumetric data set. Since the relationship between image space and object space is not fixed, a processor may in general need to access any voxel in the dataset to cast a single ray. This lack of locality can result in cache thrashing in shared-memory systems. In distributed-memory systems, large communication volumes can result, unless the entire dataset is replicated on each distributed node. On fine-grain distributed-memory systems, the cost of either replication or communication can become prohibitive, and the image partition may not be suitable for use. (An exception is [Schröder92], who were able to reduce communication costs by using regular communication patterns on a data-parallel architecture.)

Object partitioning [Dippé84, Nemoto86, Cleary86, Kobayashi87, Priol89], an alternate approach, assigns subdomains of the object space called *blocks* (e.g., rectangular solid subsets of the data volume) to processors. In the *projection step*, processors cast rays through each block individually, generating a *tile*, the volume rendered image of a single block. The tiles are sorted according to their distance from the eye point, and composited together in the *composition step* to create the final image of the entire data volume.

Most importantly for the goals of this thesis, object partitioning schemes have a natural potential to improve local memory system performance. Blocks are stored contiguously in memory, so by processing an entire block at once, locality of memory access is dramatically increased. For properly sized blocks, orientation-dependent memory system delays all but vanish, offering the potential of reliable, interactive rendering rates.

One drawback of object partitions is that global opacity clipping is infeasible because blocks are rendered individually. Intra-block opacity clipping can still be done, but it is not as effective, particularly with large numbers of small blocks. In addition, there is overhead associated with the composition phase of the algorithm; this phase is parallelizable (the most straightforward method would use an image partition for the composition step), but has a cost proportional to the total area of the projected tiles (or proportional to the cube root of the number of blocks).

Static versus dynamic load balancing

Another important characteristic of parallel partitioning methods is how an equal balance of workload can be assigned among the parallel processors.

Static load balancing algorithms [Cleary86, Kobayashi88] make a fixed assignment of units of work to processors. Static load balance is simpler to implement than dynamic load balancing. These methods rely on a statistical tendency toward equal load balance due to a high granularity of work units, and a low variance in their cost (both among themselves and temporally). If these conditions are met, then a random assignment of work units to processors will tend to yield an equal load balance. For example, one can assign individual scan lines of an image to processors in round-robin fashion. Unfortunately, this yields the side effect of very low locality of memory reference. Increasing locality among assigned work units tends to increase the correlation between their costs, causing poor load balance. Furthermore, the dynamic nature of interactive volume rendering tends to yield a high variance in the cost of work units. For image partitions, regions of the screen may have high cost, or have no cost if we are able to cheaply determine that they do not overlap the projected image of the dataset; for an object partition, subvolumes of the dataset may have a high cost, or may not currently be visible on the screen, and have no cost.

Dynamic load balancing algorithms [Dippé84, Nemoto86, Nieh92] change the assignments of work units to processors when necessary. This may be when load balance falls below a certain threshold, or when the expected improvement in load balance is worth any communication costs that will be entailed. Dynamic load balancing algorithms often rely on *frame coherence*, the tendency of one frame (and one set of costs for the units of work) to be similar to the previous frame. Under the constraint of smooth movement of the view point and view direction, the temporal variance in the cost of a given unit of work remains low. This allows continuous adjustment of load balance through a gradual change in load conditions. Setting the required minimum load balance threshold too high can result in *load balance thrashing*, in which work units are shuttled back and forth among processors, without a payoff in improved load balance.

2.4 Memory hierarchies and exploiting locality of memory access

2.4.1 Uniprocessor cache architecture

Fast memory is generally more expensive than slow memory. Fortunately, typical access patterns to memory exhibit the property of locality: A memory reference to a given location will tend to be temporally near references to neighboring locations. Therefore, a hierarchy of increasingly small amounts of increasingly faster types of memory can be constructed, yielding higher cost effectiveness than a single layer of any one kind of memory.

A typical memory hierarchy for a single-processor architecture, shown in Figure 2.8, has one or two layers of fast cache memory between the processor and the main memory, called the level one (L1) and level two (L2) caches. The L1 cache is smaller and faster than the L2 cache. The caches are divided logically into *cache lines*. When the processor first requests a word from memory, the entire line containing that word is copied into the L2 cache, then into the L1 cache, then into the processor registers. The next time the processor needs a word from that line, the request can be satisfied from fast cache memory without the necessity of going to main memory. Likewise, if the processor needs to write to a word of memory, the write will

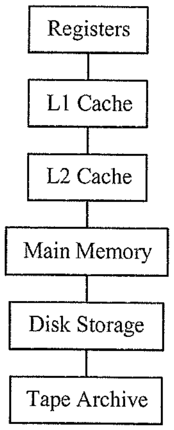


Figure 2.8: A typical memory hierarchy for a single-processor architecture

be faster if the cache line containing that word is already in the cache. The cache memory is much smaller than the main memory, so it will eventually become full, and some lines will have to be discarded. A common policy is to discard those which were least recently accessed. When a line is discarded from the cache, its contents are written back to memory if the line is "dirty" — if it has been written to.

Carefully written programs can be made to have increased locality of memory access, yielding better cache performance at each level of the cache; however, the interaction between a given program, other programs, the operating system, and multilevel cache hardware, is often complex. User programs generally do not have control over which blocks are maintained in the cache, since the caching policies are usually built into the hardware. This does not necessarily result in the most efficient use of the cache. There are certainly cases in which the programmer has information about future memory accesses that is not available to the cache hardware.

Copies of a given word of data are replicated vertically in this simple single-processor memory hierarchy. Each copy of the word can be considered to be consuming precious system resources. Algorithm designers are accustomed to conserving operations and program size; efficient use of each copy of a word of data is no less important.

2.4.2 Shared memory

Many algorithms are structured such that more than one processor can simultaneously work contribute to a solution. A shared-memory design is one way to build a parallel machine.

Shared memory systems consist of multiple processors, each with their own registers and caches, connected by a bus to a shared main memory, as in Figure 2.9. A typical example is the Silicon Graphics Power Challenge.

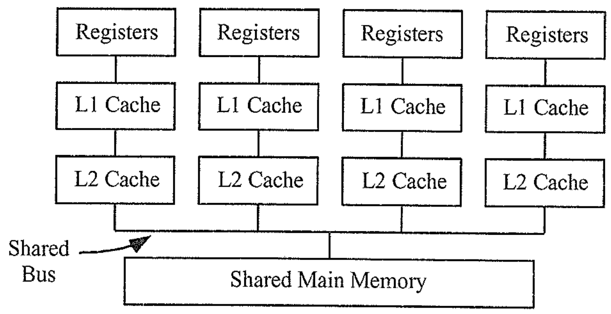


Figure 2.9: A typical memory hierarchy for a shared-memory architecture

In such a system, a given word of memory is replicated not only vertically, but can also be replicated horizontally, across the processors. This introduces the problem of maintaining *cache coherence* horizontally among the processors. For example, suppose two processors read the value "0" from the same location in memory into their caches, increment that value in their caches, and (when the cache lines are purged) each write the line back to the shared main memory. If special provision were not made to maintain cache coherence, then slight variations in the timing of these operations could result in nondeterministic results - equivalent to a single processor incrementing the value either once or twice. One hardware mechanism for assuring the property of cache coherence is referred to as *directory based*, in which a central directory of the sharing status of each cache line (whether a copy exists on more than one processor) is maintained; another is *bus-snooping*, in which each processor watches the bus for transactions involving cache lines of which it currently holds a copy.

The point-to-point bandwidth and latency of the bus shared by the processors is generally much higher than that of the interconnection network connecting processors of a distributed memory machine. However, this design is not scalable to as many processors as a distributed memory design, described below. It is possible to saturate the shared bus with requests so that *bus contention* among the processors limits the bandwidth available to each.

2.4.3 Distributed memory

Distributed memory systems consist of multiple processors, each with their own registers, cache, and local main memory; the processors are connected some interconnection network, as in Figure 2.10. A wide variety of interconnection topologies have been used.

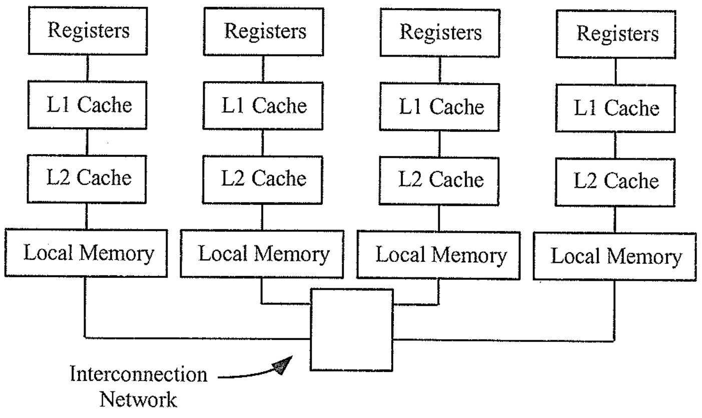


Figure 2.10: A typical memory hierarchy for a distributed-memory architecture

Many commercial distributed memory machines do not allow the programmer to access the entire memory of the machine as a single global address space. Each processor is able to read its own local memory; access to data stored in other processors' local memories must be transferred by the explicit passing of messages. On such machines, the programmer has complete control over how data moves between distributed nodes and over how what data, if any, is replicated.

On machines that do not provide a global shared address space, the programmer can implement one logically in software. This allows the use of information not available to an automatic caching hardware to optimize cache performance. We examine such a system, complete with provision for caching, in this thesis.

2.4.4 A hybrid architecture: the Power Challenge Array

The memory hierarchy of the Power Challenge Array is a hybrid: it consists of multiple shared-memory Power Challenge nodes connected by a HIPPI crossbar network, as shown in Figure 2.11. For certain problems, such non-uniform memory access (NUMA) architectures allow efficient scaling to greater numbers of parallel processors than either shared-memory machines (with their fast but non-scalable bus) or distributed memory machines (with their "scalable" but relatively slow interconnects). The deep memory hierarchy of such machines allows better use to be made of the coherence in some problems, in order to increase memory locality.

Our use of this architecture in our experimentation allows us to explore shared memory performance considerations, distributed memory considerations, and considerations for a relatively deep memory hierarchy, which requires particular attention to exploiting memory hierarchy to achieve the maximum available performance. It is quite difficult, both algorithmically, and in terms of implementation, to achieve the peak performance on such an architecture. This is one of the goals of this thesis.

2.5 The relationship between parallel partitioning and memory hierarchy performance

Most previous work in parallel volume rendering has sought to satisfy the two goals of parallel partitioning stated in section 2.3, which we restate here:

- Equal balance of workload among processors
- Minimization of communication among distributed nodes

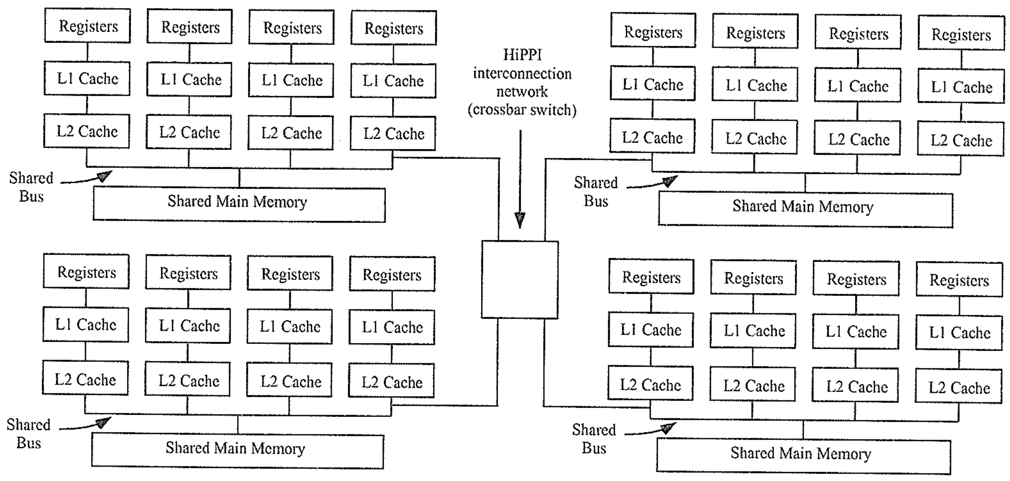


Figure 2.11: A hybrid distributed/shared memory hierarchy, the Power Challenge Array

However, past work has ignored the effects caused by a particular choice of parallel partition on the performance of lower levels of the memory hierarchy, e.g., the processor cache.

We will show in this thesis that choices made regarding parallel partitioning have fundamental implications to performance at all levels of the memory hierarchy — from the processor cache, to the level of the communication network between distributed nodes — which must not be ignored. Poor performance at any level of the memory hierarchy can nullify the performance benefits of equal load balance, and prevent effective parallel speedup.

We will amend the two goals of parallel partitioning to:

- Equal balance of workload between processors
- Maximization of locality of data access at every level of the memory hierarchy

The memory system on the individual nodes of a distributed-memory or hybrid architecture machine, along with the connection network between the nodes, should be considered as a single deep memory hierarchy. Algorithm designers face the difficult task of optimizing the memory system performance at every level of this hierarchy. Each replicated copy of a word should be considered to be consuming precious system resources; if a copy is not accessed repeatedly, the usefulness of its existence should be questioned. Such a view yields a clearer picture of the constraints that must be placed on parallel partitioning and load balancing to achieve maximum performance from the hardware investment.

Chapter 3 Experimental Goals, Methodology, and Tools

3.1 Experimental goals

As we have outlined in Chapter 2, many variants of volume rendering have been implemented on a wide variety of parallel architectures. This thesis focuses on memory hierarchy performance, something that we believe has not received due attention. Most parallel volume rendering research has focused on load balance, and minimizing communication at high levels of the memory hierarchy, i.e., the communications network between distributed nodes. This implies a simplistic and inadequate model of the memory hierarchy of the machine, since poor efficiency at any level of the memory hierarchy can have a strong impact on performance. This impact can be as important as that caused by poor load balance (which has been the gremlin preoccupying many researchers). Memory hierarchy efficiency at all levels has not received the attention merited by its performance impact.

Obtaining optimal memory hierarchy performance on parallel machines is more difficult than on uniprocessor machines because parallel machines often have deeper and more complex memory hierarchies, and because of the fundamental relationship between selection of a parallel partition and memory hierarchy performance. Choices made about distribution of data and work units dictate memory access and communication patterns. Optimization of memory hierarchy performance at several levels constrains the choice of parallel partition.

It is our goal to identify the constraints required to optimize exploitation of each level of a deep memory hierarchy, on both shared- and distributed-memory architectures; and to define parallel partitioning and load balancing algorithms which maintain good load balance while efficiently using the memory hierarchy.

3.2 Experimental methodology

Our approach proceeds from the bottom up. We study memory hierarchy performance at lower levels first, on a single processor, in Chapter 4. Our focus is level one (L1) and level two (L2) cache misses. We employ a test suite (consisting of views of the dataset from a sequence of viewpoints) designed to discover the directional dependence of L1 and L2 misses. We use three tools in this exploration: 1) algorithmic modifications designed to isolate the time attributable to cache misses, 2) a hardware performance monitoring board, which counts L2 misses on the bus, and 3) a cache simulation library, which simulates the entire state of the L1 and L2 caches, and allows detailed examination of the causes of cache misses.

In Chapter 5, we progress to memory hierarchy effects on a shared-memory multiprocessor (the Silicon Graphics Power Challenge). The same test suite is used, since our goal remains the characterization of L1 and L2 misses. Two other subjects enter the mix: first, the issue of processor contention for the shared bus, and second, the selection of a suitable parallel partitioning and load balancing algorithm. We will explore two such algorithms, one image partition and one object partition.

In Chapter 6, we move to explorations on the Silicon Graphics Power Challenge Array, which is constructed of distributed Power Challenge nodes networked together. No shared-memory model between the nodes is provided in system software, so we describe our system for replicating and caching blocks of volume data among the nodes, based on a message-passing layer. Our focus becomes the tradeoff between increased memory use (replication of data) and increased communication. We adopt a new test suite which is designed to exercise our software block caching scheme, proving that the scheme results in a very low penalty for no replication of the data, relative to the case of complete replication of the data on each node. That is, the system makes very efficient use of the memory hierarchy.

We will conclude the thesis with a discussion of the implications to algorithms that share certain characteristics with ray casting, and the implications to parallel architecture design.

3.3 Hardware platform: the Silicon Graphics Power Challenge Array

We conduct our investigations on the Silicon Graphics Power Challenge, and Power Challenge Array. The Power Challenge, a shared-memory multiprocessor, contains up to 18 MIPS R8000 CPUs clocked at 90 MHz. The processors communicate via a bus with a peak bandwidth of 1.2 GB/sec to a shared memory of up to 4 GB of RAM. The R8000 is a superscalar RISC processor, capable of 2 integer and 2 floating point operations per cycle. The R8000 has a two-level cache system: the on-chip, L1 cache consists of 512 blocks of 32 bytes for a total of 16K, and is direct mapped; the L2 cache consists of 32768 blocks of 128 bytes for a total of 4MB, and is four-way interleaved. The function of the two levels of the cache system is partitioned by data type: for integer numbers, the L2 cache serves as a streaming cache for the L1 cache; for floating point numbers, L2 serves as a single-level cache between the processor and memory. We access our data voxels as 1-byte integer values, so they receive the benefits of a two-level cache.

Up to eight Power Challenge nodes can be connected together by a HIPPI network to form a Power Challenge Array. On this system, there is no explicit sharing of memory between nodes of the array, nor is a logical global address space provided in software. The user is left to employ the message passing system of his or her choice to accomplish explicit communications between the nodes. The TCP/IP prototcol is provided over HIPPI. Several message passing libraries for TCP/IP exist which can be optimized for use over HIPPI, including MPI and SCPlib [Watts97].

The HIPPI hardware consists of a pair of boards per node: one to send and one to receive. Each of the two boards is connected by a HIPPI cable to a full crossbar HIPPI switch. HIPPI has a high peak bandwidth: 91 MB/sec using the TCP/IP protocol is cited by Silicon Graphics [Skibo95]. However, HIPPI is unable to maintain this high rate except under ideal conditions.

We have matched the SGI measurement of a peak rate of 91MB/sec using TCP/IP, in an experiment measuring the time for a large send operation (100 times as large as the 1MB buffer) to return. This measurement was made on a single unloaded machine sending to another unloaded machine over an unloaded HIPPI connection, where the message buffers were page aligned, and the message length was an integer multiple of the page size (16384 bytes). However, when one of the machines or the HIPPI link is slightly loaded, or the messages are not an integer multiple of the page size, we have observed that the practical bandwidth quickly decreases. If the messages are relatively small, e.g., 100-200KB, and there is contention on the network — such as four nodes all attempting to send to each of their three neighbors simultaneously — we have observed that the effective bandwidth of the HIPPI network decreases to 9-10 MB/sec per node.

3.4 Experimental software platform:*RendAsunder*

Over the past three years, we have developed a flexible, high-performance parallel renderer based on ray casting, called *RendAsunder*. *RendAsunder* is the primary experimental tool used in this thesis. This code can employ a number of optional parallel partitions on a variety of configurations of the Power Challenge and Power Challenge Array. RendAsunder contains many features to increase its usefulness as an experimental tool for the investigation of ray casting performance characteristics at all levels of a deep memory hierarchy. It contains other user interface and display features that allowed it to be used as a demonstration of parallel supercomputing technology at the Silicon Graphics vendor booth at several conferences, including Supercomputing '94 and '95.

3.4.1 General features of RendAsunder

At the core of RendAsunder is a high-performance ray casting kernel which we optimized heavily for efficiency on the super-scalar R8000. The inner loop of the kernel advances a ray to the next voxel that it intersects, and adds the contribution of the voxel to the accumulated color and opacity of the ray. The ray advancement primarily involves integer operations and the color and opacity calculations primarily involve floating point operations. The R8000 contains two integer units and two floating point units which can work in parallel under suitable conditions. Careful algorithmic tuning keeps each of these units maximally busy, while allowing unrolling of loops to eliminate loop overhead.

Within a single Power Challenge node, RendAsunder can use either 1) an object partition, 2) an image partition derived from a dynamic recursive subdivision, which divides the screen into a number of tiles equal to the number of processors, or 3) a static tiling into a number of tiles larger than the number of processors, which are then taken in turn by the processors off of a work queue.

Among multiple Power Challenge nodes, RendAsunder can partition its work with an object partition or the recursive image partition. The volume dataset can be completely replicated on each node, or RendAsunder can employ its software block caching system, in order to save memory. In this system, blocks move dynamically among the nodes to maintain load balance. The blocks do not have an affinity to any particular processor; however, *RendAsunder* can be configured to guarantee that at least one copy of each block is present in the collective memory of the nodes at all times.

RendAsunder can employ one of three different message passing libraries: the Silicon Graphics implementation of MPI, the mpi_ch implementation of MPI, or Jerrell Watts' SCPlib message passing library [Watts97]. Each of these uses regular Unix sockets, and can operate over ethernet or HIPPI networking hardware.

3.4.2 RendAsunder as an experimental tool

RendAsunder has been given advanced instrumentation to facilitate experimental studies, including extensive performance logging capability, which records the performance of various components of computation and communication in detail.

RendAsunder has the capability of operating and collecting statistics from the "PF3" hardware bus-snooping board for the Power Challenge. This board counts bus events of several types, including L2 cache misses by passively observing bus traffic.

nodes x procs	dataset size	resolution	partition type	fps
8 x 8	584x1878x341 (357 MB)	400x300	image w/ replication	10.0
8 virtual x 8	1610x894x5192 (7.1 GB)	800x600	object w/ caching	0.48

Figure 3.1: Sample performance points for RendAsunder

In addition, RendAsunder employs a software cache simulation library, which can be configured to mimic the cache sizes and caching policies of the R8000. This library simulates the state of every cache line. This library allows the study of both L1 and L2 miss patterns in greater depth than is afforded by the bus-snooping board.

3.4.3 The high performance of RendAsunder

RendAsunder is capable of both high frame rates in the absolute sense, and of frame rates that are high relative to the task of rendering very large datasets at high resolution. Figure 3.1 shows two sample performance points. On an array of eight Power Challenges of eight processors each, RendAsunder attained up to 10 frames per second on average rendering a large 584x1878x341 (357 MB) dataset at 400x300 resolution; this frame rate is generally considered to be a minimum for true interactivity. On a virtual array of four nodes of sixteen processors each (logically arranged into eight nodes of eight processors, to avoid a 2GB per-process memory use limit), RendAsunder was able to render images of what is, to our knowledge, the largest single dataset ever rendered: 1610x894x5192 (7.1 GB). At the high resolution of 800x600 pixels on this very large dataset, RendAsunder attained 0.48 frames per second, i.e., 2.1 seconds per frame. (The reason eight virtual nodes were mapped on to the four physical nodes was due to an operating system per-process memory limit of 2 GB.)

3.4.4 RendAsunder as a demonstration vehicle for interactive volume rendering

RendAsunder has been presented as a demonstration of supercomputing technology in the Silicon Graphics (SGI) vendor booth of four conferences: Supercomputing '94 in Washington, D.C., Supercomputing '95 in San Diego, HPCN '95 in Milan, Italy, and Parallel Computational Fluid Dynamics '95 in Pasadena. The installation was equipped with a joystick and foot pedals, for a total of six degrees of freedom, to allow visitors to fly around the 357MB Visible Human Male dataset at interactive frame rates.

RendAsunder can send its output to a single display screen; or multiple screens on a single machine; or multiple screens on multiple machines, of a Power Challenge Array. At Supercomputing '94 and '95, RendAsunder was presented in conjunction with the Power Wall project of the Laboratory for Computational Science and Engineering at the University of Minnesota.

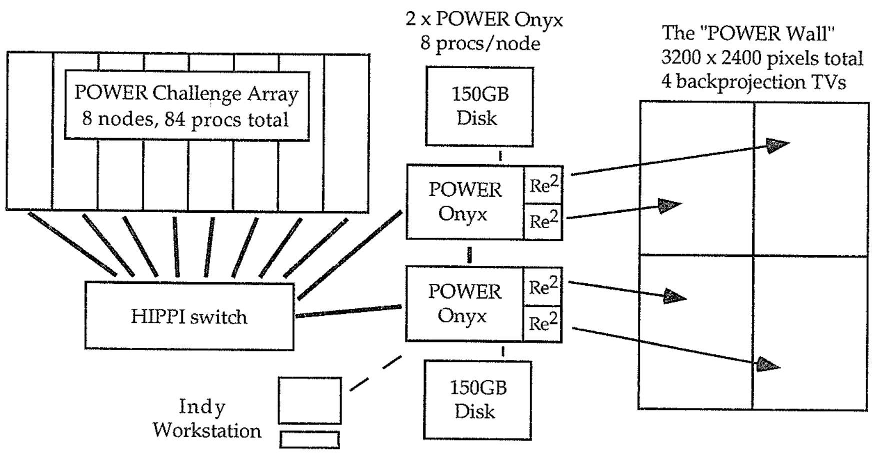


Figure 3.2: Hardware configuration at Supercomputing '95

Figure 3.2 shows the hardware configuration used: In the center are two SGI Power Onyx nodes; each contains eight R8000 processors, 2GB of shared memory, 300GB of disk, and two SGI Reality Engine^2 graphics boards. Each of the four Reality Engines drives a projection TV display at a resolution of 1600 by 1200 pixels. The four TV projectors are aligned to display one large image of 3200 by 2400 pixels. To the left of the figure, connected by two HIPPI links to the Onyx machines, is the Power Challenge Array; this consists of eight nodes, with 2GB of RAM per node, a total of 84 R8000 processors, and no graphics hardware. The Challenge Array nodes are connected to each other and to the Onyxes by HIPPI. An Indy workstation is used to run the graphical user interface. Control messages are transmitted by ethernet between the machines.

3.4.5 Summary of features of RendAsunder

- General features:
    - high-performance, software-pipeline ray casting kernel
    - configurable algorithmic optimizations, e.g., opacity clipping
- Features aiding experimentation:
    - extensive performance logging
    - provisions to collect data from bus-snooping board
    - built-in cache simulation library
- Parallel partitioning options:
    - Shared-memory parallel partitions
        * image partition with recursive bisection
        * image partition with many fixed tiles and work queue
        * object partition with many blocks and work queue
    - Distributed node parallel partitions
        * image partition with dataset replication
        * object partition for projection; image partition for composition, with dataset replication

        * object partition for projection; image partition for composition, with dynamic block caching

    - Software implemented global address space for volume blocks with provision for dynamic caching

- Choices for message passing:

    - mpi_ch version of MPI message passing library

    - SGI version of MPI message passing library

    - SCPlib [Watts97]

    - message passing over HIPPI or ethernet

- Piloting modes:

    - experimental test suites

    - interactive graphical user interface

    - joystick and footpedals interface

    - recorded flight path

- Choices of hardware configuration for display:

    - display to a local array node

    - display to an external node

    - display to multiple nodes

    - display to multiple screens on multiple nodes

- I/O:

    - input of uncompressed or run-length-encoded volume data

    - output of images in "ppm" format

- Compatible Hardware Platforms:

- One or more 64-bit R8000/R10000 single- or multi-processors
- One or more 32-bit R4400 single- or multi-processors

Chapter 4 One Processor

Figure 4.1 recalls our view of the memory hierarchy of a typical single-processor. This chapter is a detailed analysis of memory hierarchy effects for ray casting on a single processor - specifically the behavior of the L1 and L2 caches. We use three tools to characterize memory hierarchy performance: 1) experimental timings designed to isolate the time attributable to cache miss penalties; 2) a hardware bus-snooping board, which keeps a detailed log of bus traffic, including L2 cache misses; and 3) a configurable software cache miss simulator, which enables us to separate the total cache miss penalty into L1 and L2 miss portions.

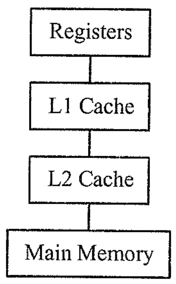


Figure 4.1: A typical memory hierarchy for a single-processor architecture

4.1 Ray casting cost analysis

Most of the time to render a frame is spent in the inner kernel of the ray caster, iteratively advancing a ray into the next voxel, reading the voxel's value, mapping it to an opacity and color, and adding the opacity and color appropriately to the accumulated opacity and color of the ray. This memory read of the voxel's value is the single memory access that causes nearly all of the memory hierarchy effects described below. The distribution of memory accesses has a great impact on performance. This depends largely on two parameters, the *number of blocks* into which the dataset is divided, and the *view direction*. These entail a certain distribution of memory accesses, and a corresponding number of L1 and L2 misses.

The total time per frame can be broken down into the following parts:

$$\begin{aligned} total\ time\ per\ frame(view\ direction, number\ of\ blocks, number\ of\ rays\ cast) &= \\ time\ per\ intersection(view\ direction, number\ of\ blocks) \times \\ number\ of\ intersections(view\ direction, number\ of\ rays\ cast) + \\ a \times number\ of\ blocks + \\ b \times number\ of\ rays\ cast + c \end{aligned} \quad (4.1)$$

where the *time per intersection* and *number of intersections* refer to the intersections of rays with voxels. a, b, and c are scalar factors associated with constant time costs per block, per ray, or per frame (e.g., loop overheads). The time on average to intersect a ray with a single voxel can be separated into:

$$\begin{aligned} time\ per\ intersection(view\ direction, number\ of\ blocks) &= \\ d * L1\ misses(view\ direction, number\ of\ blocks) + \\ e * L2\ misses(view\ direction, number\ of\ blocks) + f \end{aligned} \quad (4.2)$$

where d, e are constant factors reflecting the averaged behavior of complex cache interactions, and f is a constant factor.

We use a three-dimensional Bresenham algorithm to calculate the path of a ray. The Bresenham algorithm yields more advancement steps for diagonal rays than straight rays of the same length (i.e., it causes the *number of intersections* to be a function of *view direction*), and a division operation must be done to correct for the varying distance of each step, but this was the fastest of several ray advancement strategies we implemented, because we were able to achieve very good software pipelining on the superscalar R8000.

We do not consider optimizations to minimize the *number of intersections* in this work. Likewise, there are methods which exploit forms of coherence to reduce the total *number of rays cast* which we do not explore here (we cast one ray for each pixel). We focus on the tradeoff between decreasing the *time per intersection* due to increased locality from blocking of the data, and the resulting overhead from increasing the *number of blocks*.

The number of ray-voxel intersections in one frame varies with view direction due to several factors, including the dataset extent in each dimension, effects of diagonal rays in the Bresenham algorithm, and the portion of the dataset currently visible. Therefore a useful normalized measure of ray casting speed is the *average time per intersection*, i.e., the average time to advance a ray one step and intersect it with a single voxel.

We will show that most of this time for the slowest view directions is attributable to memory hierarchy effects, in the context of the extreme demands placed on the memory system by the casting of hundreds of thousands of rays through hundreds of millions of voxels.

4.2 Datasets

We use three datasets in our experiments: 1) the *female* dataset, derived from the Visible Human Female dataset [Ackerman95], which was produced by freezing a female cadaver and shaving it into over 5100 slices (in this chapter, we use this dataset at one-half this resolution), which were digitally photographed; 2) the *male* dataset, derived from the Visible Human Male dataset [Ackerman95], produced by shaving a male cadaver into 1900 slices; 3) a computational fluid dynamics *vorticity* dataset, courtesy of the Laboratory for Computational Science and Engineering at the University of Minnesota [Woodward95], which displays the vorticity of a simulated turbulent fluid. The dimensions of each dataset are given in Figure 4.2. Each was stored at 1 byte per voxel. Example images of the three datasets are shown in Figures 1.1 and 4.3. Note that since the female and male datasets were produced photographically, the colors are true to the actual colors of the bodily tissues represented.

Dataset Name	Dimensions	Total Size
vorticity	512x512x512	128 MB
male	584x1878x341	357 MB
female	840x2595x480	1 GB

Figure 4.2: Sizes of experimental datasets

Whereas the performance heuristic of opacity clipping will be shown to be useful for the female and male datasets, in that they both contain large opaque objects, the vorticity dataset is different since it is largely translucent. In addition, the vorticity dataset is cubical, which is useful for removing effects due to non-unity aspect ratios when comparing performance between views from different directions.

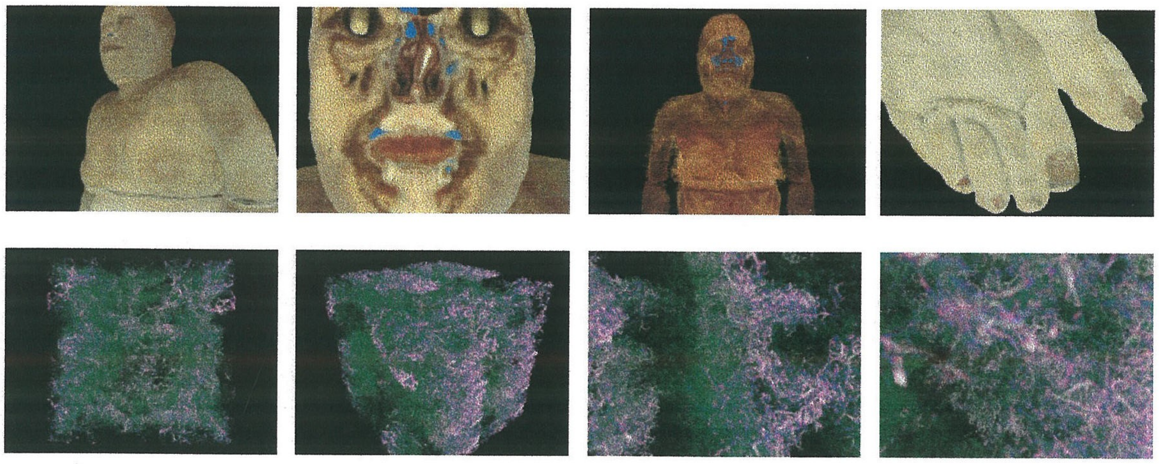


Figure 4.3: Example images of the female (top) and vorticity (bottom) datasets

4.3 The *axisorbit* test suite

The *axisorbit* test suite consists of a sequence of 132 views of the dataset. This suite is intended to uncover the directional dependence of the time per intersection and of the two types of cache miss. Selected frames from this suite, called the "axisorbit" suite, are shown in Figure 4.4. Note that the suite is broken into four discrete segments, each consisting of the rotation of the dataset about a different axis. In

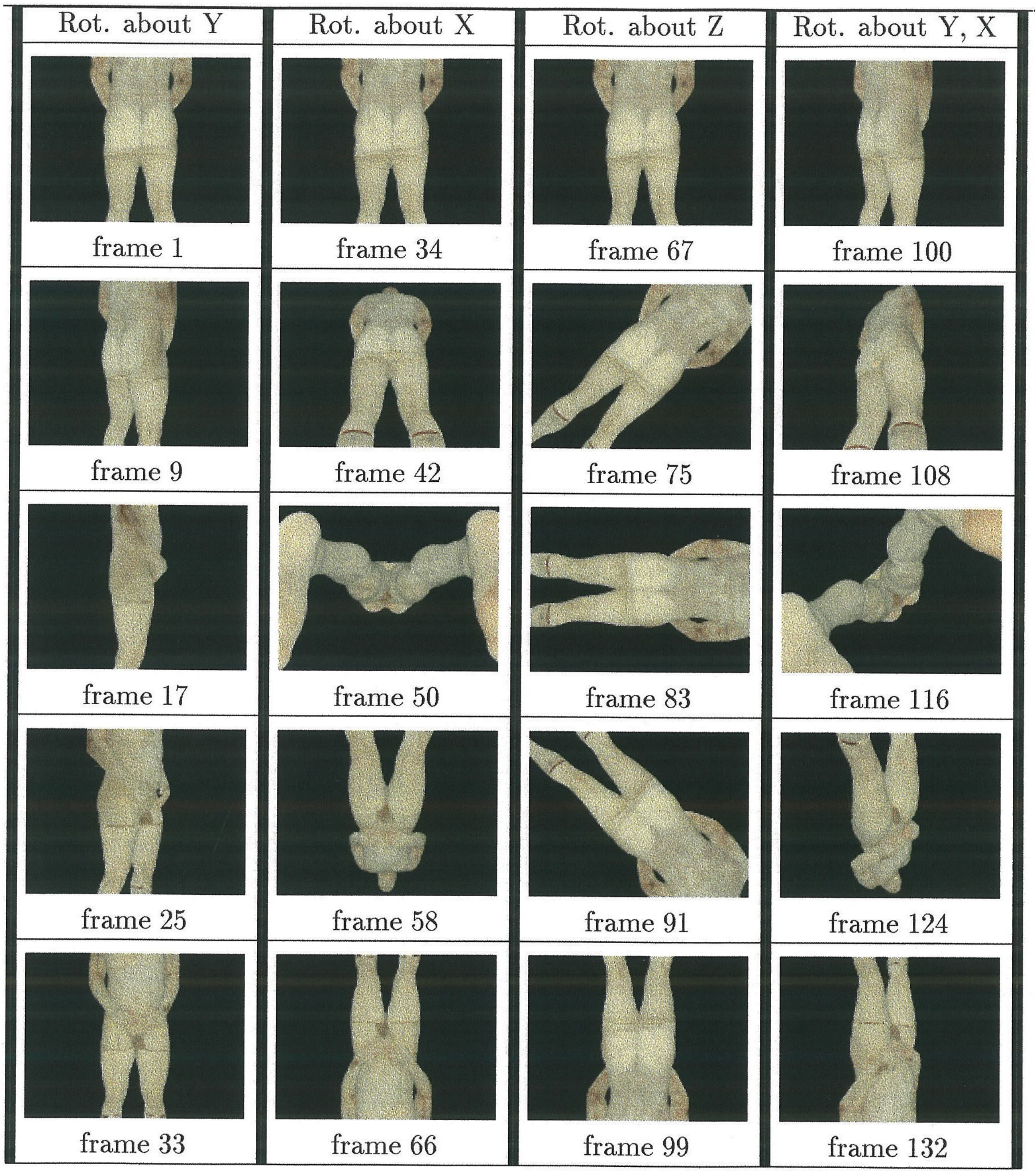


Figure 4.4: The *axisorbit* test suite

frames 1-33, the dataset is rotated 180 degrees about Y; in frames 34-66 about X; in frames 67-99 about Z; and in frames 100-132 the dataset is initially rotated by 45 degrees about Y, and then from 0 to 180 degrees about X. All views are rendered at a resolution of 400 by 300 pixels.

The three-dimensional datasets are linearized in memory by X first, then Y, then Z. When the cadaver in the female or male datasets is standing upright and facing the viewer, the object space X direction points to the viewer's right, the Y direction points up, and the Z direction is toward the viewer. This will be important to the consideration of cache effects, which we will find have a strong dependence on the view direction.

4.4 Experimental procedure

We conducted all experiments in this chapter on one processor. Each dataset was divided into multiple blocks, and each block was processed serially by a single processor. We assured ourselves that there would be no interblock effects that would prevent these results from carrying over to multiple processors by performing an experiment in which the cache was flushed between the processing of blocks - we found negligible interblock effects. This should be expected since blocks do not interleave in memory and so share at most one cache line.

4.5 View direction dependence of time per ray-voxel intersection

Figure 4.5 establishes the strong dependence of the time per ray-voxel intersection on view direction, and that this dependence can be controlled by blocking. We conducted all experiments in this section on one processor with opacity clipping turned off. The figure plots frame number versus time per intersection for datasets blocked to different degrees, between 1 and N blocks, where N is 1024 for the female dataset, 256 for the

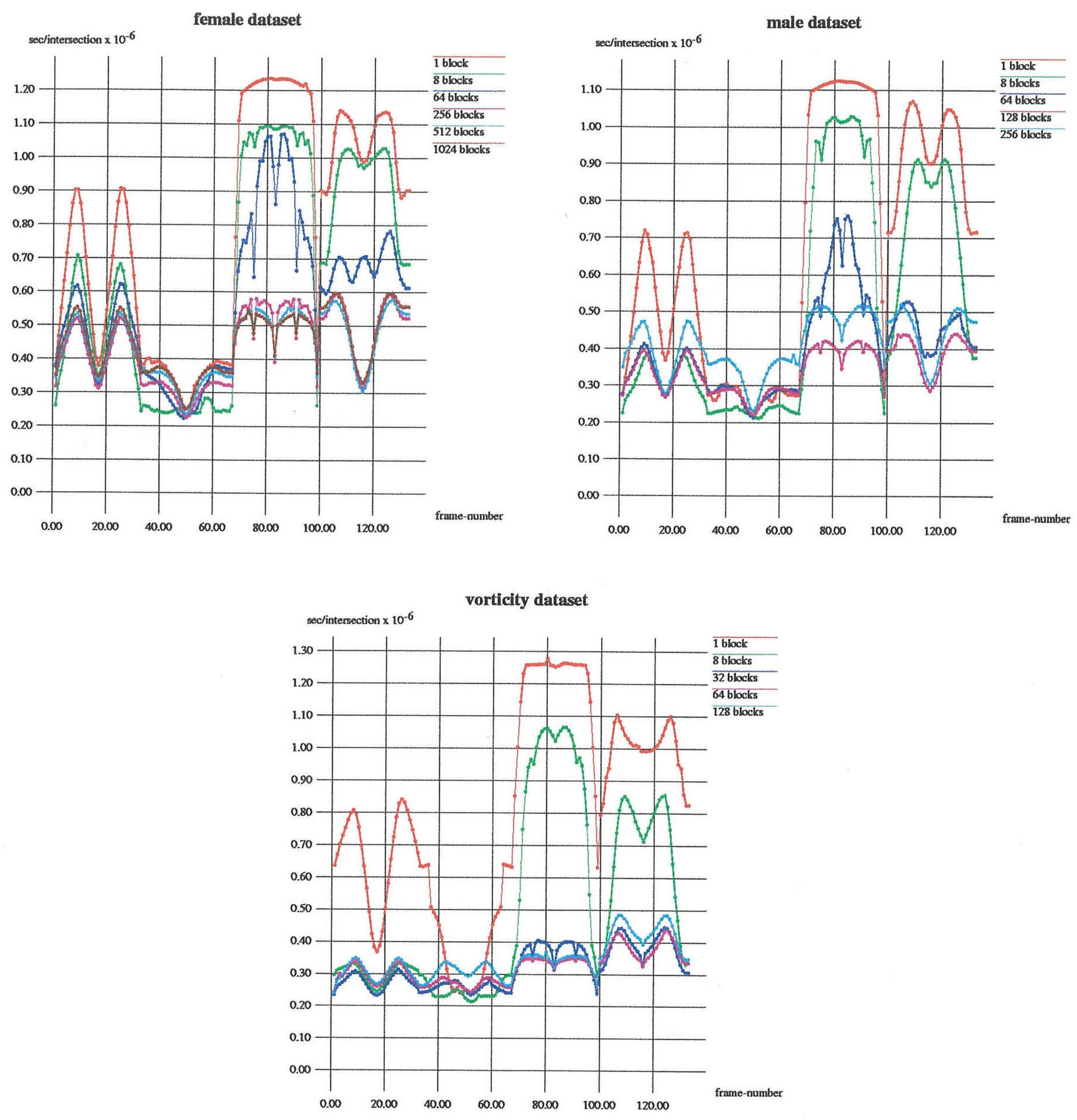


Figure 4.5: Dependence of time/intersection on view direction

male dataset, and 128 for the vorticity dataset. The maximum number of blocks N was chosen for each dataset to yield minimum sized blocks of approximately 1 MB, a fraction of the size of the 4 MB L2 cache (i.e., we selected the minimum-size block to be smaller than what we expected would be the optimal size).

The time for a single block (shown in red) shows strong directional dependence for all three datasets. Appropriately-sized blocks reduce this dependence: as the number of blocks increases, locality increases, and cache hit rates improve. However, making the blocks smaller past a certain point yields no benefit, and there is an overhead cost associated with increasing numbers of blocks. For the female dataset, the optimal number of blocks for a broad range of view directions is 512 (light blue, female dataset). For the male and vorticity datasets, respectively, the number is 128 (purple, male dataset), and 64 (purple, vorticity dataset). For the three datasets, these numbers yield block sizes of 2 MB, 2.8 MB, and 2 MB. for the three datasets, respectively. This size fits easily into the 4 MB L2 cache.

We determined experimentally that interblock effects are negligible by comparing the ordinary algorithm with one that completely flushes the cache between blocks. However, interpixel effects are quite significant; the specific effects manifested can be unintuitive. We consider a single large block (red in Figure 4.5). The data is linearized in memory by X first, then Y, then Z. One might naively assume that viewing directly down the X direction (e.g., frame 17 in Figure 4.4) would yield the best cache performance, and therefore the lowest time per intersection. This is not the case; instead, a view directly down the Y axis (frame 50) is better. This is due to somewhat complex interpixel effects:

Recall that the dataset is linearized in memory X first, then Y, then Z. Recall also that, as we cast rays, we scan across the display screen in horizontal rows. The R8000 has an L1 cache line length of 32 bytes and an L2 cache line length of 128 bytes.

In the case in which a ray travels directly down the X axis (frame 17 in Figure 4.4), it is intersecting voxels in steps of 1 byte through memory. It misses in the L1 cache only once in every 32 ray-voxel intersections, and misses in L2 only once in every 128 intersections. The first intersection that misses will fetch into L1 the values for the next 31 intersections, and into L2 the values for the next 127 intersections, yielding good cache performance. However, (in the approximation that all rays are traveling in parallel down the X axis) none of these cached voxels will be used by the next ray, which starts one pixel to the right of the last one on the display screen.

In contrast, when a ray is traveling directly down the Y axis (frame 50 in Figure 4.4), the first ray misses in cache at every ray-voxel intersection: it is traveling through memory in steps equal to the X dimension of the dataset, which is greater than both 32 and 128 bytes for all three datasets. However, it leaves behind in the cache an entire X-Y plane of voxel values in the L2 cache. The dimensions of this "plane" are the entire Y dimension of the dataset in one direction, by the length of a cache line in the other direction. (Since the L1 cache consists of 512 lines of 32 bytes, and the datasets each have a Y dimension of 512 or greater, it is likely that casting a single ray down the Y direction completely flushes the L1 cache for all three datasets. However, such a ray will not flush the L2 cache completely.) In the approximation that it travels straight down the Y axis, the next ray (one pixel to the right on the display screen) is translated from the last ray by 1 voxel in the X direction. It lies entirely in the same X-Y plane of voxels that was fetched by the last ray, so it does not miss L2 cache at all - and nor do the next 127 rays.[1] Note that frame 116 is just 45 degrees rotated from frame 50, but it gets far worse cache performance because rays that are adjacent on the screen do not share an X-Y plane of voxels.

A ray traveling directly down the Z axis will miss cache every time, but will leave an X-Z plane of voxel values behind in the L2 cache. There are two interesting cases in our test suite: 1) If the next ray cast is translated from the last ray by 1 voxel in the X direction (as in frames 1, 33, 34, 66, 67, 99), it will hit cache every time. This yields quite good performance - nearly as good as the best case of frame 50, above, for the female and male datasets. 2) However, if the next ray cast is translated from the last ray by one voxel in Y (as in frame 83), it will miss every time. Furthermore, by the time the ray one row down on the screen (which would intersect the same

[1]The question of why looking directly down the X axis is not at least equally good to looking directly down the Y axis has not been answered by this simple model; it will turn out to involve rays being cast in perspective projection. We discuss it further in section 4.8.

X-Z plane of voxels) is cast, the plane has already been flushed from the cache by the intervening cache activity. This is the worst possible cache performance: every ray misses at every intersection. However, note that the worst performance case of frame 83 could be turned into the very good performance case (near to the best case for the male and female datasets) of frame 1 simply by scanning across the display in columns instead of by rows.

The cache lines fetched into the cache by a ray traveling orthogonally down the X, Y, or Z axes are shown in Figure 4.6. Note that the cache lines always go in the X direction. A ray traveling directly down X experiences fewer cache misses, but leaves fewer lines behind in the cache where they might be reused by other rays; a ray traveling down the Y axis or the Z axis leaves a Y-X or a Z-X plane of voxels in the cache, respectively.

Analysis of the cache behavior for perspective-projection rays traveling in directions not parallel to the coordinate axes becomes so complex that a cache simulator must be used, as we do below in section 4.8.

In summary: Interframe and interblock pixel effects are negligible. Complex interpixel effects cause the dependence of ray-voxel intersection time on view direction. This directional dependence can be minimized with appropriate blocking of the data.

4.6 Isolation of time due to cache effects

In order to determine how much of the total frame time is due exclusively to memory hierarchy effects, rather than to other overheads, we modify the ray casting kernel, replacing the access to the currently intersected voxel with an access to a single, fixed voxel. This of course does not generate correct images of the dataset, but eliminates the costs associated with cache miss penalties. After the first access to this fixed voxel, subsequent accesses will result in no cache misses. We call the modified algorithm the "no-mem" algorithm. Figure 4.7 shows the result for a single block. The time per intersection of the original algorithm is labeled "with-mem" (red). The time per intersection of the modified algorithm is labeled "no-mem" (green); this is the time

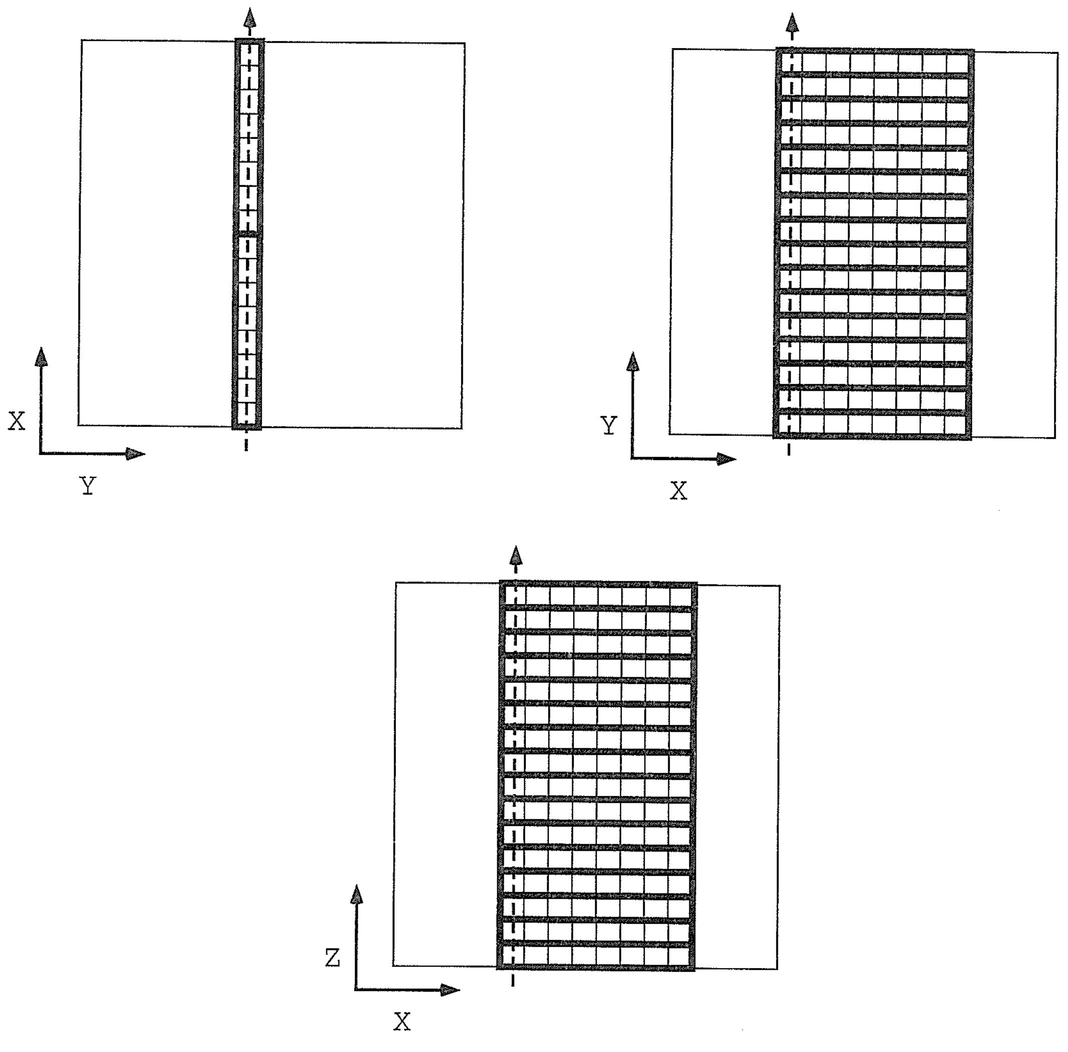


Figure 4.6: Cache lines fetched by a ray traveling down the X, Y, or Z axis

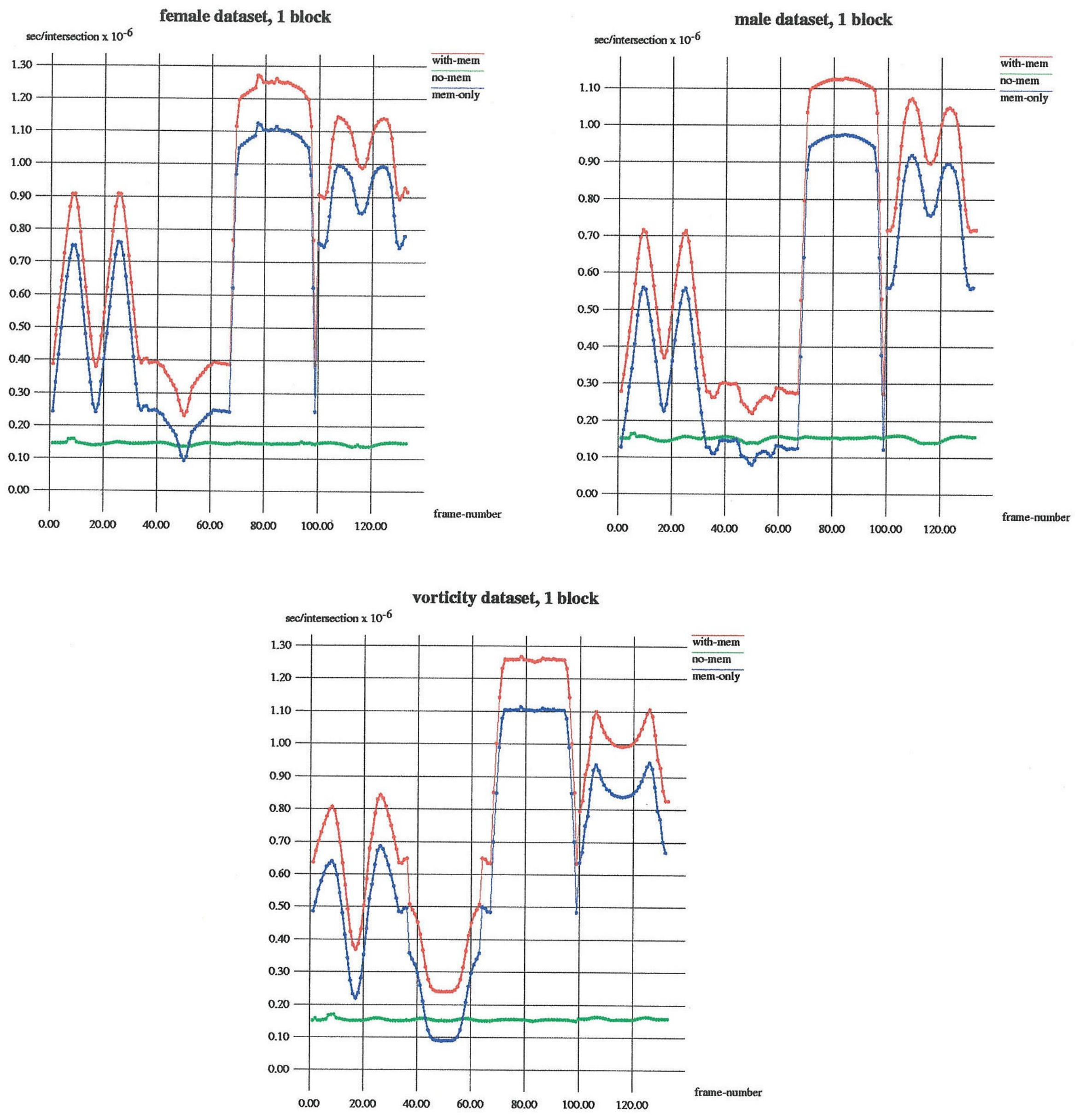
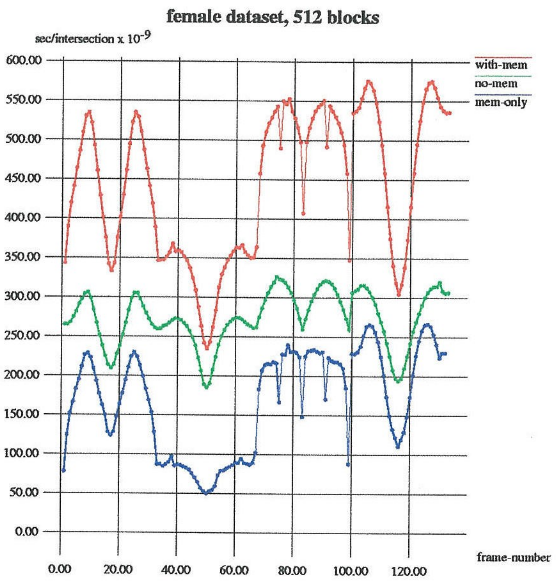
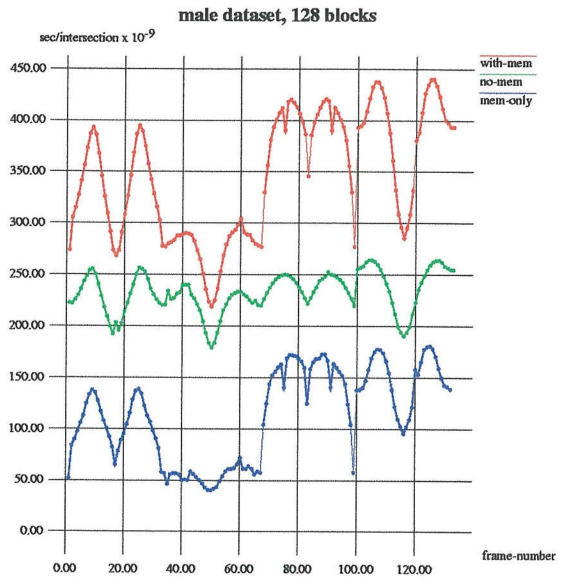


Figure 4.7: Time/intersection: with-memory, no-memory, and memory-only, 1 block





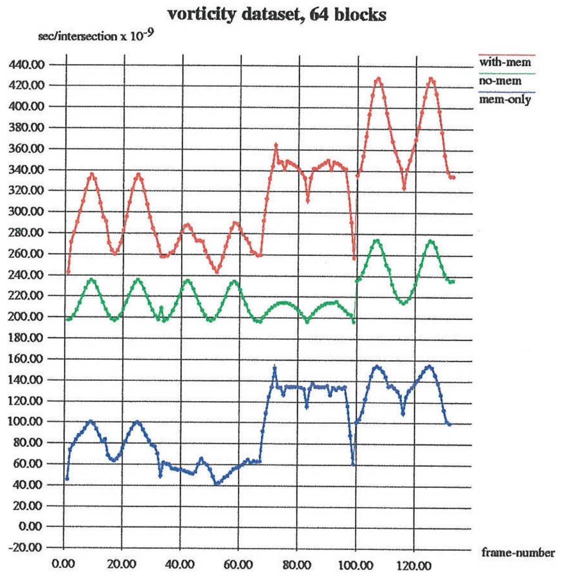


Figure 4.8: Time/intersection: with-memory, no-memory, and memory-only, optimal number of blocks for a given dataset

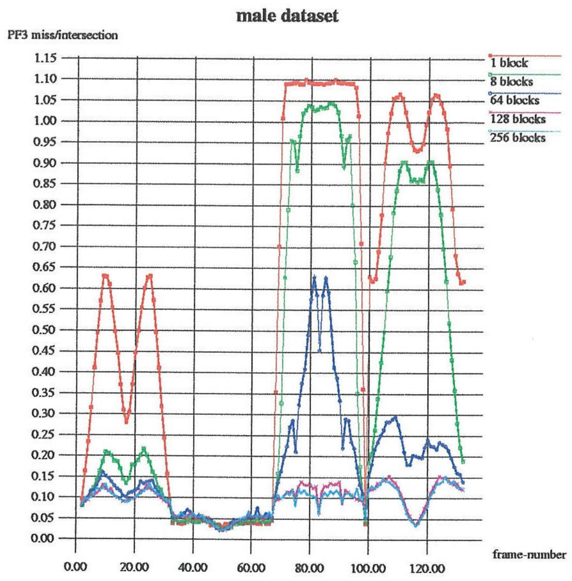
attributed to everything but memory hierarchy effects. The line labeled "mem-only" (blue) is the difference between the "with-mem" and the "no-mem" lines. This is the time per intersection which we attribute directly to memory hierarchy effects. The "mem-only" time per intersection is the dominant fraction of the total cost. With the optimal number of blocks for each dataset (not shown here), the "mem-only" time is actually reduced to slightly below the "no-mem" time for all viewpoints. In addition, it is strongly directionally dependent. We will next separate the time we attribute to cache miss penalties into L1 and L2 miss portions, and compare the total time attributable to cache misses to the "mem-only" measured time.

In Figure 4.8, the datasets are blocked into their optimal number of blocks (512, 128, and 64 for the female, male, and vorticity datasets, respectively). This reduces the cost attributable to memory hierarchy effects for all views, and greatly reduces the directional dependence of this cost, which is important for reliable interactive performance.

In summary: When a single block is used, memory system delays account for the vast majority of the directionally dependent cost associated with ray-voxel intersections. Appropriate blocking cuts this fraction to less than half of the total cost, for all view directions.

4.7 Hardware bus-snooping board

The hardware bus-snooping board passively monitors the bus, counting various bus events, including L2 cache misses (L1 misses do not directly generate bus events). The left side of Figure 4.9 shows hardware measurements of L2 misses per intersection for several different numbers of blocks for the male dataset. Note that the maximum number of misses per intersection (between frames 70 and 96) is up to ten percent higher than 1.0: there are slightly more L2 misses than intersections. This is because the bus-snooping board records not only misses from accesses to the data voxels, but those from all other sources. Blocking into 128 blocks reduces cache misses; division into 256 blocks does not yield further benefit.



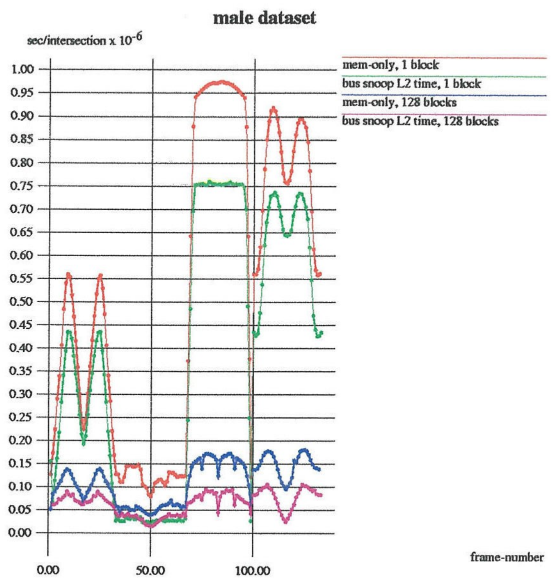


Figure 4.9: Left: bus-snoop L2 misses/intersection. Right: "mem-only" and "bus-snoop L2" time/intersection

On the right side of Figure 4.9, we compare the "mem-only" time for the male dataset for one and 128 blocks, to the time we can attribute to L2 misses measured by the bus-snooping board. The latter is derived by multiplying the number of L2 misses per intersection by an L2 miss penalty factor, to get time per intersection attributable to L2 misses. The commonly cited L2 miss penalty factor for the R8000 is 690ns. It turns out that L2 misses cannot account for all of the "mem-only" time. (If it could, the red and green lines would match, and the blue and purple lines would match.) Not only is the bus-snoop L2 time too low, it is too low by a non-constant factor: i.e., the 690ns penalty factor may be low, but that alone could not make up for the difference. The additional memory hierarchy cost that we have not included here is due to L1 misses, as identified in the next section.

In summary: Estimated time derived from actual L2 cache misses accounts for most, but not all, of the time we attribute to memory system delays.

4.8 Cache simulator

To investigate the behavior of the cache system, we designed a cache simulator which simulates the state of the R8000 L1 and L2 caches at the cache-line level. In our ray casting kernel, we replace the memory read of the current voxel with a call to the simulator. The call returns the proper voxel value, with the side effect of updating the state of the simulated caches and counting both L1 and L2 cache misses. We use the simulator to count only accesses to the data voxels, which cause the vast majority of all cache misses, ignoring all other memory accesses. The simulator can be configured to mimic the caching policies and cache sizes of the R8000: a direct-mapped L1 cache of 32 byte blocks totaling 16 KB, and a four-way interleaved L2 cache of 128 byte blocks, totaling 4 MB. We use a random block replacement policy.

Figure 4.10 shows the simulated counts of L1 and L2 misses per intersection, for the three datasets, for two block sizes each. Note that here, unlike the bus-snooping results on the left side of Figure 4.9, the maximum number of misses per voxel intersection is 1.0, since the simulation counts only cache misses due to accesses to the voxels in the dataset.

As we predicted in section 4.5, the L2 miss rate for 1 block (blue) is nearly 1.0 for frames 72 to 94 (when we are looking down the Z axis and adjacent rays do not share an X-Z plane); and for frames 105 to 127 (when we are looking down a skew axis except for frame 116); or for frame 116 (when we are looking down the Y axis and adjacent rays do not share and X-Y plane). Also as we predicted, L1 misses for one block (red) are low when we are looking directly down the X axis (frame 17) but are otherwise relatively high. The reason for worse L1 performance for the vorticity dataset than the other datasets over a wide range of views is likely to be due to the fact that the cubical vorticity dataset fills more of the screen than the other two datasets, so that more rays are at skew angles to the data due to perspective projection.

We wish to analyze more precisely those frames for which we are not looking directly down one of the major axes of the dataset. We define three more coordinate

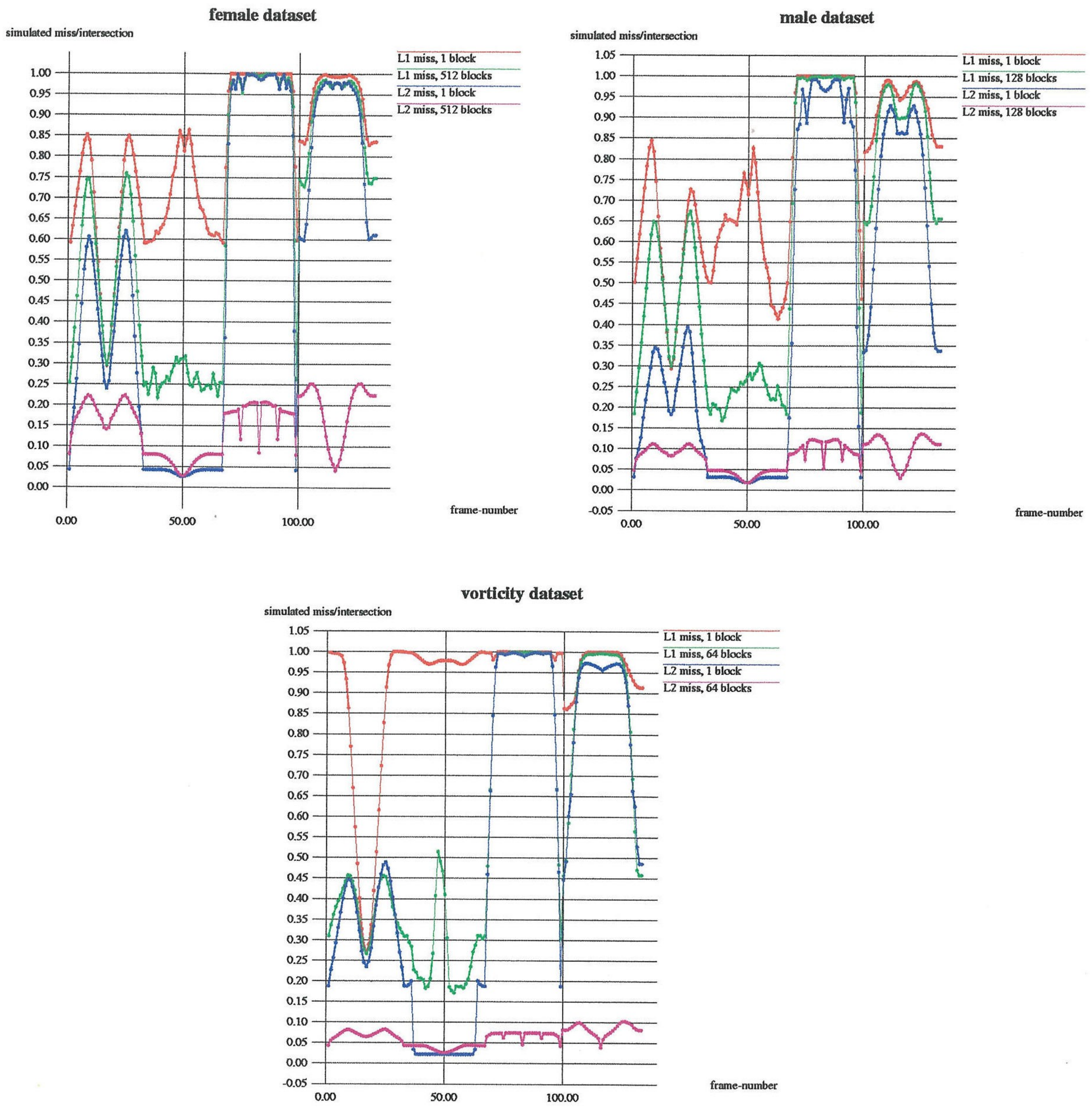


Figure 4.10: Simulation of L1 and L2 cache misses/intersection

axes. Whereas the X, Y, and Z axes were fixed to the voxel dataset, we will define axes U, V, and W to be fixed to the frame of reference of the viewer (or alternatively, to the display screen). U is the viewer's "up" direction; V is the "straight ahead" view direction, and W points to the viewer's left (in order to make U, V, W a right-handed coordinate system). When a ray accesses a voxel and an L2 cache hit occurs, there are two alternative reasons for that cache line to already be present in the cache: 1) it was fetched during a previous voxel intersection by the same ray; we will call such hits "help yourself" hits; or 2) it was fetched by a previous ray; we will call these hits "help your neighbor" hits.

We can expect "help yourself" hits to be most common when the V axis approaches the X axis: when the viewer is looking directly along X. As V turns gradually away from X, these hits will drop off gradually: with the perspective projection, all rays diverge slightly from V except the one in the exact center of the screen; as V moves away from X, the number of rays which point nearly down X and are still visible on the display screen will decrease slowly. Before V becomes perpendicular to X, there will be no such rays on the screen.

We can expect "help your neighbor" hits to be most common when the W axis approaches X: when the 128 byte cache lines are aligned with our horizontal scan lines across the screen. In this situation, rays that are cast near to one another in time will tend to share cache lines. (In fact, the cache lines generated by several rows of rays can be held in the cache at once).

Note in Figure 4.4 that, throughout frames 34 to 66, W is exactly aligned with X. In Figure 4.10, this sequence of frames has nearly uniform low L2 miss rates due to many "help your neighbor" hits.

The case for frames 1 to 33 is quite interesting. Frame 1 begins with W aligned with X: we get many "help your neighbor" hits. As the dataset rotates about Y, L2 misses increase. By frame 9, we reach a local maximum of L2 misses. However, as we move toward frame 17, where V aligns with X, we begin to get more "help yourself" hits. Frame 17 is a local minimum of L2 misses. As the dataset continues to rotate, we reach another local maximum of cache misses at frame 25, where X is again 45 degrees between V and W. Miss rates decrease again as we approach frame 33, which has many "help your neighbor" hits as W again aligns with X.

In frames 70 through 96, X is aligned with neither V nor W; we expect few "help yourself" or "help your neighbor" hits. This region indeed has the worst cache performance. The case is similar for frames 103 to 129, except we expect a small number of "help your neighbor" hits as W gets within 45 degrees of X at frame 116; this effect is observed as a slight reduction in cache misses near this frame.

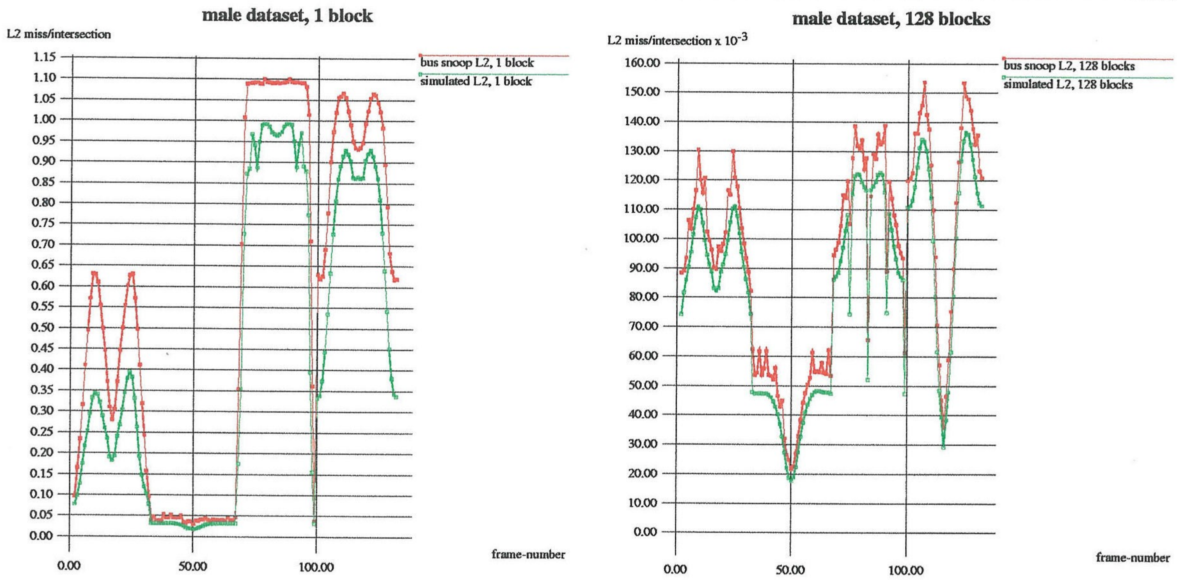


Figure 4.11: Simulated and bus-snooping board L2 misses/intersection, male dataset

Figure 4.11 validates the simulation results, showing the close correspondence between actual L2 misses per intersection counted by our simulated misses per intersection (green), and those actually measured by the bus-snooping board (red). The simulated misses are always an underestimate of actual misses, because the simulator takes into account only misses arising from access to the current dataset voxel. Not only does this neglect misses due to accesses to other data structures (such as the pixel accumulation buffer), it also neglects cache thrashing between the data voxels and other data structures. We would expect this to cause more error when L2 miss rates are very high and the cache begins to thrash more heavily. Indeed, note that the pair of lines for 128 blocks (right side of the figure) matches more closely than the pair for 1 block (left side of the figure). We can expect our simulator to yield a slightly low estimate when there are large numbers of cache misses.

This comparison also tells us that the memory access to the current data voxel is the source of the majority of all L2 misses in the algorithm. We cannot directly validate the simulation of L1 misses because L1 misses cannot be measured on the R8000; however, we have some degree of confidence in our simulator due to its success in matching the complex irregular features of the measured L2 misses in the figure.

Figure 4.12 compares the simulation results for two block sizes against our "mem-only" time per intersection (red and blue). The simulated time per intersection (green and purple) is derived from the cache miss counts of Figure 4.10. We multiply the counts per intersection of both types of miss by their corresponding penalty factor, and sum them, to generate the green and purple lines. Commonly cited estimates for L1 and L2 miss penalties are 89ns and 690ns, respectively.

The resulting simulated time (green and purple) and the corresponding empirical mem-only time (red and blue) lines are strikingly similar in shape for all three datasets and both numbers of blocks. For the optimal number of blocks for each dataset, the correlation between the blue and purple lines is extremely close. For one block (red and green), the simulated time is consistently somewhat lower than the "mem-only" time, most likely because of L2 cache thrashing between data voxels and other data structures when miss rates are high, which is not included in the simulated time.

In summary: The accuracy of our cache simulator is validated for L1 and L2 cache misses by a close match to L2 cache miss counts, and a close match to the time we attribute to memory system delays, to which both L1 and L2 misses contribute. Complex interpixel and intrapixel effects explain the directional dependence of cache miss patterns; these effects depend primarily on the relationship between the object space coordinate system, and the coordinate system of the viewer.

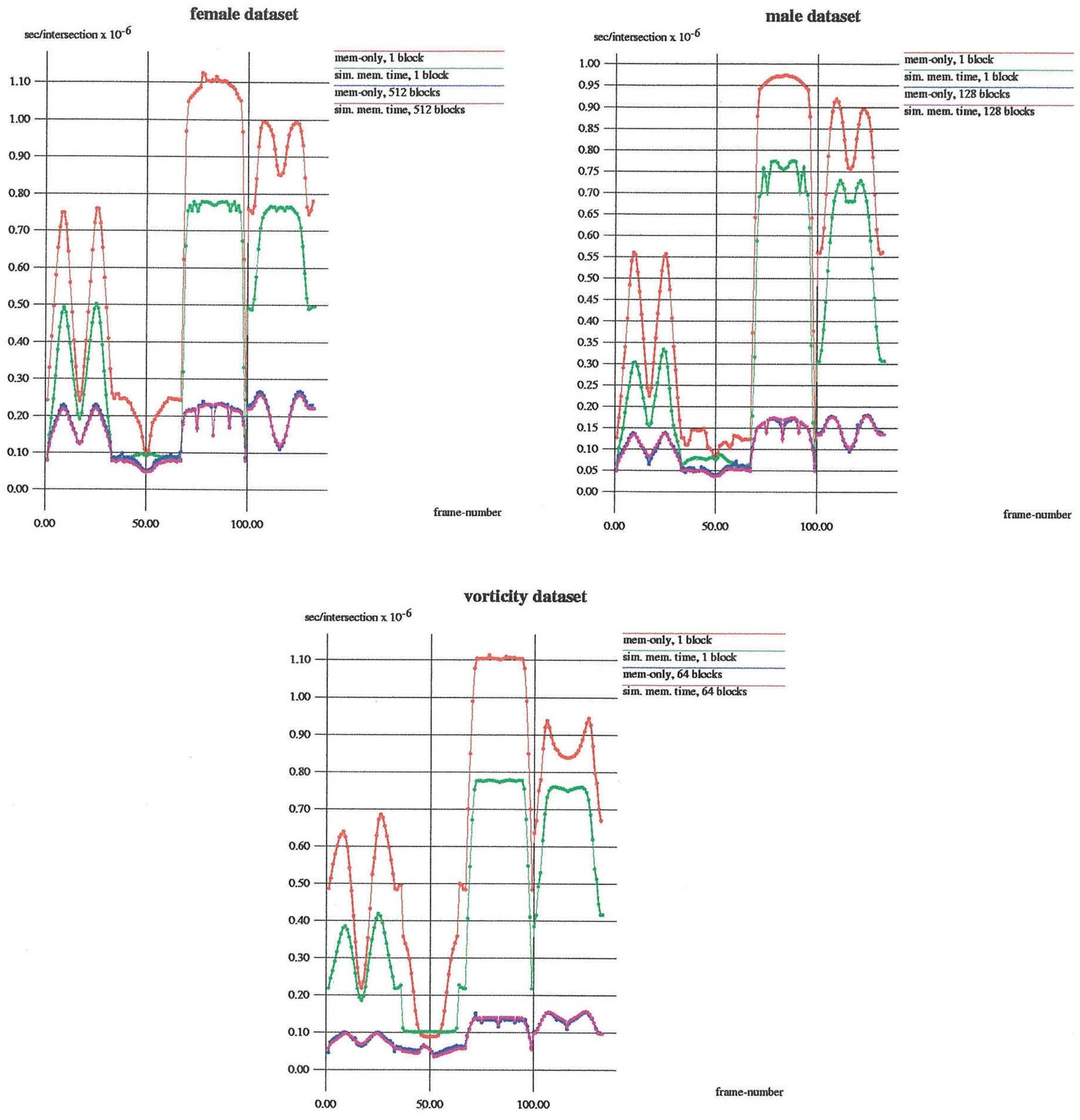


Figure 4.12: Simulated and "mem-only" memory hierarchy time/intersection

4.9 Conclusions

In this chapter, we have analyzed in detail the memory hierarchy effects present in uniprocessor ray casting through regular volumetric data, using experimental timings, bus monitoring hardware, and a software cache system simulator.

We have identified and characterized the main expense in a ray casting kernel with highly efficient instruction scheduling: cache miss penalties due to reads of the dataset voxels, in the context of extreme demands on the memory system. We have shown that this expense is extremely dependent on view direction. We have also demonstrated, however, that this dependence on view direction can be controlled by appropriate blocking of the data. The optimal block size for varied datasets is between 0.5 and 1.0 times the size of the L2 cache. We have isolated the L1 and L2 cache miss components contributing to this cost. Complex interpixel and intrapixel effects account for the majority of the dependence of miss rates upon view direction. These in turn depend on the relative orientations of the V and W axes (in the frame of reference of the viewer) to the X axis (in the frame of reference of the dataset). Interblock and interframe cache effects are negligible because of the high rates at which the cache is completely flushed.

We can predict that, were the bandwidth from the main memory increased to the bandwidth of the L2 cache (or, equivalently, if the L2 cache were the same size as main memory), performance for all frames would resemble the performance of frames 34 to 66, where L2 miss rates are almost zero. Using Figures 4.5 and 4.8, we can estimate that this would result in an average time per intersection of 320 ns/intersection for the female dataset; most of this time (approximately 250 ns/intersection) is due to "no-mem" costs, i.e., the time required by the processor to process instructions. In comparison to this, a single block requires 800 ns/intersection on average for all directions, and the optimal number of blocks requires 430 ns/intersection. Therefore, appropriate blocking achieves performance not far inferior to what could be achieved on a machine whose memory system was effectively "all cache." Further performance gain would require a faster processor clock.

We next apply these lessons to parallelization of the algorithm on the shared-memory Power Challenge architecture. Since we found interblock effects to be negligible, we expect our uniprocessor results to be largely applicable to the case of multiple shared memory nodes, independent of the processor to which a block is assigned, or the order in which the blocks are rendered. Minimizing cache miss rates will become even more important as multiple processors contend for use of the shared bus.

Chapter 5 One Shared-Memory Node, Multiple Processors

Figure 5.1 recalls our view of the memory hierarchy of a shared memory machine. On this architecture, the control of cache miss rates becomes even more important, because multiple processors contend for use of the bus. We will show that this inhibits parallel speedup in the case of poor cache performance, but that we can obtain excellent parallel speedup on all sixteen nodes when we manage cache performance carefully. Our choice of parallel partition will fundamentally affect how well we can exploit coherence to increase memory locality, and thereby improve memory system performance.

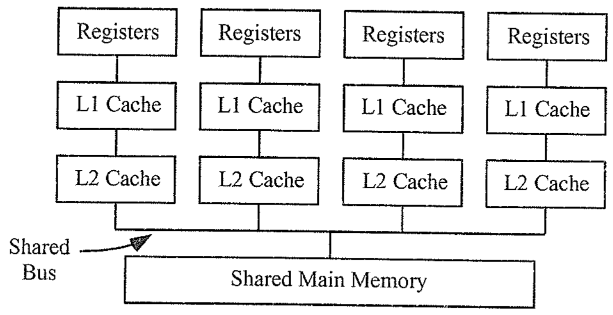


Figure 5.1: A typical memory hierarchy for a shared-memory architecture

5.1 Load balancing for ray casting on a shared-memory machine

We examine two combined parallel partitioning/load balancing algorithms suitable for ray casting on shared-memory architectures. Note that cache coherence can be exploited most effectively by grouping data voxels into roughly cubical blocks (rather than thin slabs or long columns of voxels). This maximizes the average chance over all directions that if a ray intersects the block, a neighboring ray will also intersect the block. Likewise, if an image partition is used, grouping of rays into roughly square tiles (instead of scanlines) maximizes locality of memory access. We will use these guidelines to eliminate many of the myriad possible choices of parallel partition.

5.1.1 Image partition

The first partition, the *image-tiled* partition, is a two-dimensional static subdivision of the image into M regular rectangles, or *tiles*, where M is greater than or equal to N, the number of processors. Load balance is achieved by *dynamic self-scheduling.* Tiles are placed on a work queue, and idle processors take tiles from the queue and render that sub-rectangle of the entire image. Good load balance can be achieved if M is sufficiently large relative to N, and in particular if the tiles can be roughly sorted in decreasing order of their cost. We use a tile's cost from the last frame as a predictor for its cost in the current frame, exploiting a graphics concept known as *viewpoint coherence*, the tendency for one viewpoint to be close to the previous one.

5.1.2 Object partition

The second partition, the *object-blocked* partition, divides the three-dimensional dataset into M regular rectangular solids, or *blocks* with M again larger than N. These blocks are placed on a similar work queue to the one used in the *image-tiled* partition, and load balancing is achieved in a similar manner. In the first phase of the algorithm, the *projection step*, idle processors take an individual block and cast rays through it to generate a *tile*, the projected image of the block. In the second phase of the algorithm, the *composition step*, the resulting M tiles are then sorted in increasing depth order and *composited* together to generate the entire image. There is some overhead associated with the composition step which is proportional to the total area of all tiles. The composition step is parallelizable by an image partitioning method which recursively combines the partial images. This two-step method is similar to that used in [Ma93] (but we do not use the same "binary swap" in the second step, because we are using a shared-memory architecture and do not have to communicate the partial images).

The image-tiled partition can easily accommodate global opacity clipping, since the disposition of a pixel is entirely determined by a single processor. In the object-blocked partition, the value of a pixel depends on the contributions of many tiles, rendered *a priori*, so global opacity clipping is infeasible. Intra-block opacity clipping is still possible, but is less effective. Object partitions, however, quite naturally conform to subdivision of the data into blocks that are stored contiguously in memory, providing the memory hierarchy performance gains associated with data locality discussed in the previous chapter. The image-tiled partition can be expected to have low data access locality since it treats the dataset as one large block.

5.2 The shared bus of the Power Challenge

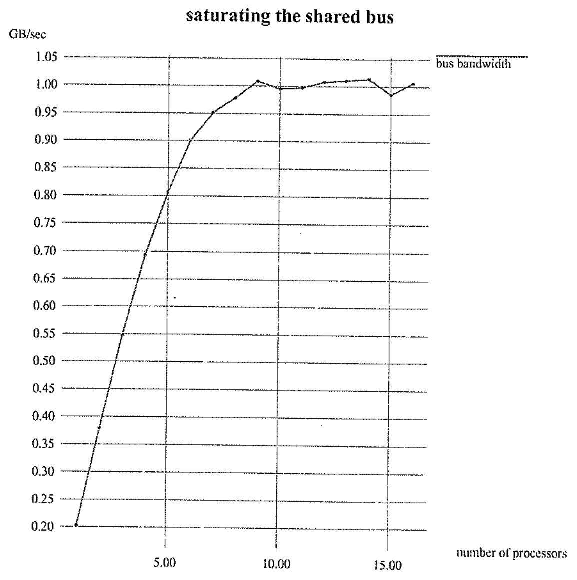


Figure 5.2: Saturating the shared bus

To measure the maximum total bus bandwidth of the Power Challenge, we wrote a simple program that causes as many L2 caches misses as possible. It allocates an array many times the size of the L2 cache, and then steps through it at a stride of 128 (missing in L2 each time), reading the values and adding their sum to an accumulator. We ran between one and sixteen such processes concurrently, obtaining the plot of total achieved bus bandwidth versus number of processors shown in Figure 5.2. We measured a maximum bus bandwidth of 1.0 GB/sec. (The machine we used for our experiments had two-way memory interleaving. We also measured the bus bandwidth of an eight-way interleaved machine, and found a maximum bandwidth of about 1.1 GB/sec.) The figure shows that more than eight processors making concurrent memory requests as fast as possible can saturate the bus. Therefore, other parallel programs with poor cache hits rates (e.g., our volume renderer when using a single large block) may saturate the bus and not exhibit parallel speedup past nine processors.

5.3 Experimental procedure

For the object-blocked partition, we determined empirically that for the female, male, and vorticity datasets, respectively, the optimal number of blocks (limiting ourselves to numbers that are powers of two) was 512, 128, and 64 (corresponding to block sizes of 2 MB, 2.8 MB, and 2 MB, respectively). This number is low enough that composition overhead does not begin to dominate, high enough to break the dataset into blocks smaller than the L2 cache, and high enough to allow good load balance on sixteen processors (we found that at least four blocks per processor provided the necessary granularity). All of the object-blocked experiments presented below were conducted with the optimal number of blocks for each dataset.

For the image-tiled partition, we found that there was negligible overhead associated with increasing numbers of tiles (up to at least several thousand), and that at least 32 tiles per processor were needed to provide the necessary granularity for good load balance from a variety of viewpoints. The optimal number of tiles did not depend on the dataset. Therefore, all image-tiled experiments, for all three datasets, were conducted with 512 tiles, yielding 32 tiles per processor when running on sixteen processors.

5.4 Parallel speedup

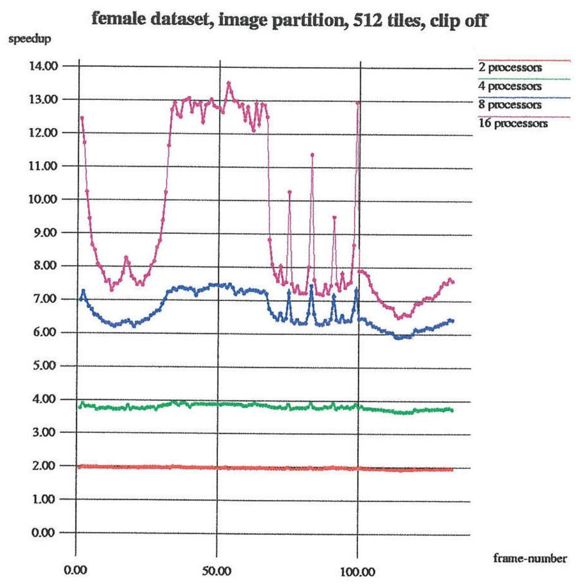
Figure 5.3 shows parallel speedup, compared to a single processor, for the image partition. The image partition does not exhibit parallel speedup past eight processors (as predicted by our bus saturation experiments, which found that nine or more processors experiencing high L2 miss rates can saturate the bus). The figure shows good parallel speedup on one, two, and four nodes for all view directions. However, for those views that have high L2 cache miss rates (i.e., 6 to 12, 22 to 28, 69 to 97, and 102 to 130), eight processors do not exhibit perfect speedup, and sixteen processors exhibit poor speedup. The results match those we obtained with our cache saturating experiment in Figure 5.2: the bus can become saturated by more than eight processors querying it as fast as they can.

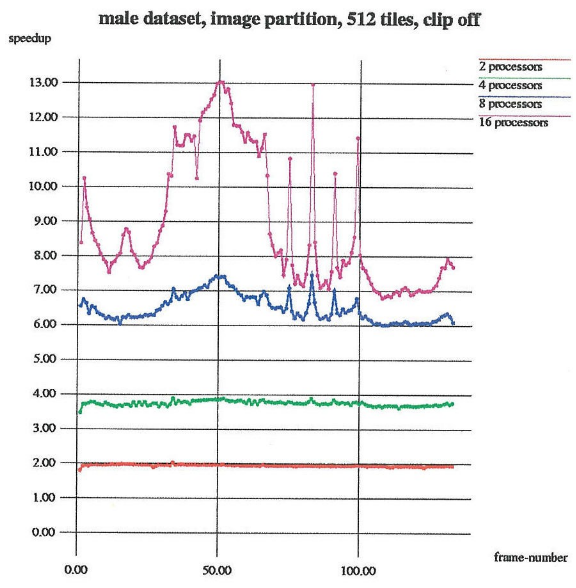
As shown in Figure 5.4, however, the object partition obtains much better parallel speedup: better than 7.5 on eight processors for all datasets; and on sixteen processors, averages of 13, 14, and 14.5 for the female, male, and vorticity datasets, respectively. Speedups with clipping turned on are not shown here for brevity but are qualitatively similar.

5.5 Load balance

Figures 5.5 and 5.6 show a measurement of the load balance for the image and object partitions, respectively. Our metric of load balance takes the ratio of the average time for all processors to complete a frame to the maximum time for any processor. If the value of this ratio is one, then the maximum and average are equal, and the time for all processors is equal: perfect load balance. As the ratio decreases, the average processor spends more time idle waiting for the slowest processor to finish. Note that a log scale is used so that small deviations from perfect load balance are visible.

For the image partition, in Figure 5.5, load balance is extremely good across all





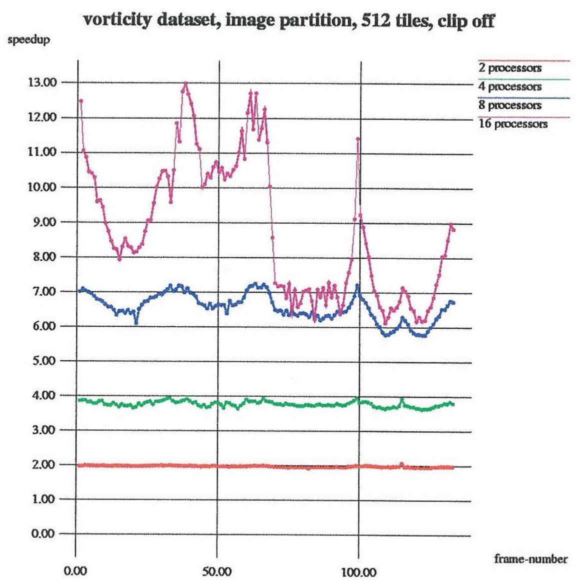
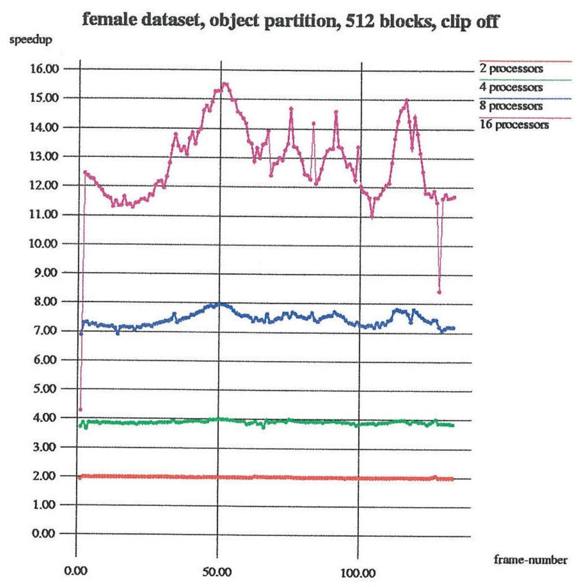
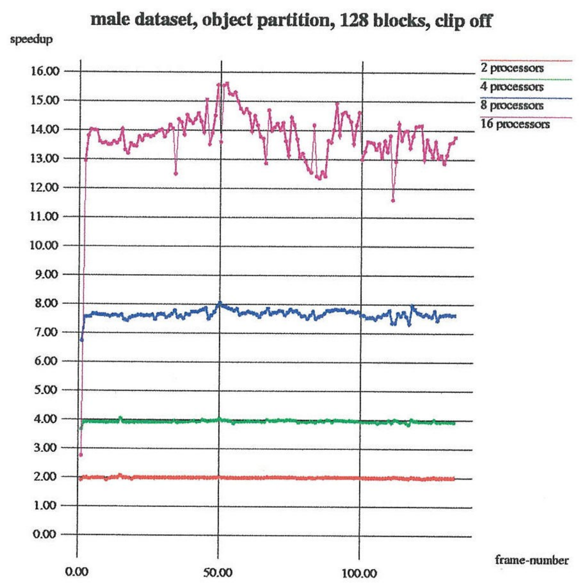


Figure 5.3: Parallel speedup, image partition, clipping off





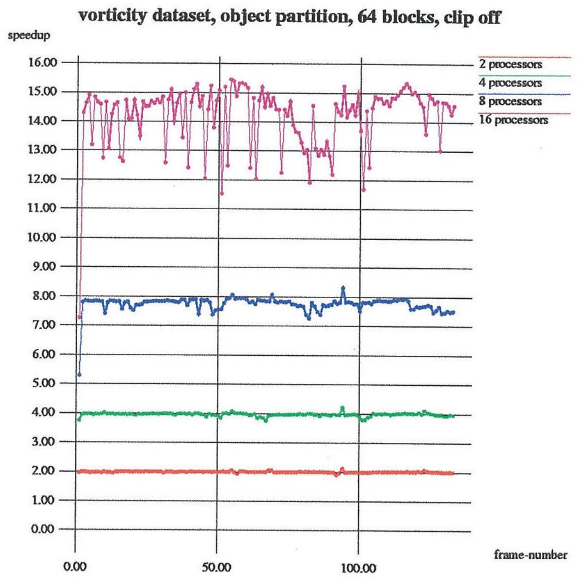
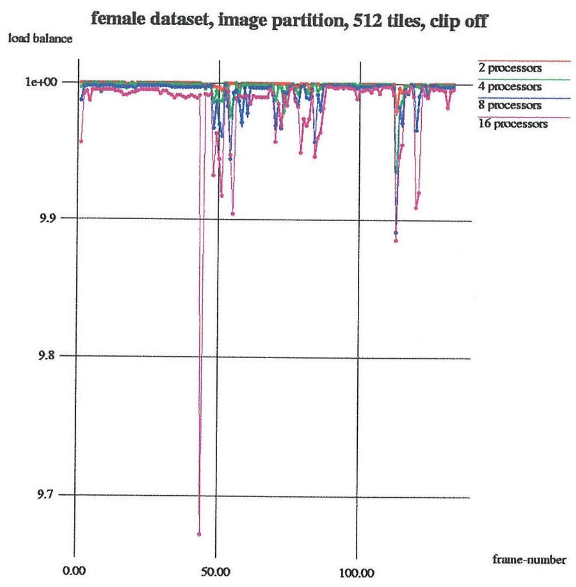
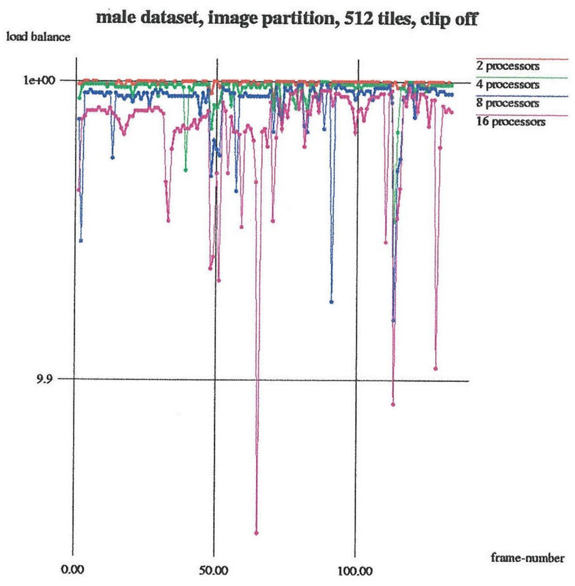


Figure 5.4: Parallel speedup, object partition, clipping off





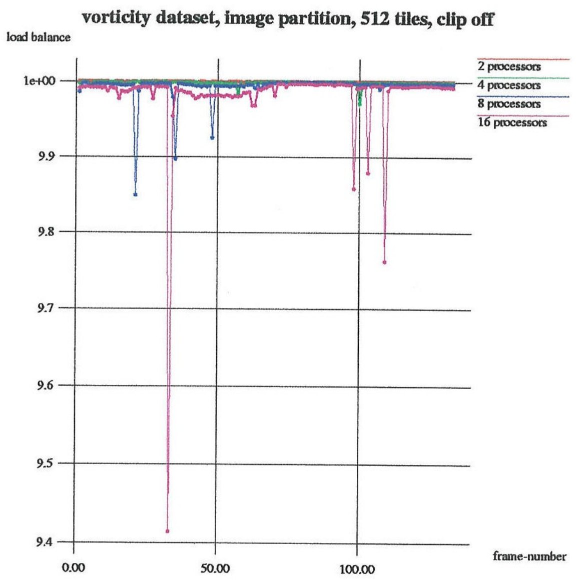
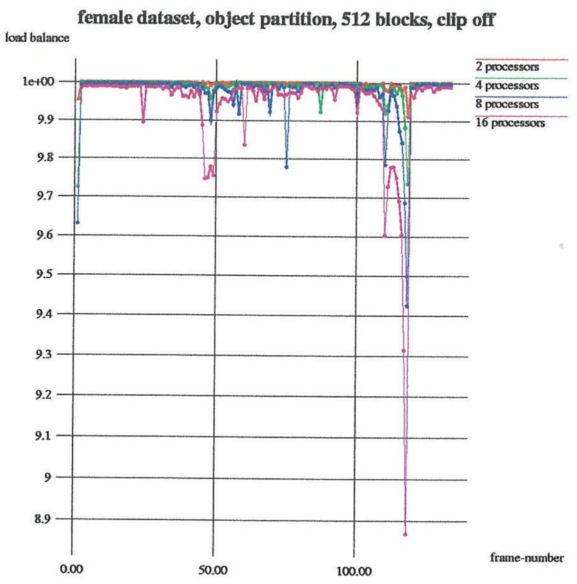
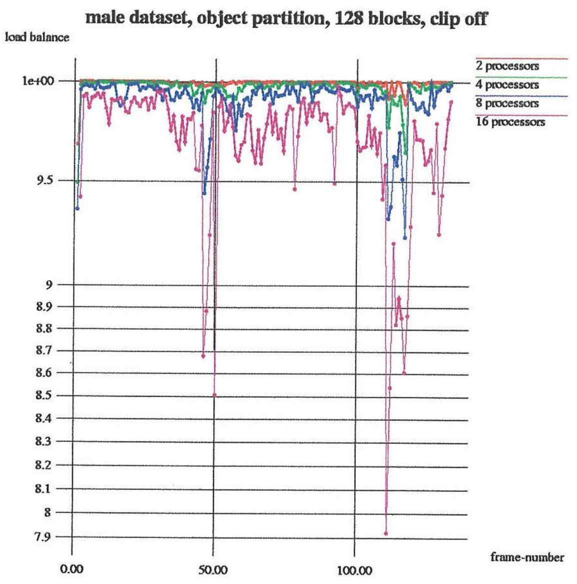


Figure 5.5: Load balance, image partition, clipping off





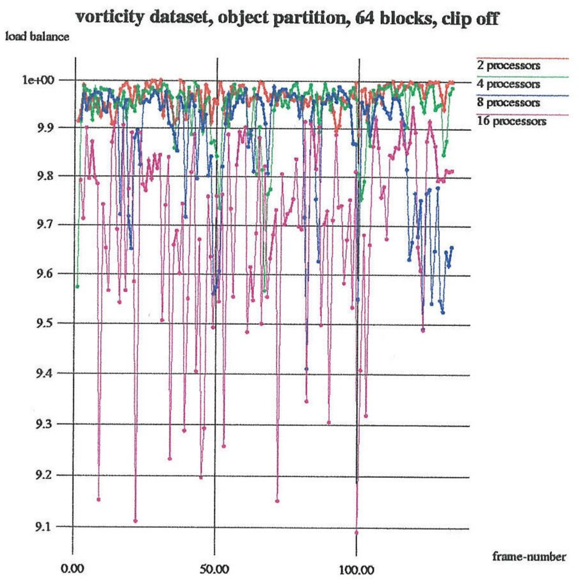


Figure 5.6: Load balance, object partition, clipping off

datasets and all frames. For all datasets, load balance never falls below .94, and is almost always above .99. The lack of parallel speedup of the image partition past eight processors is not due to poor load balance - all processors slow down equally, which accords with our explanation that bus contention is responsible for the lack of speedup on sixteen processors.

For the object partition, in Figure 5.6, the load balance metric is generally above .95. Only during frames 50 and 116 for the male and female datasets is the load balance slightly worsened. In both of these cases, the feet of the cadaver swing quickly past the viewer. This causes rapid changes in the rendering costs for the blocks in the feet of the cadaver: at one moment they are off the screen, and have no cost; at the next moment they take up a large part of the screen, and have a very high cost. Since we use costs from the previous frame to estimate the cost of the current frame, this rapid change can cause some blocks to be sorted out of order in terms of their true cost, causing load imbalance. Note that the vorticity dataset does not show this worsening of load balance at these frames; since the dataset is cubical in shape, one "end" of the dataset does not sweep particularly close to the viewer at these frames, so the rapid change in costs does not occur.

The generally high levels of load balance also further assure us that the lack of parallel speedup, which we are attributing to bus saturation due to poor cache performance, was not simply to poor load balance.

5.6 Comparison of partitioning methods

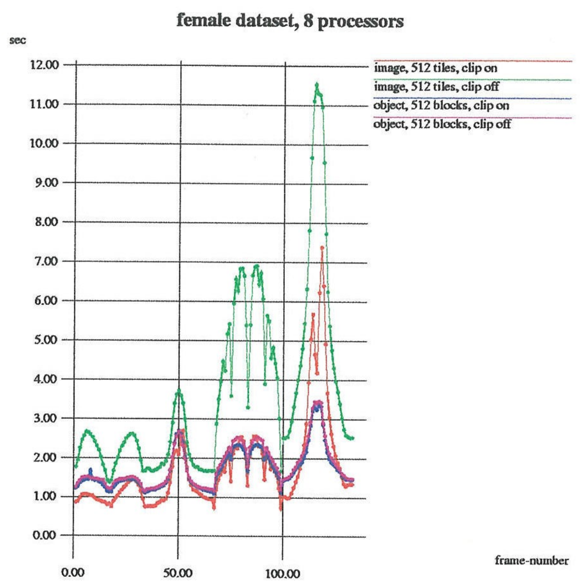
The image partition is inferior to the object partition as both a serial and a parallel algorithm. On a single processor it simply obtains worse cache performance; on more than eight processors, this poor cache performance leads to saturation of the bus, and the image partition is unable to produce further parallel speedup. However, recall that the image partition does have one advantage over the object partition: it allows the optimization of global opacity clipping to be effective. The efficacy of this optimization is dataset-dependent, but can be significant. (The object partition can only do per-block opacity clipping, which is much less effective.)

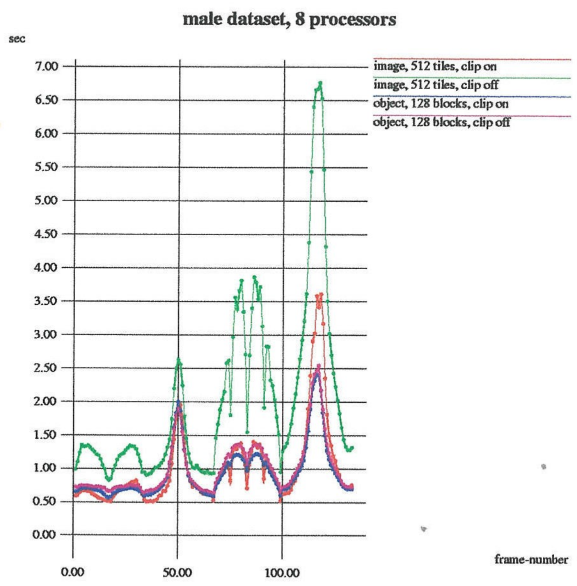
Figure 5.7 compares four variations of the algorithm on eight processors: the image partition with clipping on (red), and clipping off (green); and the object partition with clipping on (blue), and clipping off (purple). The image partition with clipping on (red) is slightly superior to the object partition (blue and purple) for the male and female datasets for many of the frames with low L2 miss rates (frames 1 to 61). Both datasets contain a large opaque object, and these algorithmic performance gains outweigh the other factors. However, for frames with worse L2 miss rates, the object partition is superior. Furthermore, on the largely translucent vorticity dataset, opacity clipping is less effective. On this dataset, the object partition is much superior to the image partition for all views except where L2 miss rates are lowest (frames 42 to 58) — there it is slightly superior. Notice further that for the vorticity dataset, the object partition with clipping off (purple) is actually faster than the object partition with clipping on (blue); rays become opaque within a single small block so rarely that it is cheaper not to bother testing for them at all.

To further champion the object partition, Figure 5.8 compares the same four variations with sixteen processors. On sixteen processors, the image partition does not run much faster than on eight, but the object partition continues to speed up. For the female and male datasets on sixteen processors, the object partition with clipping on (blue) is fastest for most views. For the vorticity dataset, the object partition with clipping off (purple) is again the fastest for nearly all views.

5.7 Maximum frame rates

Taking the best partition for each dataset in Figure 5.8, we calculate the average frame rate across all viewpoints for each dataset. For the 1 GB female dataset, on sixteen processors at a resolution of 400 by 300 pixels, we get an average of 1.0 frames per second, using the object partition with clipping on. This is faster than has been previously cited in the literature for a dataset this large. Furthermore, we have attempted to find the most expensive view directions and include them in our





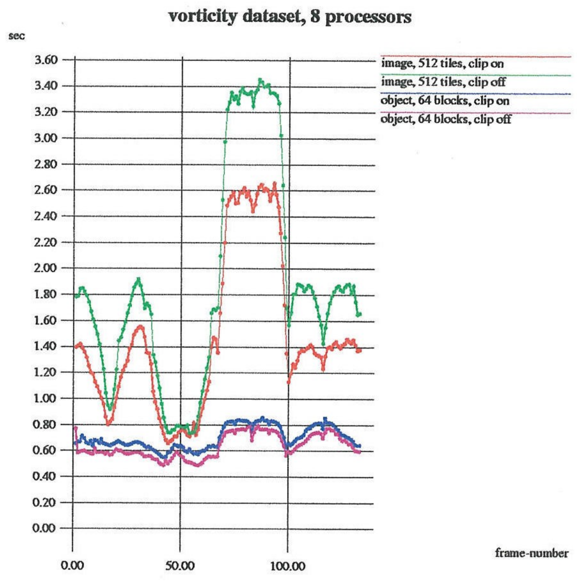
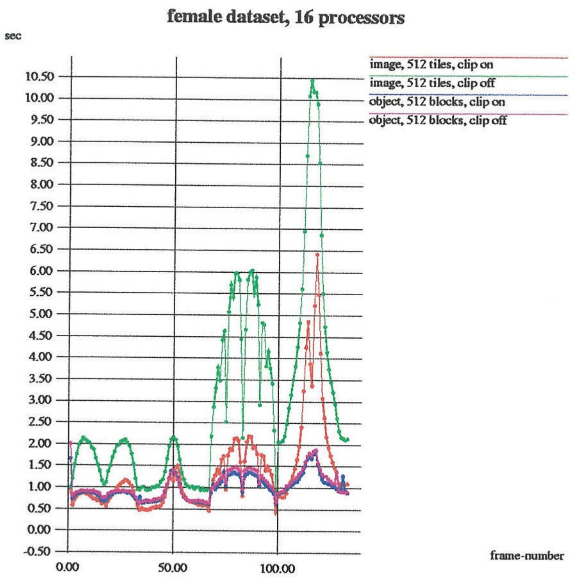
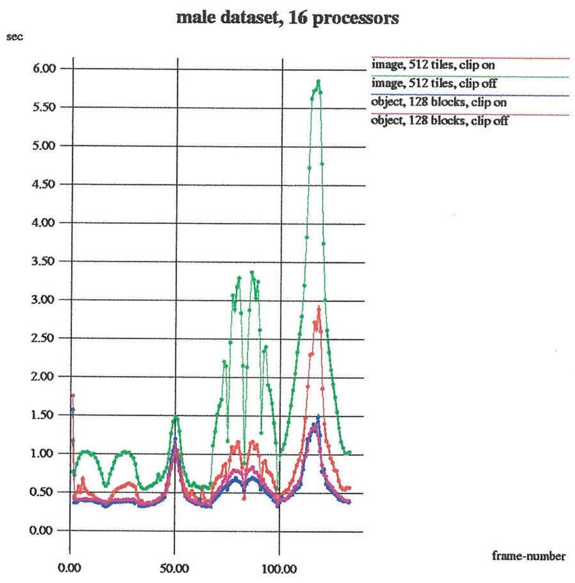


Figure 5.7: Time per frame, eight processors





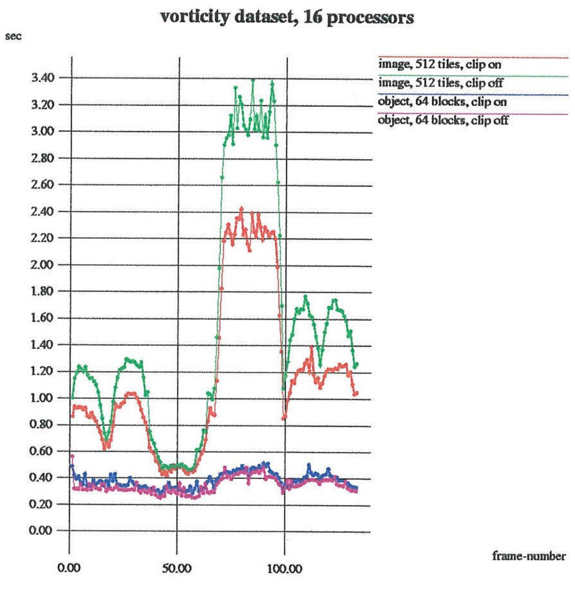


Figure 5.8: Time per frame, sixteen processors

average. For the 357 MB male dataset, using the same partition, we get an average of 1.9 frames per second. For the 128 MB (512^3) vorticity dataset, using the object partition with clipping off, we get an average of 2.9 frames per second.

5.8 Conclusions

In this chapter, we have applied our results regarding the management of memory locality from the previous chapter to rendering on multiple shared-memory processors. In this context, the goal of minimizing L2 miss rates becomes even more important, since multiple processors must contend for use of a shared bus. We have shown that, if considerations about L2 misses are not taken into consideration, parallel speedup beyond eight processors (on the Power Challenge system) can be thwarted due to bus saturation; however, if the coherence present in ray casting is carefully exploited, cache performance can be improved, and excellent speedup on all sixteen processors can be obtained.

We have demonstrated the consequences of memory hierarchy effects on two types of parallel partition: one object partition, and one image partition. The object partition is generally superior because of the natural blocking of the data that it affords. Only in the limited case of certain views with low L2 miss rates, on datasets with large opaque areas which make opacity clipping effective, is the image partition more effective. We obtain good load balance on both partitions, and good parallel speedup up to sixteen nodes on the object partition, which achieves better cache performance.

We believe that the hardware designers of the Power Challenge struck a good balance between the bus bandwidth and the combined processor power of the Power Challenge. It is possible for the processors to saturate the bus, but careful algorithmic design can help avoid this.

We can predict that frame rates could not be achieved simply by adding more processors onto the same bus: With the poor cache performance of a single block, the image partition saturates the bus for most view directions on more than eight nodes. With the better cache performance due to appropriate blocking on the object partition, we achieve good parallel speedup on sixteen processors. However, the fact that this speedup is better for the vorticity dataset than for the larger female dataset (and, in particular, the way in which the parallel speedup for sixteen processors on the female dataset in Figure 5.4 drops off for exactly those frames with worst cache performance), tells us that on the female dataset, we are close to saturating the bus with sixteen processors, even with good blocking. Therefore, if we added more processors to the bus, our performance would not improve unless the bus performance were also proportionally increased.

This focus on the two primary algorithmic factors important to shared-memory rendering performance: efficient exploitation of the low levels of the memory hierarchy; and a parallel partitioning algorithm that provides accurate dynamic load balance; has allowed us to attain the highest frame rates for large 357 MB to 1.0 GB datasets that we have seen heretofore cited in the literature, 1.9 and 1.0 frames per second average, respectively, on the *axisorbit* test suite, which was specifically designed to reveal poor cache performance.

The results of chapters 4 and 5 can be found summarized in [Palmer97a]. In the next chapter, we further extend our characterization of memory hierarchy effects for ray casting to multiple shared-memory nodes.

Chapter 6 Multiple Distributed Shared-Memory Nodes

Figure 6.1 recalls our view of the hybrid distributed/shared memory architecture. The task before us in this chapter is to combine what we have learned from previous chapters, with the answers to the new empirical questions posed in this chapter, to devise the optimal means to exploit the entire deep, heterogeneous memory hierarchy of this system.

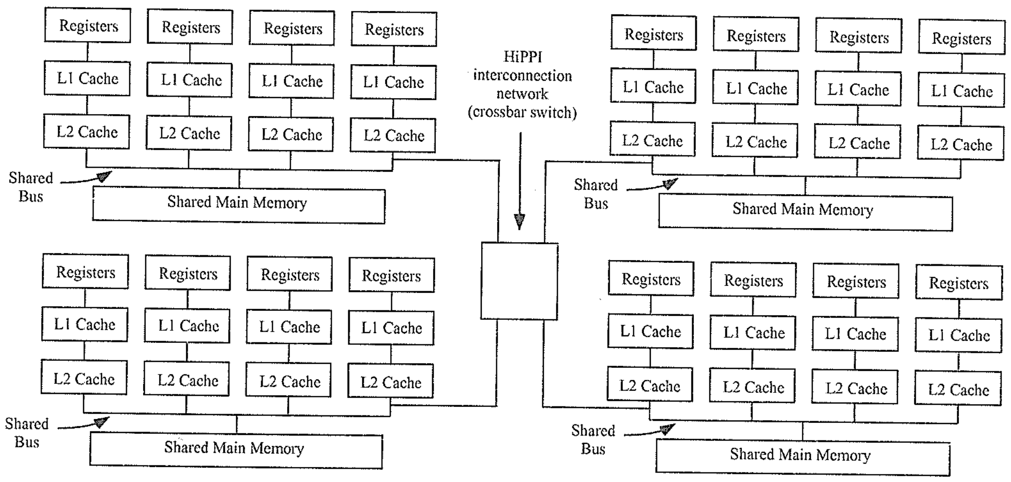


Figure 6.1: A hybrid distributed/shared memory hierarchy, the Power Challenge Array

Connecting multiple nodes together with a network adds another level to the memory hierarchy. As before, this level is characterized by slower communication than the level beneath it.

Chapter 4 demonstrated that memory locality within a single node can be optimized by blocking the dataset into blocks between 0.5 and 1.0 times the size of the

L2 cache. Chapter 5 shows that this approach was required for good parallel speedup within a single shared-memory node. This helps us to prune the vast space of possible parallel partitions. To maximize cache performance within each node, we will continue, within a single node, to use the methods developed in these chapters. We will also use the same blocks as a unit of data to be distributed and cached across the parallel nodes.

In a distributed memory system, memory "accesses" from remote nodes take the form of explicitly passed messages, and no automatic caching of these accesses is available to the applications programmer. Since we have more information available to us than is available to an automatic caching system, we can make extremely efficient use of the available cache memory.

The system we have implemented dedicates a fixed amount of memory per node to storing the dataset. This amount is a factor between 1 and N times the size of the dataset. If we set the factor to 1, we have exactly as much distributed storage as is needed for one copy of all blocks. As the factor approaches N, we can begin to replicate blocks, until there is enough storage to replicate the entire dataset on each node. Blocks will not have an affinity to any particular node; they may float dynamically among the nodes. However, we will guarantee that there remain at least one copy of each block in the collective memory of the nodes. Since there is no "master" copy of any block, the distinction between "cache memory" and "main memory" is made irrelevant: one node's locally "cached" copy of a block is another node's remote "main" copy.

6.1 Volume rendering requires a global gather operation

Since every voxel will in general make a contribution to an image that will ultimately reside in the memory of a single node, volume rendering requires a global *gather operation.* Consider a ray intersecting the data volume. This ray intersects an ordered set of voxels, whose colors and opacities must be composited together to generate the final color of the ray. On a distributed computer, these voxels may be stored in either: 1) a replicated fashion, in which every node holds a copy of the entire dataset (which is not a scalable solution), or 2) a distributed fashion, in which each node holds a subset of the voxels. In the latter case, all the voxels intersected by a ray will not, in general, be stored on the same node. Therefore, in the absence of complete replication of the dataset, global communication is an integral part of the generation of a frame. Some representation of the entire dataset must pass over the HIPPI communication network for every frame. Our goal is to minimize the communication required for this task (without sacrificing memory hierarchy performance at lower levels).

Such communication is to be distinguished from occasional communication for the purpose of load balance, which is not an integral part of the global gather operation of the ray casting algorithm *per se*.

6.1.1 Transferring 2D subimages versus 3D subvolumes

How can we accomplish the global gather operation required to generate a frame with minimal communication? Since composition is associative (but not commutative), as described in section 2.1.4, we have several options of how to partition the task among the distributed nodes. We will suggest two prominent ones, distinguished by whether they require the transfer of volume blocks, or image tiles.

As we consider object and image partitions below, we will restrict ourselves to near-cubical blocks (i.e., not slabs or shafts) and low numbers of near-square tiles (i.e., not scanlines), in order to maximize locality within tiles and voxels.

Perhaps the most straightforward method would be to assign a region of the screen to each node (an image partition), and send to each node a copy of all volume blocks overlapping its region of the screen. This transfer would represent the required global communication to generate a frame. This node could then cast rays through all the blocks without further communication. However, this method would, in general, require that the volume blocks move again for the next frame. In the worst case, the entire dataset might have to move for each frame; although we could hope this would be lessened by frame coherence, which would cause blocks to tend to overlap the same areas of the screen for one frame as they did for the previous frame.

Note, however, that if we have chosen to replicate the entire dataset on each node, then this image partitioning method can function with no communication at all (except to display the final image at a central location, which is required for any partition we choose). We will note this and continue under the assumption that we would like to avoid replication of the entire dataset.

Alternatively, we could keep the volume blocks stationary (except for occasional load balance), and transfer the tiles generated by projection. Each node will generate the projected two-dimensional tiles from the blocks it holds. The composition step will require that overlapping sets of tiles be collected at some location to be composited together. We may use an image partition to assign single large regions of the screen to processors for the composition step; each processor will receive a copy of all tiles that overlap its region of the screen (this is the required global communication to generate a frame), and will generate the final image for that region.

The choice is between using an image partition, and moving the 3D blocks; and using an object partition, and moving the 2D tiles. Roughly equal numbers of objects will need to be transferred in either case. In earlier chapters, we showed that the blocks are larger than typical tiles: The block size for optimal L2 cache performance on the R8000 is between 2MB and 4MB. The average tile for typical views of the dataset in our experiments was one-fifteenth this size at 150 KB (i.e., 100^2 pixels, at 4 floating point values per pixel). Our conclusion is that, for the case of a distributed dataset, communication volume can be reduced by using an object partition, which requires that we communicate the 2D tiles, but allows the 3D blocks to remain in place.

We therefore employ an object partition among each of the distributed nodes for the projection step, and an image partition among the nodes for the composition step. The global gather operation will consist of the transfer of tiles. (Note that some transfer or replication of blocks may still occasionally be required for load balance.)

If we do choose to replicate the entire dataset on each node, however, note that the image partition can operate with no communication at all, other than transport of the final image to a central location for display, which is required by any partition.

6.1.2 The tradeoff between replication of data and communication expense

A continuum of memory expenditure exists between complete replication of the entire dataset on each node, and the distribution of the dataset such that only a single copy of each volume block exists anywhere in the collective memory of the nodes: we can also partially replicate any fraction of the dataset. That is, if the number of distributed nodes is N, we can usefully expend any amount of memory between 1 and N times the size of the entire dataset. Blocks may be statically replicated, or may (occasionally) move dynamically among the distributed nodes, with some scheme of caching. Communication can be decreased when the dataset is completely replicated on each node. However, we would like to minimize replication so that we can render a larger dataset on a given machine. The interesting engineering question becomes: As the replication factor of the data decreases from N to 1, how can we minimize the associated performance degradation?

Before discussing the dynamic caching/replication scheme used in our most recent work, we will describe two of our early attempts at each extreme of the replication continuum:

Complete replication

Our first attempt to extend our work on single shared-memory nodes to multiple nodes involved complete replication. We were considering only a few (four to eight) distributed nodes, and sought the highest performance. The dataset under consideration was 357MB, much smaller than the memory (2GB to 4GB) of the individual nodes, giving us no reason not to replicate the entire dataset on each node.

We used the recursive binary image partition shown in Figure 6.2. The screen is divided by N recursive cuts where N is the total number of processors on all nodes. The lines dividing the screen can be adjusted dynamically to maintain load balance, both among the distributed nodes, and among the processors within each node.

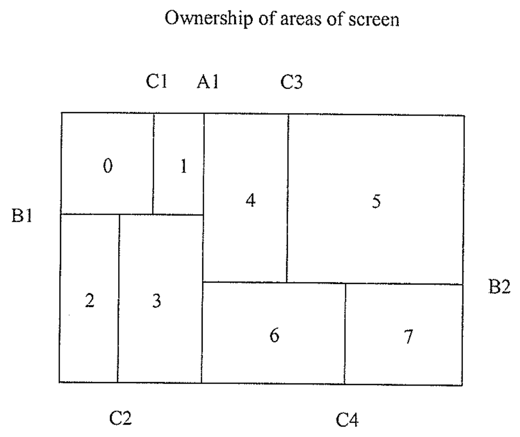


Figure 6.2: An image partition based on binary subdivision

This partition was the one used at Supercomputing '94 and '95, and it attained extremely high performance. For the 357MB male dataset, we attained rates of up to 10 frames per second for a "fly through" sequence of frames, at a resolution of 400 by 300 pixels, using a cluster of eight Silicon Graphics Power Challenge machines with a total of 64 processors. This is better performance than previously cited in the literature for this size of dataset.

No replication

Prior to our work on the Power Challenge, we implemented a volume rendering system for irregular tetrahedral data on the Intel Delta machine, using projection methods. This system is described in [Palmer94b] and Appendix A of this thesis. We chose to use an image partition on a relatively fine-grained MIMD machine, and did not replicate the data. Communication (predictably, in hindsight) was the limiting factor of the algorithm.

6.2 A software block caching system for the Power Challenge Array

Between the two extremes of no replication and N-fold replication lies the option of partial replication, or caching, of some the data blocks. Although a Power Challenge Array consists of only two to eight nodes (as opposed to the 512 of the Intel Delta), replication by those factors is still a high memory overhead to pay. If we could save this memory overhead, we could render datasets two to eight times larger in total size. (For instance, below we render a 7.1 GB dataset, which is larger than the memory of any individual node.)

Although volume rendering requires global operations, most MIMD machines do not provide to the programmer a logical global address space for the collective memory of all the nodes. Therefore, we have implemented in software the semantic equivalent of a logical global address space for the volume blocks, with a mechanism for caching blocks. The object partition we have chosen does not require communication of the blocks as an integral part of the algorithm; however, the replication of some blocks, and the occasional communication of some blocks, gives us great flexibility with respect to reassigning blocks to nodes for load balancing. As the replication factor of the blocks increases, the flexibility in choices of reassignment of blocks that do not require communication also increases. As the replication factor decreases, it will become more common that attaining adequate load balance will require some communication of blocks. In addition, this allows us to avoid the communication of many tiles: we can assign many blocks for projection to the same node that will require them for composition, eliminating the need to transfer a tile.

In the absence of automatic caching between the nodes, we can implement whatever caching scheme we want. Because we have absolute control over what blocks to send to each node and complete information on what blocks are currently held by each node, we can achieve more efficient use of memory than could be achieved by an automatic cache. We can also precisely regulate communication: when assigning blocks to nodes for projection, we can be sure to assign blocks only to nodes which already hold a copy of them. Although maintaining load balance will sometimes require the transfer of blocks, this communication will only occur when we explicitly initiate it. This is not the case for an automatic cache, for which the programmer can only have a rough idea of how to reduce cache misses on average. In particular, we will specify that there be no "master" copy of any block, although we will guarantee that a copy of each block exists somewhere in the collective memory of the nodes. This causes a blurring of the distinction between a separate "cache" and "main memory."

We designed and implemented the following distributed memory caching system to handle the storage, communication, and caching of the data blocks. This system has a number of parameters that are adjustable at run time.

Each node has a fixed number of cache slots that it can use to store a block (which are all nearly the same size). Given a number of blocks B in the dataset, the *replication factor* we choose specifies how many slots will be required collectively on the N nodes. A replication factor of R requires that each node maintain $\frac{R \times B}{N}$ cache slots, for a total of $R \times B$ slots on all nodes collectively.

The contents of the caches of all nodes are maintained and coordinated from a central master process — the same process that is connected to the user interface and sends the viewpoint to be rendered to all nodes at the beginning of each frame.

The B blocks are initially assigned to the nodes for projection in round-robin fashion. (We found that random assignment worked equally well.) The intent is to avoid concentrating the blocks assigned to any node in any region of the dataset; this would tend to cause load imbalance since the costs of adjacent blocks are highly correlated. Before starting the projection step for the first frame, each node will load from disk the blocks which it is assigned to project. If R is greater than one (1), then there will be room in the collective memory of the nodes for more than one copy of each block. The master process instructs the nodes to fill up their caches with (fairly assigned) additional blocks that they will not be projecting for this frame.

The nodes then project their blocks, send the resulting tiles to the nodes whose assigned regions of the screen for the composition step overlap the tiles, perform composition, and send the final results to a central location for display.

At the end of each frame, the master receives information about the times to render each block for that frame. This information is used for dynamic load balancing after the first frame. The blocks are placed on a list sorted in decreasing order of cost. Blocks are removed, most expensive first, and assigned to nodes by a modified greedy algorithm, in the following way:

If the projected tile of a block overlaps the screen area assigned to a node, we will say that the block *requires the tile for composition*. Based on the *load balancing threshold* and the work required for the last frame, we determine the *work threshold*, the maximum amount of estimated work that can be assigned to any node. The algorithm will assign no more blocks to a node that has passed this threshold (unless all nodes have).

We desire an ordered set of priorities for the algorithm: 1) keep the estimated load balance above the load balancing threshold, 2) avoid communication of blocks, 3) assign blocks to the same nodes which require them for composition (thereby avoiding communication of tiles). We implement this set of priorities with the following multitiered greedy algorithm.

```
For each block, in decreasing order of cost:
{
 Find the node with the lowest assigned work so far, which holds a
   copy of the block, and requires its tile for composition.

 If the total estimated work assigned to this node, plus the
   estimated cost of the block is under the work threshold, then:
 {
   Assign the block to this node.
 } else {
   Find the node (besides the previous one) with the lowest assigned
     work so far which holds a copy of the block.
   If the total estimated work assigned to this node, plus the
```

```
        estimated cost of the block is under the work threshold, then:
      {
        Assign the block to this node.
      } else {
        Find the node with the lowest assigned work so far.
        Assign the block to this node.
      }
    }
  }
```

If it is required that a block be transferred, another node must be found that holds a copy of the block. If more than one other node holds a copy, we use a fair algorithm to choose between these nodes, so that the average number of blocks served by each node is the same.

Since the caches of the nodes are generally kept completely full, a node must delete a block to make room for each block it receives. The master decides which blocks each node will delete from its cache, making sure that at least one copy of each block remains somewhere in the collective memory of the nodes (alternatively, if the communication network is much slower than disk access, the master could allow blocks to be flushed from the cache to be later reloaded from disk, but that is not the case with HIPPI).

We know that complete replication is likely to yield better performance than no replication. The primary experimental question we wish to answer is the following: as we decrease the replication factor, how drastically does performance suffer? We will show that the algorithm outlined above performs extremely well to exploit coherence to generate memory locality, and only requires a small performance penalty as the replication factor is decreased.

6.3 Datasets and the *typicalview* test suite

For our multiple node experiments, we were interested primarily in the internode

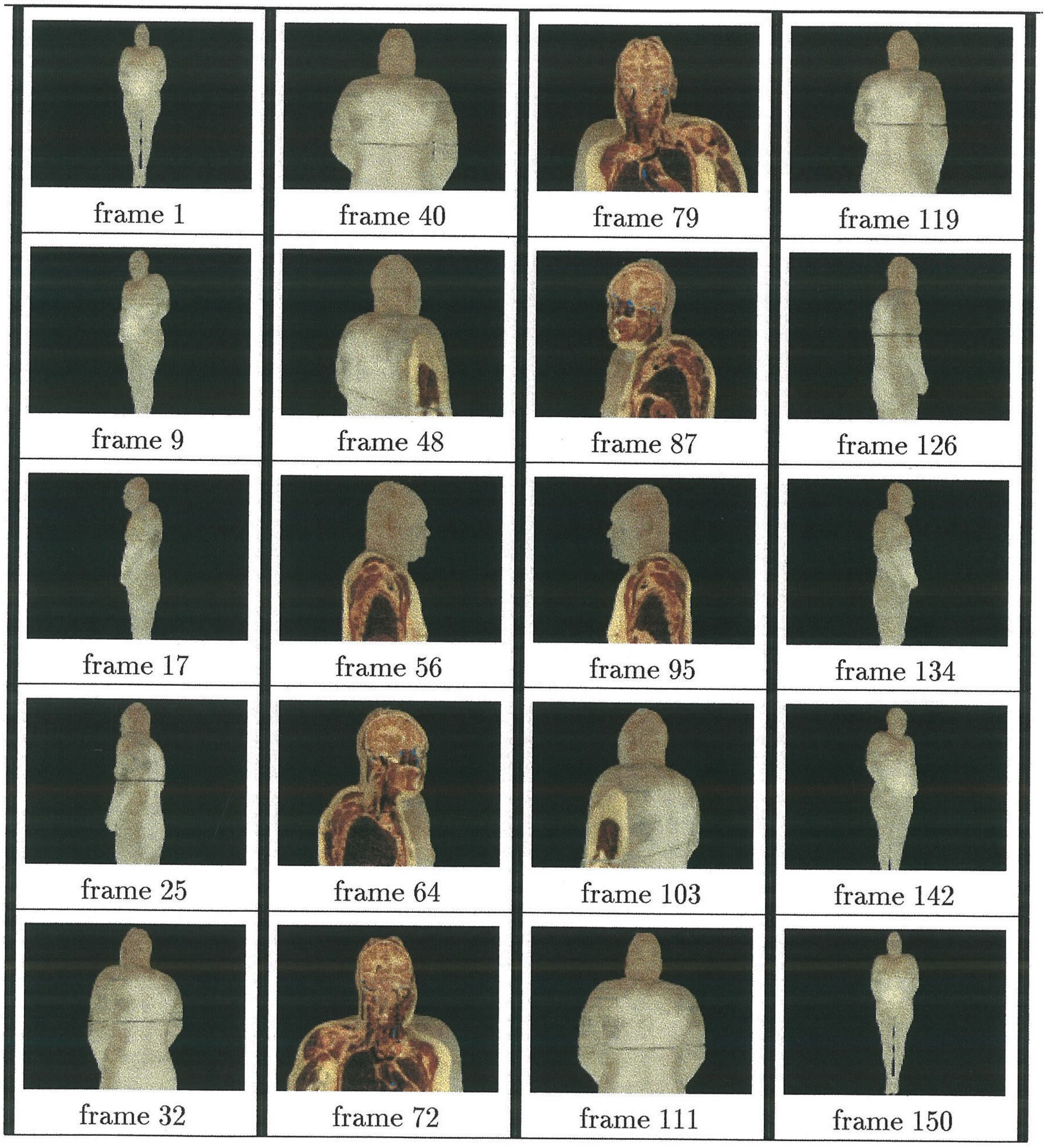


Figure 6.3: The *typicalview* test suite

caching effects of our logical distributed global address space. To exercise this system fully, a different test suite was required from the *axisorbit* test suite used in previous chapters, which was intended to explore the directional dependence of low-level cache performance. Figure 6.3 shows the *typicalview* test suite. This suite is intended to create 1) concentrations of load in physical space, and 2) high temporal and interblock variance in the projection costs of the volume blocks. These tend to increase the need for load balancing by block transfer. In frame 1, all blocks are visible, and the variance in their costs is low; the viewer zooms in and up so that by the midpoint at frame 75 (not shown), many blocks are not visible on the screen, causing them to have no cost. All the cost of the projection step is concentrated in the blocks in the upper quarter of the dataset. The viewer continues to rotate around the body and zoom out and down again, back to the original viewing position by frame 150, where all blocks are again in view, and again all have roughly equal cost. The suite is named the *typicalview* suite because it follows a typical smooth path that a viewer might take under the constraint of frame coherence.

Dataset Name	Dimensions	Total Size
female	840x2595x480	1 GB
male	584x1878x341	357 MB
full-female	1610x894x5192	7.1 GB

Figure 6.4: Sizes of datasets for multiple node experiments

Figure 6.4 shows the sizes of the datasets used for our multinode experiments. We have deleted the vorticity dataset from our experiments, because 1) it is too small for running on multiple Power Challenge machines; and 2) its cubical shape makes it unsuitable for the *typicalview* suite, which zooms in to the top half of the dataset to intentionally push some blocks out of view. We add an additional very large dataset, the *full-female* dataset. This dataset is derived from the same Visible Female data as the *female* dataset, but it is at the full spatial resolution of the original data, whereas the female dataset was subsampled in each dimension by a factor of two.[1]

[1]The female and full-female datasets are rotated 90 degrees about X relative to one another;

The full-female dataset cannot be used in all our experiments because of its size: it is too large to fit on less than four nodes (including the size of the dataset and other overheads associated with rendering). At 7.1 GB this is, to our knowledge, the single largest regular volume dataset yet rendered.

6.4 Experimental results

6.4.1 Experimental procedures

We conducted our multinode runs on the Power Challenge Array at the National Center for Supercomputing Applications at the University of Illinois, Urbana-Champaign. We selected to use eight processors per node in most runs, for two reasons: 1) we wanted to concentrate on the highest level of the memory hierarchy, and did not want to incur the chance of saturating the bus, a lower-level effect, which could complicate our results; 2) NCSA had available four R8000 machines; two of them had sixteen processors, and two had eighteen processors. It has been our experience on the Power Challenge that it is beneficial to leave one or two processors of the machine free for operating system tasks, to prevent unpredictable contention. If this contention occurred on two of the nodes, and not on the other two, it could introduce noise into our timings that would be difficult to analyze. Therefore, for most of our experiments, we used eight processors per node (an exception: the attempt in Section 6.4.8 to achieve maximum frame rates on the 7.1 GB dataset). This reduces the speed of projection and composition, but does not effect the volume of data communicated among the nodes, which is the primary subject of study of this chapter.

Certain complications existed for the 7.1 GB full-female dataset. First, this dataset, plus the other memory overheads of our implementation, is too large to fit in the collective memory of only one or two nodes (which have 4GB of RAM each). Therefore, we run the full-female GB dataset on four nodes only. In addition, the

however, we rotate the view matrix for the full-female dataset before rendering it, so the body has the same final orientation on the screen (although the memory stride is still rotated about X). The orientation of the full-female is the orientation of the original source data.

IRIX operating system contains a 2 GB limit on the total memory allocated to a single process. Therefore, we were required to map eight virtual nodes on to the four physical nodes. Each of the eight rendering processes then required less than 2 GB of total memory. This virtual mapping makes it impossible to compare communication timings between runs of the full-female dataset and the other datasets, because messages sent to "remote" nodes may sometimes physically be on the same node, and may have different performance characteristics.[2] However, valid comparisons can still be made about the number of bytes communicated, which would be identical to the number of bytes sent among eight true physical nodes.

We chose not to use any dataset-dependent algorithmic optimizations for most of our runs (an exception: the attempt in Section 6.4.8 to achieve maximum frame rates on the 7.1 GB dataset) In this chapter, we are most concerned with studying the movement of data at top level of the memory hierarchy, rather than optimizations to ray casting, and want our results to be generalizable to any dataset regardless of the values of the voxels.

We chose to render a larger image size (800x600 pixels) for the multinode experiments, for two reasons: 1) the image size we used in the previous chapters (400x300 pixels) is small relative to the increased computational power in a multinode array; 2) since this chapter concerns the movement of data among the nodes, which consists primarily of the transfer of image tiles, we wanted to increase the average size of the tiles to generate more communication load.

We conducted our experiments with four settings of the "load balance threshold" (.65, .80, .90, and .95) described in Section 6.2, and for three cache replication factors (on four nodes, factors of 4, 2, and 1; on two nodes, replication factors of 2, 1.5, and 1). We are primarily interested in these two variables' effect on: 1) the total bytes of data blocks communicated (for the purpose of load balance), 2) the total bytes of image tiles communicated (for the purposes of the global gather operation of ray

[2]There is the possibility that the operating system will recognize network packets intended for the local machine, and not send them physically over the HIPPI network; also, hardware contention might exist, for instance for resources on the HIPPI board, that would not exist on eight true physical nodes.

casting), 3) the achievable load balance, and 4) the final average frame time.

6.4.2 Communication due to block transfers

Figure 6.5 shows the average bytes per frame due to the transfer of volume blocks, for the male and female datasets on four nodes. Each table shows the results of twelve experimental runs (four values of the load balancing threshold by three replication factors). The replication factor is expressed as a fraction: number of blocks cached per node over the total number of blocks. For example, "512/512 cached" means that each node cached all 512 of the 512 total blocks; on four nodes, this corresponds to a replication factor of four.

The values in the table show the total number of bytes sent by any node to any other node, summed over all 150 frames of the *typicalview* test suite.

Female dataset, four nodes:

lb threshold	512/512 cached	256/512 cached	128/512 cached
.65	0	0	0
.80	0	0	0
.90	**0**	**262,156**	**180,232**
.95	**0**	**1,179,703**	**4,980,970**

Male dataset, four nodes:

lb threshold	128/128 cached	64/128 cached	32/128 cached
.65	0	0	0
.80	0	0	910,860
.90	0	273,258	4,076,100
.95	0	1,502,919	7,173,026

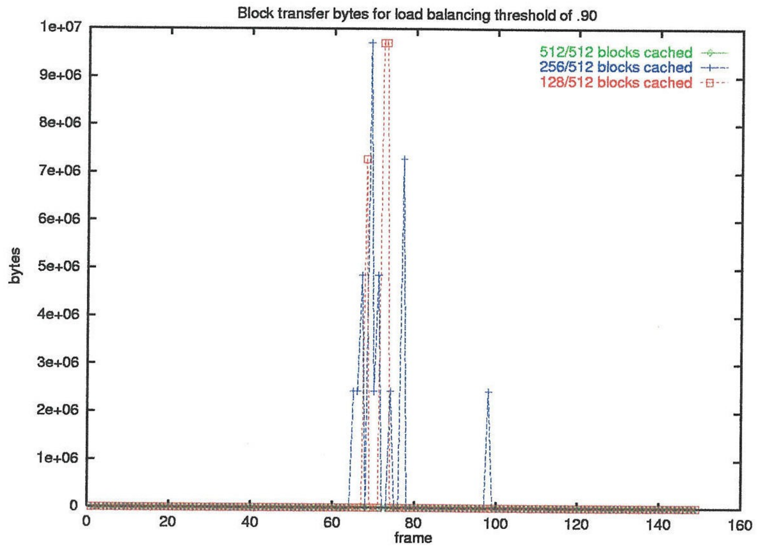
Figure 6.5: Average block transfer bytes per frame on four nodes of eight processors

Note that no communication of blocks is required for low load balancing thresholds or high replication factors. As the load balancing threshold increases from top to bottom, the number of bytes transferred tends to increase, as we demand that the algorithm attempt to achieve higher load balance. As the replication factor decreases from left to right, it becomes generally harder to satisfy a given load balancing threshold with the blocks currently held by each node, so the algorithm attempts to rectify this by moving blocks. Note that there is one exception to this tendency: the values 262,156 and 180,232 are reversed in this respect. However, the difference between the two "reversed" values is only 80 KB, whereas a single block is between 2 MB and 3 MB. This represents a difference of only four or five blocks over the entire test suite. This type of noise can occur, because the attempt to improve load balance by moving a block is not guaranteed, and can result in several frames of adjustment. We can still expect a general tendency of increasing block transfers when the load balance threshold increases, or the replication factor decreases.

The **boldface** cells in Figure 6.5 indicate the cells that are plotted in detail in Figure 6.6, which shows, frame by frame, the number of block transfer bytes that were sent. The upper half of Figure 6.6 corresponds to a load balance threshold of .90, for three different replication factors. The case for 512/512 blocks cached on each node (green) requires no block transfers for any frame; this is full replication of the dataset on each node. The cases for 256/512 cached (blue) and 128/512 cached (red) are the "reversed" pair of values mentioned above. Note that all transfers are concentrated about frame 75, the middle of the *typicalview* suite, where the observer zooms in closely to the data, and many blocks are out of view, causing a rapid shift in block costs, and a need for block transfer to balance the changing workload. The number of total blocks transferred per frame (the number of bytes per frame can be seen to vary in increments of 2.5MB, the size of a block) is low, according with our explanation of random noise for this "reversal."

The lower half of the figure corresponds to a load balancing threshold of .95. This causes the transfer of many more blocks for the cases for 256/512 cached (blue) and 128/512 cached (red). In this case, 128/512 clearly requires more transfers than 256/512. Again, the transfers are clustered about the rapid changes in block costs in the middle of the test suite.

Figure 6.7 shows average block transfers on two nodes. Note again the general tendency for transfers to increase as either the load balancing threshold increases, or



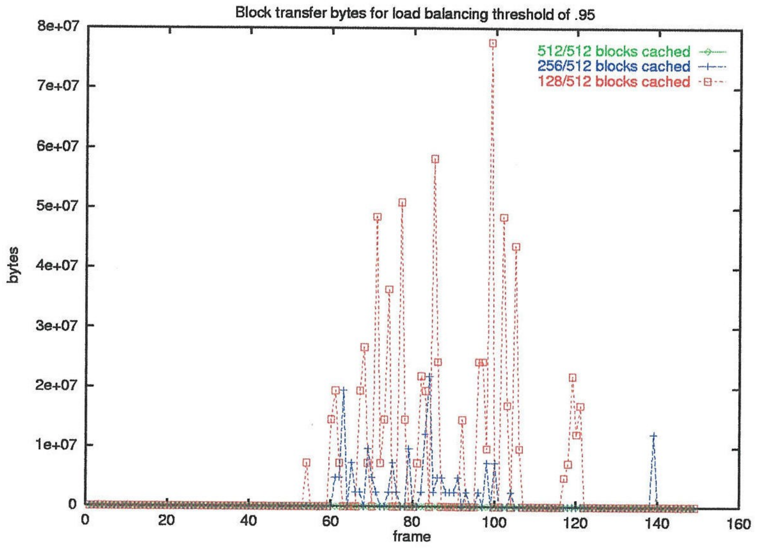


Figure 6.6: Block transfer bytes per frame on four nodes of eight processors, female dataset

the replication factor decreases.

Female dataset, two nodes:

lb threshold	512/512 cached	256/512 cached	128/512 cached
.65	0	0	0
.80	0	0	0
.90	0	0	0
.95	0	458,774	1,343,551

Male dataset, two nodes:

lb threshold	128/128 cached	64/128 cached	32/128 cached
.65	0	0	0
.80	0	45,543	91,086
.90	0	569,288	318,801
.95	0	1,594,006	2,322,694

Figure 6.7: Average block transfer bytes per frame on two nodes of eight processors

Figure 6.8 shows average bytes due to block transfers per frame, using the 7.1 GB full-female dataset on virtual eight nodes of four processors each. Recall that we were required to map eight virtual nodes onto four physical nodes for this large dataset, as described in Section 6.4.1. Using four processors per virtual node results in a total of eight active processors on each of the four physical nodes. For this very large dataset, we ran only at a replication factor of 1 (i.e., 256/2048 blocks cached on each of eight nodes).

No block transfers were required for this large dataset up until a load balancing threshold of .90. (We did not run an experiment using .95 because four-node time on the NCSA machines was extremely limited, and we were confident that this would require a greater volume of transfers. As shown below, the extra communication for a load balancing threshold of .90 consisted of "load balance thrashing", and did not result in performance benefits.)

In summary, the volume of bytes communicated due to block transfers, in general, tends to increase: 1) as we demand a higher load balancing threshold; and 2) as the replication factor is decreased, allowing less flexibility in the assignment of blocks to

Full-Female dataset, eight virtual nodes:

lb threshold	256/2048 cached
.65	0
.80	0
.90	26,989,872

Figure 6.8: Average block transfer bytes per frame on eight virtual nodes of four processors

meet a given load balancing threshold.

6.4.3 Load balance

There are two factors contributing to load balance: 1) Active attempts to achieve load balance by the transfer of blocks; and 2) a high granularity, together with round-robin initial assignment of blocks to eliminate concentrations of cost, will tend statistically to yield a near-average value on all nodes. This force will continue to act even as block costs change, as long as the changes are also well distributed among the nodes. We will call this effect *stochastic load balance.*

If the load balancing threshold is set too high, our algorithm may transfer blocks without reaping a corresponding improvement in load balance. We will call this situation *load balancing thrashing.* This can happen, for instance, because our estimates for the costs of blocks, based on the costs for the last frame, are imperfect.

Figure 6.9 shows the average load balance achieved for the female and male datasets. The results are complex to interpret. Recall that our load balancing algorithm is attempting to make a compromise between several factors, in this order: 1) requiring that the estimated load balance be over a certain threshold, 2) minimizing block transfers, and 3) minimizing tile transfers by assigning blocks for projection to the same nodes that will require their tiles for composition. Note that attempting to satisfy factors 2 and 3 can actually result in *slightly worse* estimated load balance; the only constraint is that the estimated load balance be *no worse* than the threshold, Female dataset, four nodes:

lb threshold	512/512 cached	256/512 cached	128/512 cached
.65	**0.883**	**0.946**	**0.960**
.80	0.887	0.947	0.961
.90	0.912	0.947	0.956
.95	**0.934**	**0.951**	**0.956**

Male dataset, four nodes:

lb threshold	128/128 cached	64/128 cached	32/128 cached
.65	**0.924**	**0.928**	**0.890**
.80	0.923	0.927	0.896
.90	0.924	0.930	0.914
.95	**0.934**	**0.932**	**0.928**

Figure 6.9: Average load balance on four nodes

not that it be maximized.

For the female dataset, the reader may be initially surprised to see that load balance values in the column for 512/512 blocks cached are actually *worse* than those for 128/512 blocks. This is because the load balance is good even without the transfer of blocks. When the estimated load balance is above the required threshold, the algorithm allows some imbalance to occur, in order to increase the number of blocks assigned to a node that would require them for composition (factor 3, above). In the 128/512 case, the algorithm did not have this flexibility; it could not change the assignments without incurring communication cost. The result is that the load balance for 128/512 is better, but more tiles have to be communicated.

Note further, that for the male dataset, the situation is reversed: the 128/512 column achieves worse load balance than 512/512 case. This is because the granularity of the dataset is much lower. This results in less stochastic load balance; in turn, as the estimated load balance falls below the thresholds, this causes the algorithm to attempt many more block transfers. The resulting load balance thrashing yields an overall worse load balance for the 128/512 case. At this low granularity, our estimates of block cost are not accurate enough to yield better values of load balance than .92

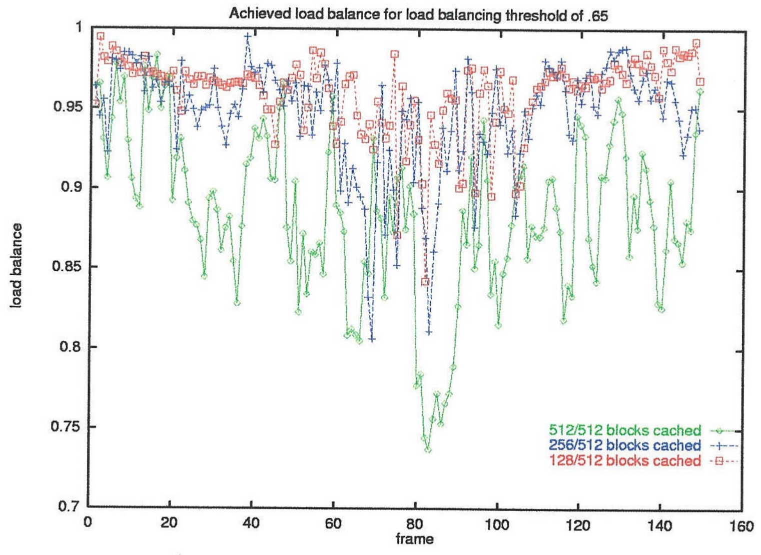
or .93 (on average) for the male dataset; the algorithm will therefore experience this thrashing if the load balance threshold is set too high; the problem is exacerbated at load replication factors.

Figures 6.10 and 6.11 show detail from the **boldface** rows of Figure 6.9. The upper half of Figure 6.10 shows the detail for a load balancing threshold of .65 for the female dataset. Note that for a load balance threshold of .65, no block transfers occurred (recall the block transfer bytes in the upper half of Figure 6.6). In the case of 128/512 blocks cached (red), stochastic load balance at this high granularity provides excellent load balance even without these transfers. The case for 512/512 blocks (green) is able to take advantage of this load balance, which far exceeds the required threshold, and improve the assignment of tiles for composition to the nodes that projected them. The 128/512 case does not have this flexibility of tile assignment, since reassignment would require communication of blocks; it can only take this windfall in the form of good load balance. The case for 256/512 nodes has a moderate amount of flexibility (less than the 512/512 case), and so exchanges some of the "excess" load balance for a better assignment of tiles.

The lower half of Figure 6.10 shows the case for a load balancing threshold of .95 for the female dataset. The stochastic load balance is so good that even for this extreme load balancing threshold, the 512/512 case (green) is still able to exchange some extra load balance for improved tile assignment.

The upper half of Figure 6.11 shows a load balance threshold of .65 for the male dataset. Note again, that no block transfers occurred for this case (recall the block transfer bytes in the lower half of Figure 6.6). With the reduced granularity of the 128 blocks of the male dataset, the 32/128 (red) case struggles to maintain a load balance above the threshold, because stochastic load balance is less effective. However, the 64/128 and 128/128 cases do better, and are able to again exchange some load balance for improved tile assignment.

In the lower half of Figure 6.11, for a load balancing threshold of .95, all three replication factors must resort to many block transfers to try to improve their load balance. However, it does not do much good. The accuracy of our load balancing



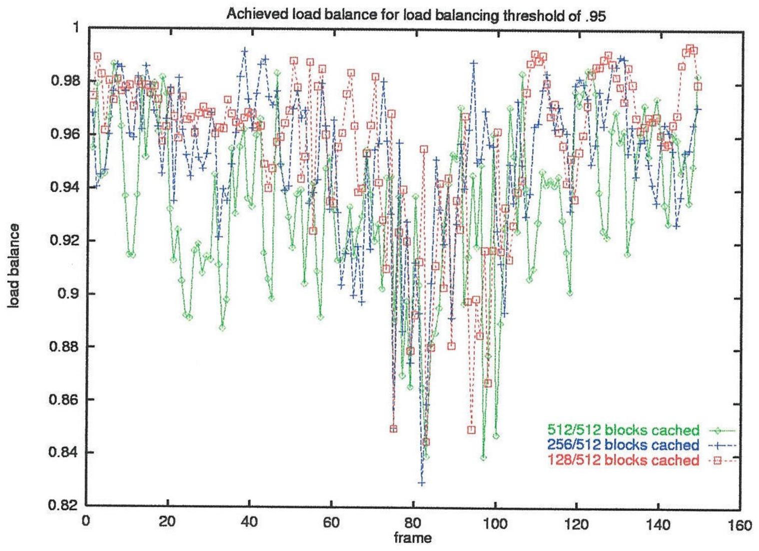
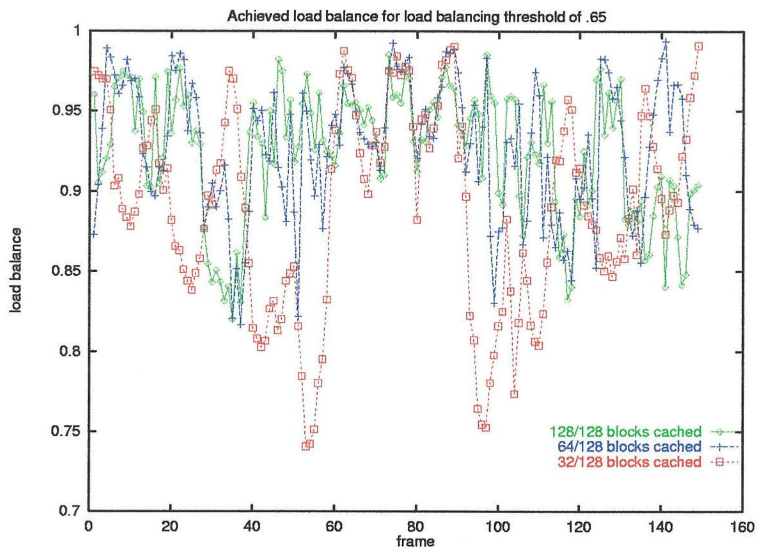


Figure 6.10: Load balance on four nodes, female dataset



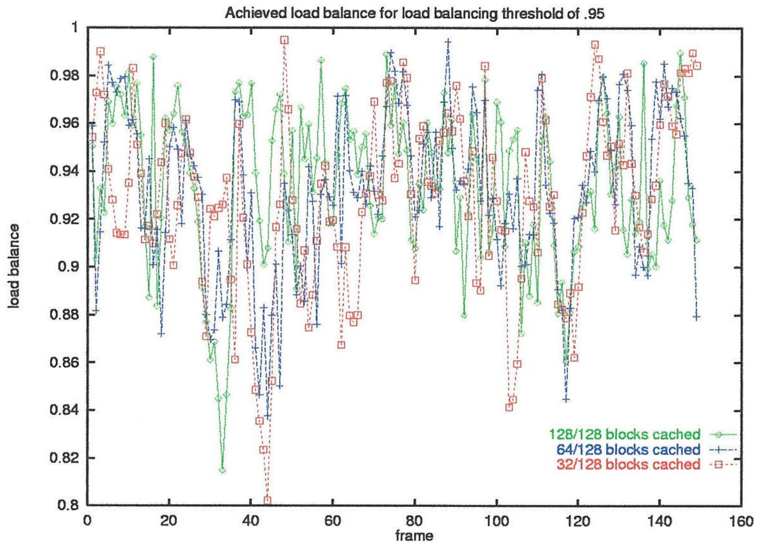


Figure 6.11: Load balance on four nodes, male dataset

is limited by our imperfect estimates of the costs of blocks for a future frame. In this case, the algorithm is unable to achieve good load balance despite the transfer of many blocks, that is, it is experiencing load balance thrashing.

Female dataset, two nodes:

lb threshold	512/512 cached	256/512 cached	128/512 cached
.65	0.926	0.981	0.978
.80	0.926	0.982	0.978
.90	0.942	0.982	0.978
.95	0.963	0.983	0.976

Male dataset, two nodes:

lb threshold	128/128 cached	64/128 cached	32/128 cached
.65	0.947	0.931	0.907
.80	0.947	0.940	0.925
.90	0.949	0.941	0.945
.95	0.956	0.958	0.951

Figure 6.12: Average load balance on two nodes

Figure 6.12 shows average load balance on two nodes. The trends seen in the four node case are repeated. For the high granularity of the female dataset, a lower replication factor leads in general to better load balance (but with the side effect of worse tile assignments). For the lower granularity of the male dataset, a higher replication factor can achieve better load balance than a low replication factor, in the absence of good stochastic load balance by turning its flexibility of assignment into better average load balance.

Full-Female dataset, eight virtual nodes:

lb threshold	256/2048 cached
.65	0.915
.80	0.914
.90	0.923

Figure 6.13: Average load balance on eight virtual nodes of four processors

Figure 6.13 shows load balance for the full-female dataset. We observe that load balance increases slightly with increasing load balance factor; however, recall from Figure 6.8 that this required enormous amounts of block transfer. It is better, with the extremely high granularity of this dataset, to rely on stochastic load balance, and select a lower load balancing threshold to avoid these block transfers.

In summary, the load balance achieved results from a complex interaction of granularity, replication factor, the competing goals of the load balancing algorithm: requiring that estimated load balance be above a threshold, avoiding block transfers, and improving tile assignments when possible), and the accuracy of our predictions of future block costs.

High granularity yields benefits in term of stochastic load balance. In cases of high replication factor, the algorithm can take advantage of this with better load balance (in the case of a high load balance threshold) or with improved tile assignments (in the case of low load balance threshold); in the case of low replication factor, the gains may only be taken as improved load balance.

Low granularity yields worse stochastic load balance. In cases of high replication factor, the algorithm is able to use its flexibility of assignment to its advantage, gaining in either load balance or improved tile assignments; in cases of low replication factor, the algorithm is less equipped to do well in the absence of stochastic load balance.

6.4.4 Communication due to tile transfers

There are two reasons a tile may have to be transferred: 1) it overlaps the screen area of some node, but since optimization of tile transfers is only the third priority of our load balancing algorithm, it was assigned to be projected by a different node; or 2) it overlaps the screen areas of two nodes; one of them must be selected to project it, and a copy of the resulting tile must be sent to the other. The transfers resulting from reason 1 are those which we expect can be decreased when the block replication factor is high.

Figure 6.14 shows the average number of bytes communicated as a result of tile

Female dataset, four nodes:

lb threshold	512/512 cached	256/512 cached	128/512 cached
.65	**6,018,071**	**9,032,861**	**11,382,769**
.80	6,079,764	9,033,617	11,382,769
.90	6,335,405	9,105,326	11,390,147
.95	**6,570,445**	**9,233,823**	**11,341,547**

Male dataset, four nodes:

lb threshold	128/128 cached	64/128 cached	32/128 cached
.65	4,362,365	5,552,798	6,589,883
.80	4,368,541	5,553,604	6,591,031
.90	4,441,437	5,640,588	6,708,717
.95	4,556,869	5,902,784	6,694,678

Figure 6.14: Average tile transfer bytes per frame on four nodes of eight processors

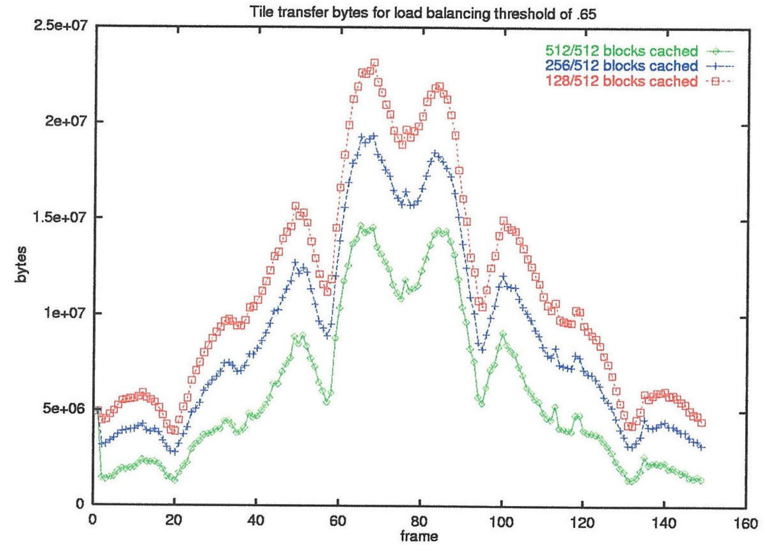
transfers. There is a clear pattern of increasing tile transfers as replication factor decreases from left to right, which is to be expected due to the flexibility of assignment associated with higher replication factors.

In addition, there is a somewhat weak tendency for increasing load balance factor to require more tile transfers. This tendency is more pronounced for the male dataset, which has lower granularity than the female dataset; this results in less stochastic load balance, yielding less "excess" load balance above a given threshold which can be applied to improving tile assignments.

The **boldface** rows from Figure 6.14 are detailed in Figure 6.15. The features to be noted are: 1) that for every frame (not just on the average), a higher replication factor yields fewer bytes of tile transfers; and 2) that tile transfers are not very volatile, even with a load balance factor of .95, because they are the last priority of the load balancing algorithm.

Figure 6.16 shows tile transfers for two nodes. Figure 6.17 shows tile transfer bytes for eight nodes on the full-female dataset. The same general trends we noted in Figure 6.14 appear.

In summary, there is a strong tendency for decreasing replication factor to yield



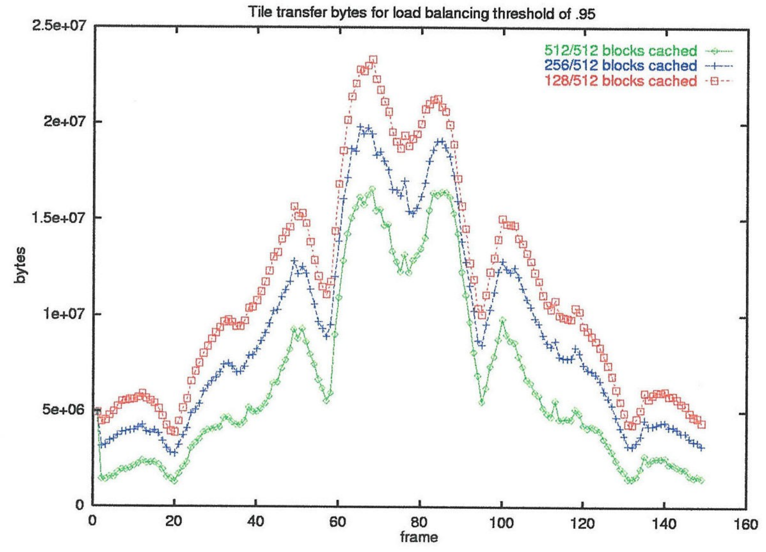


Figure 6.15: Tile transfer bytes per frame on four nodes of eight processors, female dataset

Female dataset, two nodes:

lb threshold	512/512 cached	256/512 cached	128/512 cached
.65	1,742,831	5,143,475	5,438,966
.80	1,760,193	5,143,475	5,438,966
.90	2,018,093	5,143,475	5,438,966
.95	2,239,843	5,095,816	5,454,861

Male dataset, two nodes:

lb threshold	128/128 cached	64/128 cached	32/128 cached
.65	1,804,691	3,201,983	3,311,906
.80	1,804,692	3,155,543	3,315,993
.90	1,832,170	3,019,116	3,364,375
.95	1,893,350	2,924,169	3,398,617

Figure 6.16: Average tile transfer bytes per frame on two nodes of eight processors

Full-Female dataset, eight virtual nodes:

lb threshold	256/2048 cached
.65	25,420,754
.80	25,420,754
.90	25,442,592

Figure 6.17: Average tile transfer bytes per frame on eight virtual nodes of four processors

increasing tile transfers, and a weaker tendency for increasing load balancing factor to require increased tile transfers.

The tradeoff between replication and tile transfers

Figure 6.18 compares replication factors of one and four, on four nodes. Each quadrant of the screen represents the area assigned to one of the four nodes for composition. (Each of the eight white squares shows the area assigned to one processor; there are two processors on each of the four nodes.) The colored squares indicate tiles projected from their corresponding volume blocks by a particular node (each of the four colors corresponds to a specific node). The left side of the figure shows a replication factor of one, and the right side shows a replication factor of four. On the right side, due to the high replication factor, most blocks can be assigned for projection to the same nodes which will need them for composition. On the left side of the figure, the low replication factor requires that most blocks be assigned for projection to nodes other than those that will require them for composition (specifically, to the nodes which hold a copy of them). The situation on the left side of the figure, with low replication factor, entails many more tile transfers than the situation on the right side of the figure, with high replication factor.

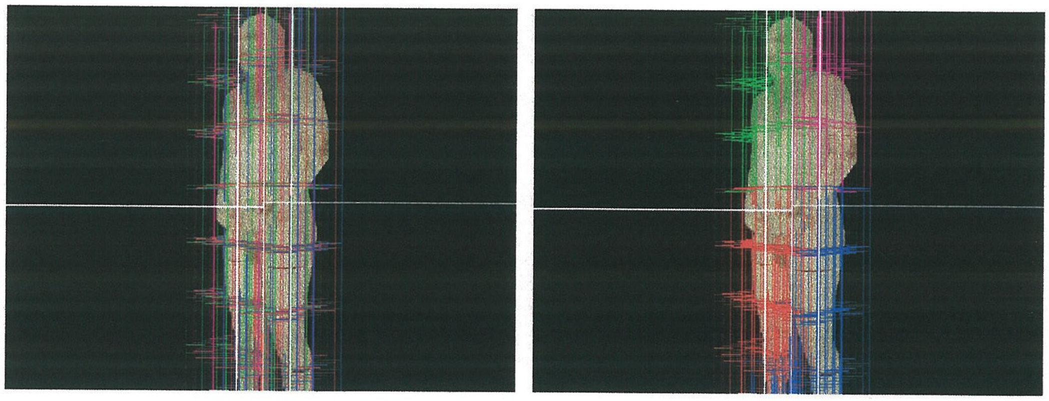


Figure 6.18: Assignments of blocks to nodes for projection, for replication factors of one (left) and four (right)

6.4.5 Frame times

The ultimate measure of the success of choosing a particular load balancing factor, and of the consequences of using a low replication factor, is the frame rate achieved. All the tradeoffs between load balance, two types of communication, and replication factor, can be judged by the metric of the resulting frame rate.

Female dataset, four nodes:

lb threshold	512/512 cached	256/512 cached	128/512 cached
.65	1.615	1.652	1.723
.80	1.613	1.676	1.716
.90	1.619	1.668	1.705
.95	1.638	1.701	1.837

Male dataset, four nodes:

lb threshold	128/128 cached	64/128 cached	32/128 cached
.65	1.229	1.258	1.261
.80	1.243	1.267	1.300
.90	1.220	1.282	1.395
.95	1.233	1.346	1.499

Figure 6.19: Average frame times on four nodes of eight processors (sec)

Figure 6.19 shows the average frame rates achieved on four nodes for the female and male datasets for our twelve combinations of parameters. The primary feature of interest is that as we move from left to right along each row, the penalty for using a low replication factor is not large. In other words, our load balancing algorithm and caching system function very effectively to exploit the memory locality present in ray casting. There is some benefit to be had in complete replication, but our algorithm degrades smoothly as the replication factor is decreased, and does extremely well with no replication at all.

The second feature of interest is that frame rates are, in general, superior with a low load balancing factor, particularly when the replication factor is low. The granularity of both datasets is high enough to make this the case. We would expect a higher load balancing threshold to be beneficial: 1) as the granularity became lower, or 2) as the cost of communication decreased. We will show in Section 6.4.6 that the HIPPI performance is well below the peak of 91 MB/sec cite which SGI claims [Skibo95].

Female dataset, two nodes:

lb threshold	512/512 cached	256/512 cached	128/512 cached
.65	2.500	2.624	2.636
.80	2.503	2.624	2.641
.90	2.490	2.621	2.653
.95	2.492	2.665	2.712

Male dataset, two nodes:

lb threshold	128/128 cached	64/128 cached	32/128 cached
.65	1.726	1.829	1.844
.80	1.730	1.833	1.831
.90	1.714	1.859	1.862
.95	1.731	1.886	1.950

Figure 6.20: Average frame times on two nodes of eight processors (sec)

Figure 6.20 shows that the same patterns hold for two nodes: our algorithm effectively manages locality to keep the penalty for no replication quite low.

Figure 6.21 shows the frame times for the full-female dataset. Since the full-female dataset was so large, we render it at a replication factor of one (1) only. However, as expected, the figure shows that for the extremely high granularity of the full-female dataset, high load balancing factors result in slower frame times due to load balance thrashing.

In summary, our load balancing algorithm and caching system function very effectively to exploit the memory locality present in ray casting. There is some benefit to be had in complete replication, but our algorithm degrades smoothly as the replication factor is decreased, and does extremely well with no replication at all.

Full-Female dataset, eight virtual nodes:

lb threshold	256/2048 cached
.65	4.543
.80	4.539
.90	5.196

Figure 6.21: Average frame times on eight virtual nodes of four processors (sec)

6.4.6 Parallel speedup

On a single node, it took 4.291 seconds on average to render a frame for the female dataset, and 2.827 seconds for the male dataset. Note that, for a single node, neither load balance factor (i.e., the load balance factor among parallel nodes) nor replication factor are relevant parameters.

Figure 6.22 shows parallel speedup on two and four nodes, compared to the times to render a frame on one node. While our speedups are less than sometimes expected for algorithms that do not require a global gather operation, ray casting does require this operation. Speedup will never be perfect unless the communication is free.

Note that, on a shared-memory architecture, our jump from one to sixteen processors did not require the addition of an extra level of the memory hierarchy: both a single processor and sixteen processors had to communicate with memory over the same bus. We were able to achieve good parallel speedup until the capacity of this bus ran out (up to eight processors for the poor cache performance of the image partition, and up to sixteen for the better cache performance of the object partition). When jumping from one to multiple distributed nodes, a new level of the memory hierarchy is created that did not exist on a single node. Parts of the global gather operation in ray casting must travel over the new, slower channel, ruling out perfect parallel speedup.

We can derive the effective HIPPI bandwidth achieved, in the context of heavy contention during the many-to-many transfers required during a frame, by dividing the total number of bytes sent for tile and block transfers, by the total time spent

Female dataset:

	512/512 cached	256/512 cached	128/512 cached
speedup on 2 nodes	1.72	1.64	1.63
speedup on 4 nodes	2.66	2.60	2.49

Male dataset:

	128/128 cached	64/128 cached	32/128 cached
speedup on 2 nodes	1.64	1.55	1.53
speedup on 4 nodes	2.30	2.25	2.24

Figure 6.22: Parallel speedup over single node (percentage of number of nodes)

sending or receiving those bytes. For blocks, HIPPI achieved an effective bandwidth per node of 9.6 MB/sec; for tiles, HIPPI achieved an effective bandwidth of 8.7 MB/sec. This is well below its 91 MB/sec peak rate.

Since we know the details of our communication costs, we can make the prediction that if the HIPPI network were five times faster on average (i.e., achieving on average approximately half of its 91 MB/sec peak), then, with no replication, we would attain 3.0 times speedup on four nodes for the female dataset, and 2.5 times for the male dataset.

6.4.7 Relative costs of computation and communication

Figure 6.23 shows the relative costs of projection, composition, block transfer, and tile transfer, for two typical experiments: the female dataset, with no replication, for load balancing thresholds of .65 and .95. These costs are all summed over the four nodes. In real time, there was high overlap among the four nodes for each of these components; therefore the wall clock time of each component was approximately one-fourth as large.

The primary computational cost is projection. Composition, even for this large number of tiles (512), represents a tiny fraction of the cost of projection.

For a load balancing threshold of .65, there are no block transfers; the entire communication cost is due to tile transfers. At the midpoint of the test suite, when there are few large tiles on the screen, this communication cost approaches the computational cost of projection. For a load balancing threshold of .95, there are many block transfers, particularly about the midpoint of the test suite, where the costs of all the blocks change dramatically. The cost of block transfer, in wall time, is bursty, sometimes surpassing both the cost of tile transfers, and the cost of projection.

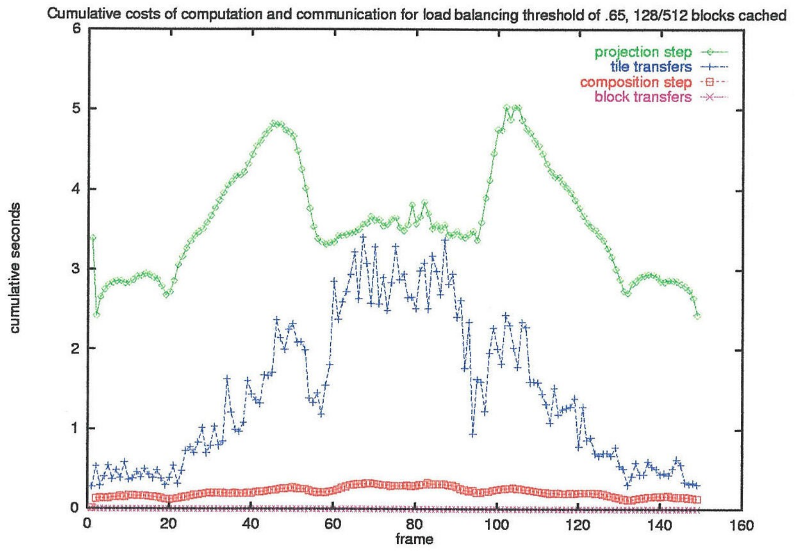
6.4.8 Performance on a very large dataset

Wanting to push the limits of our system, and take full advantage of its ability to efficiently render without replication of data, we attempted to achieve the maximum frame rate on the full-female dataset, which at 7.1 GB is the largest single volume dataset yet rendered. We doubled the number of processors to eight, on each of the eight virtual nodes. With no replication, and a load balancing factor of .65, at a resolution of 800x600 pixels, we achieved an average of 3.4 seconds per frame for the *typicalview* suite, using no dataset-specific optimizations. When we added two dataset specific algorithmic optimizations (opacity clipping and the discarding of completely empty volume blocks), we achieved a rate of 2.1 seconds per frame on average. This is very high performance relative to the dataset size and image resolution.

6.5 Conclusions

We have used lessons from previous chapters — specifically that using blocks of 0.5 and 1.0 times the size of the cache maximizes cache performance, that roughly equilateral blocks and tiles are superior, and knowledge about effective parallel partitioning within a single node — and used them to prune our choices of parallel partitioning and load balancing on a higher level of the memory hierarchy. We determined that, for typical image resolutions, it is cheaper to achieve the global gather operation required by ray casting by moving the image tiles than moving the volume blocks.

This led us to the following ordered set of priorities for our load balancing algorithm: 1) keep the estimated load balance above a certain threshold, 2) avoid the transfer of volume blocks, 3) assign blocks for projection to a node that will require



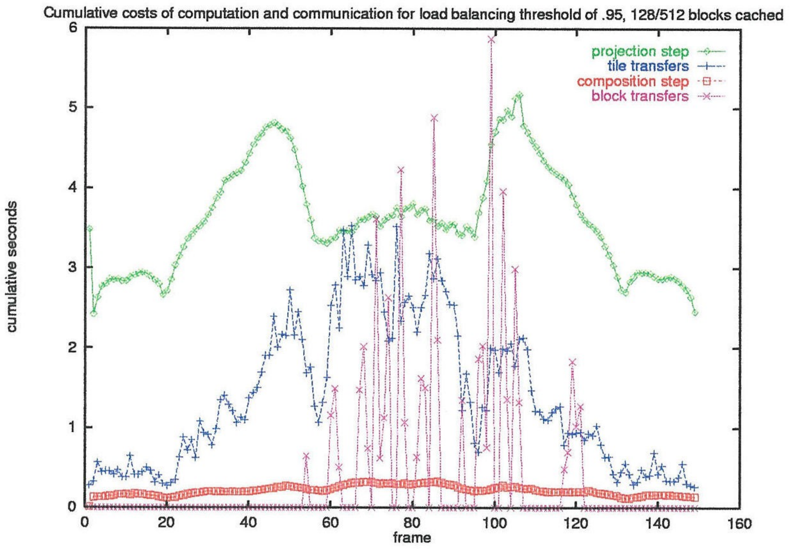


Figure 6.23: Relative costs of computation and communication on four nodes, female dataset (sec)

the associated tile for composition (thereby avoiding the transfer of tiles). We implemented this set of priorities in a tiered greedy algorithm, which functioned effectively to exploit locality. The cost for rendering at a replication factor of 1 (no replication) was only slightly more than rendering with complete replication (a factor of 2 or 4 on two and four nodes, respectively). This is possible despite the global gather operation required by ray casting; and possible because ray casting contains sufficient coherence, which is effectively exploited by our algorithm to produce memory locality.

The volume of bytes communicated due to block transfers for load balance, in general, tends to increase: 1) as we demand a higher load balancing threshold; and 2) as the replication factor is decreased, allowing less flexibility in the assignment of blocks to meet a given load balancing threshold.

Attainable load balance depends on many factors: interaction of granularity, replication factor, the competing goals of the load balancing algorithm (requiring that estimated load balance be above a threshold, avoiding block transfers, and improving tile assignments), and the accuracy of our predictions of future block costs.

High granularity yields benefits in term of stochastic load balance: High replication factors allow the algorithm to use this to improve load balance, or to decrease communication of tiles; low replication factors do not have this flexibility, and can only receive these gains as improved load balance.

Low granularity yields worse stochastic load balance. High replication factors can draw on their flexibility to maintain load balance and relatively low communication of tiles, whereas low replication factors perform less well.

With no replication, we attain speedup on four nodes of 2.48 times on the female dataset, and 2.24 times on the smaller male dataset. There is more communication relative to computation for the smaller dataset. In the context of heavy network contention, HIPPI attains on average 8.7 and 9.6 MB/sec for blocks and tiles, respectively; well below its peak performance of 91 MB/sec. If HIPPI attained half its peak performance on average, we would expect, without replication, 3.0 times speedup on four nodes for the female dataset, and 2.52 times for the male dataset. As internode communications costs drop to zero, we can increase the required load balancing threshold with a vanishing communications cost. The improved load balance together with decreased communications cost would improve parallel efficiency.

Our methods allowed us to exploit memory locality, and thereby to pay only a small cost for no replication of data. This allowed us to efficiently render a dataset larger than the individual memory of any node. We attained a frame rate of 2.1 seconds per frame on this 7.1 GB dataset at a resolution of 800x600 pixels, including dataset-specific optimizations; and 3.4 seconds per frame without such optimizations. This is the largest dataset yet rendered. Rendering this dataset, and rendering it at a reasonable frame rate, was made possible only by the efficient exploitation of many levels of the memory hierarchy of the hybrid architecture of the Power Challenge Array.

The results of Chapter 6 can be found summarized in [Palmer97b].

Chapter 7 Related Work

7.1 Closely related papers

7.1.1 Lower levels of the memory hierarchy

In [Challinger91], it was noted that using a task size of one pixel as the unit of parallelism resulted in worse performance than a task size of one scan line. The author hypothesized that this was due to less efficient use of the processor cache in the former, but did not explore this in depth.

Papers by a group of colleagues at Stanford University, [Lacroute95, Singh94, Nieh92], did consider the lower level memory hierarchy characteristics of volume rendering. In [Nieh92] and [Singh94], which used the same volume rendering code), however, cache miss penalties were not found to be the main cost component of the algorithm, and the strong directional dependence of caches miss rates was not identified. We suspect two reasons for this: 1) we systematically searched the sphere for the worst viewpoint directions – the variation could otherwise be easily overlooked; and 2) we spent a considerable effort in optimizing our inner loop for the R8000: this involved tuning the source code to allow the compiler to efficiently software pipeline loops, yielding a performance gain of three or four times. This factor of improvement was due to more efficient scheduling of instructions, not actual reductions in numbers of memory accesses, or significant changes in the order of memory accesses. It is to be expected that the cost of memory accesses would become the largest remaining cost.

[Singh94] contained cache simulation results, and identified, for a 256^3 dataset, an "important working set" (or optimal cache size) of 16K. The implication is that making the cache any larger is not useful. This work did not discover the the utility of a large cache like the 4MB L2 cache of the Power Challenge, or investigate the qualitatively different performance effects of a cache large enough to contain an entire volume block. We found that caches of this size could either perform very effectively or very poorly (both in terms of number of cache misses, and in terms of the cost of memory system delays), depending on whether or not we actively took steps to improve memory locality (by appropriate blocking).

Memory hierarchy costs were pointed to in [Lacroute95] as a primary cost component, but it was difficult to further improve the locality of the shear-warp factorization algorithm used for rendering.

7.1.2 Higher levels of the memory hierarchy

[Funkhouser92] discussed data management for interactive architectural walkthroughs, using a predictive model to load from disk the parts of an architectural database which would soon come into the view of the user of the system.

Two papers coauthored by Corrie and Mackerras, [Corrie92] and [Mackerras94], implemented a global distributed caching system for volume blocks on the Fujitsu AP1000. They used an image partition, which required much communication of volume blocks. They distinguished between a *persistent list* of blocks, and a *cache list*; the former were fixed in the memory of a given node, while the latter were temporary replicated copies. They found that performance had a directional dependence, but analyzed this in terms of the top level of the memory hierarchy only. In order to avoid the communication of the entire dataset each frame, as would be required by an image partition without replication, they replicated the dataset in what they term *neighborhoods*. Each neighborhood of processors would own one entire copy of the data. In the extreme case, they placed each processor in its own neighborhood, with a complete copy of the data. We believe, for the reasons outlined in section 6.1.1, that a method that involves moving the volume blocks in order to accomplish the global gather operation is inherently less efficient that one involving the movement of image tiles. However, this work was quite innovative in its implementation of a software block caching system. In addition, this work recognized that square tiles are superior to elongated ones, citing their superior memory locality characteristics. [Corrie92] briefly mentions memory hierarchy effects at a lower level, but does not explore them in depth.

On the Cray T3D, [Law96a] recently implemented a distributed block caching and replication scheme which was a "cache only" memory system, like our system. That is, it allowed volume blocks to migrate and replicate dynamically among the nodes, without a "master copy" of any block. On 128 nodes, they found that replication factors of up to four and six, for 256^3 and 128^3 datasets, respectively, yielded significant performance benefits. In [Law96b], these authors addressed the problem of "thrashing" of volume blocks among nodes with an "advancing ray front" method, which increases memory locality by advancing a plane of rays through the set of volume blocks that are currently in local memory. They also extended this method to be "thrashless" across a sequence of similar frames. In [Law96c], the same authors used disk storage to deepen the memory hierarchy to render very large datasets. They stored blocks of data in compressed form on disk.

7.2 Performance comparison

The following tables contain maximum performance claims from a number of parallel volume rendering papers. The rows of the tables contain the volume dataset size, the image resolution, and the time (in seconds) required to render a frame; the tables for RendAsunder also note the test suite used. Note that RendAsunder has the highest frame rates for datasets of 128 MB and more, and is the only implementation to render a dataset larger than 1 GB (i.e., 7.1 GB). The two times shown for the 7.1 GB dataset are with, and without, dataset-specific optimizations.

[Schröder91], CM-2, 64K proc, 1-D shears

128^3 (2 MB)	128×128	.502 sec

[Nieh92], Stanford DASH, 48 proc, ray casting

$256 \times 256 \times 226$ (14 MB)	416×416	.34 sec

[Mackerras94], Fujitsu AP1000, 1024 proc, ray casting

$512^3 \times 4$ bytes (512 MB)	256×256	315 sec

[Wittenbrink94], Proteus, 32 procs, permutation warp

256^3 (16 MB)	256^2	1.4 sec

[Karia94], Fujitsu AP1000, 128 proc, ray casting

$256 \times 256 \times 113$ (7 MB)	512^2	33 sec

[Singh94], SGI Challenge, 16 proc, ray casting

256^3 (16 MB)	416×416	.25

[Lacroute95], SGI Challenge, 16 proc, shear-warp

128^3 (2 MB)	256^2	0.032 sec (grayscale)
256^3 (16 MB)	256^2	0.077 sec (grayscale)

[Westermann95], CM-5, 64 proc, ray casting

256^3 (16 MB)	256^2	6.7 sec
512^3 (128 MB)	512^2	55 sec
1024^3 (1 GB)	1024^2	445 sec

[Law96a], Cray T3D, 128 proc, ray casting

256^3 (16 MB)	512^2	0.25 sec

RendAsunder, SGI Power Challenge, 16 procs, ray casting

512^3 (128 MB)	400×300	0.34 sec	axisorbit
$584 \times 1878 \times 341$ (357 MB)	400×300	0.53 sec	axisorbit
$840 \times 2595 \times 480$ (1 GB)	400×300	1.0 sec	axisorbit

RendAsunder, SGI Power Challenge Array, 8×8 procs, ray casting

$584 \times 1878 \times 341$ (357 MB)	400×300	0.10 sec	fly through

RendAsunder, SGI Power Challenge Array, 4×16 procs, ray casting

$1610 \times 894 \times 5192$ (7.1 GB)	800×600	2.1 sec	typicalview, w/ opts
$1610 \times 894 \times 5192$ (7.1 GB)	800×600	3.4 sec	typicalview, no opts

Chapter 8 Conclusions

8.1 Results and implications for ray casting volume rendering

In this thesis, we have analyzed methods to optimize the performance of all levels of a deep memory hierarchy, in the context of ray casting volume rendering through regular volumetric data. We have analyzed in detail the memory hierarchy effects present in single-processor, shared-memory, and distributed-memory ray casting.

On a single node, using 1) algorithmic modifications to isolate cache miss costs, 2) bus monitoring hardware, and 3) a software cache simulator, we have identified and characterized the main expense in a ray casting kernel with highly efficient instruction scheduling: cache miss penalties due to reads of the dataset voxels, in the context of extreme demands on the memory system. We have shown that this expense is extremely dependent on view direction. We have also demonstrated, however, that this dependence on view direction can be controlled by appropriate blocking of the data. We have isolated the separate L1 and L2 cache miss components contributing to this expense. Complex interpixel and intrapixel effects account for the majority of the dependence of miss rates upon view direction. These in turn depend on the relative orientations of the V and W axes of the viewer to the X axis of the dataset. Interblock and interframe cache effects are negligible, because of the high rates at which the cache is completely flushed.

Grouping volume data into roughly cubical blocks, and pixels into roughly square tiles, offers superior memory locality, and therefore superior memory hierarchy performance at all levels, to flattened slabs, or scan lines, respectively. The optimal block size for varied datasets is between 0.5 and 1.0 times the size of the L2 cache.

On a shared-memory multiprocessor, we have examined the fundamental relationship between parallel partitioning the memory hierarchy performance at all levels, studying one object partition, and one image partition. The object partition is generally superior because of the natural blocking of the data that it affords, and the relative directional independence of its performance. The image partition allows global opacity clipping, but the benefit of this algorithmic optimization is dataset dependent. We obtain good load balance on both partitions, and good parallel speedup on the partition with better cache performance.

We demonstrate that high rates of L2 cache misses, such as those that occur during ray casting through a single large block, can cause bus saturation of the Power Challenge when using more than eight nodes. This bus saturation compounds the performance degradation of the poor performance at the L2 cache level. This makes the impact of cache performance considerations on algorithm design on such machines even more important.

On a cluster of distributed nodes, we have considered a number of parallel partitioning and data replication options in terms of the constraints of achieving good memory hierarchy performance at all levels. We have explored the tradeoff between communication cost and memory expenditures for data replication.

We have implemented in software a global logical address space for the volume blocks, complete with caching, and developed a caching/load balancing method which balances the competing goals of load balance, minimizing communication of volume blocks, minimizing communication of projected tiles, and minimizing memory use. We have shown that this system performs extremely well with no data replication at all: relative to the N-fold replication of the data on N nodes, only a small performance penalty must be paid.

Our parallel speedup across multiple nodes (2.6 times on four nodes, 1.65 times on two nodes) was less than is commonly expected for algorithms that do not require global communication as an integral part of the algorithm; however, volume rendering does require this global communication, and our parallel speedup is in accordance with that achieved in other research. We predict that, were the HIPPI network capable of four times its measured average bandwidth, under contention, of 10MB/sec, we would achieve 3.0 times speedup on four nodes.

Our focus on the efficient exploitation of all levels of the deep memory hierarchy has allowed us to break several performance records, including the fastest recorded time for 357 MB to 1.0 GB dataset, and the first rendering of a 7.1 GB dataset.

We do not expect that the next step for ray casting volume rendering will be to render a yet larger regular dataset. Figure 8.1 illustrates why. The projected image of dataset of 1600x900x5200 pixels, at full resolution, contains more information than can be displayed in an image of typical resolution, e.g., 800x600. Many voxels in such a dataset would simply fall between the rays generated by the array of pixels, unless the view is zoomed in close to the data (in which case many other voxels are out of view). The transfer of these missed voxels represents a waste of memory hierarchy bandwidth. More efficient would be the multiresolution replication of data: the data is loaded at full resolution, and copies at (e.g.,) 1/2 and 1/4 resolution are automatically generated. Rays diverge at a constant rate, and it would be inexpensive to dynamically choose the copy of the dataset at the appropriate resolution for the current distance between the rays. This idea has been explored in [Laur91, Danskin92], but was not understood in terms of efficient use of memory hierarchy bandwidth.

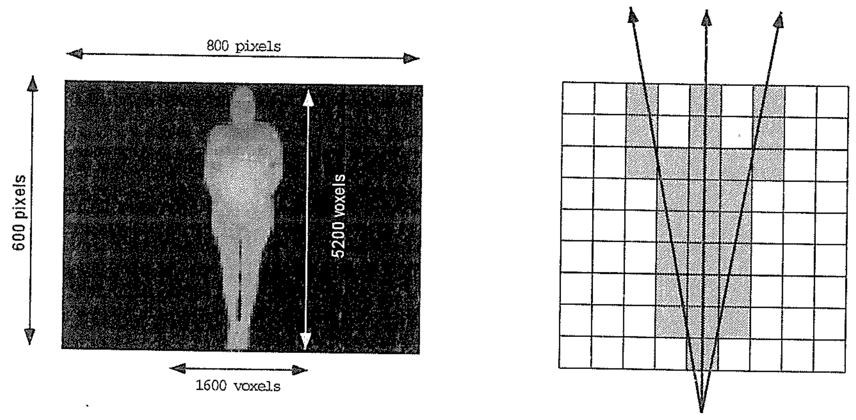


Figure 8.1: The impact of ray divergence

8.2 Implications for other parallel algorithms

Ray casting volume rendering is only one example of a parallel algorithm that requires a global gather operation.

Consider the extreme case of a global sum: each node of a Power Challenge Array contains a large set of integers; we wish to know the global sum of all of them. In this case, the most straightforward method happens to yield optimal memory hierarchy performance: At the top level of the memory hierarchy, each node sums its own integers; then the N resulting integer values from each node are sent to a central location and summed. At the lower levels of the memory hierarchy, each individual node will sum its own integers in the order they are stored in memory.

We need only make the example slightly more complicated for memory hierarchy effects to dominate the algorithmic design: for example, take a large two-dimensional array, stored in contiguous, uniform squares on N nodes; we want to find the sums of the rows and columns of the array. The optimal order in which to perform operations is now more complex: values will be used more than once, and we would like to read each value from main memory as few times as possible. Within a node, we would use blocking to improve local cache performance. We would find the sums of rows and columns for individual nodes, and send these results to a central location where the partial results could be combined.

As a third example, imagine that the array is not stored contiguously, but that its elements are dealt out in round-robin fashion to the nodes. The optimal order of operations and communications pattern would now depend on many things, including the relationships between the width of the array, the number of nodes, and the sizes of the cache lines. In addition, if the width of the array were not an integer multiple of the number of nodes, we might profitably pad the array with extra columns of zeros, to increase locality down the columns.

The experimental results in this thesis, and the analytical models they support, provide a useful framework to analyze the parallel memory hierarchy performance of other problems in which: 1) the operator which combines partial results is associative,

2) the entire distributed dataset contributes to a solution that will end up stored on a single node, and 3) there exists coherence which can be exploited to increase memory locality.

This class of problems requires global communication. Furthermore, the amount of communication increases with increasing numbers of processors, making them strictly not scalable algorithms. However, they contain coherence that allows careful algorithmic design to increase memory locality.

8.3 Implications for parallel architecture design

Ray casting volume rendering is perhaps not best suited to MIMD processors, connected by relatively slow interconnection networks. This class of problems is better suited to parallel machines employing a fast, shared bus, but these machines can only scale to a low number of processors before the bus becomes saturated. The Power Challenge Array, a hybrid distributed/shared memory architecture, was an attempt to go beyond this number of processors by connecting multiple shared-memory nodes. However, the decision to use a slow but "scalable" HIPPI crossbar network limits the achievable parallel speedup. A better means to scale to a moderate number (hundreds) of total processors would be to connect multiple shared-memory nodes with a network designed for maximum speed between a limited number of nodes. A logical global address space, with caching built into hardware, would simplify implementation. Ideally, there would be provisions for the programmer to give "hints" to the automatic cache, indicating which blocks should remain in local memory. This type of architecture would provide good peak performance for any problem that satisfies conditions 1 and 2 in the previous section. It would allow more efficient exploitation of memory locality for problems that contain coherence.

Such an architecture has already been designed by Silicon Graphics: the recently released Origin 2000. The findings in this thesis validate many of the design choices in the Origin 2000. Attaining peak performance on this architecture would still be difficult, requiring careful attention to cache optimization at multiple levels, but, when this care is taken, this architecture is likely to be capable of much better parallel speedup than either a hybrid architecture with a slow interconnect, like the Power Challenge Array, or a true "scalable" distributed-memory architecture. Since the communication in this class of problems increases as the number of nodes increases, it is not indefinitely scalable. However, on moderate numbers of nodes, these problems contain coherence which can be used to increase memory locality; doing this effectively requires a deep memory hierarchy.

Appendix A A fine-grained MIMD implementation with no replication

Prior to our work on the Power Challenge, we implemented a volume rendering system for irregular tetrahedral data on the Intel Delta machine, using projection methods. This system was unsuccessful at achieving interactive performance: we attained rates of 30 seconds per frame on 512 processors. (Note that irregular volume rendering can be expected to be somewhat slower than regular volume rendering for the same number of volume elements.) However, the system was innovative. In a desire to go against the conventional reasoning in the literature, we chose to use an image partition on a relatively fine-grained MIMD machine, and did not replicate the data. At each frame, every node was assigned a square region of the image, using the recursive binary image subdivision. To render its region of the screen, each node required a copy of every tetrahedral voxel overlapping that region. The tetrahedral voxels did not have a fixed location, but moved dynamically about the machine. We guaranteed (with a distributed algorithm) that at all times there would be at least one copy of each voxel somewhere on the machine. When a voxel overlapped a boundary between two processors' screen regions, more than one copy of that voxel would be temporarily created.

Communication (predictably, in hindsight) was the limiting factor of the algorithm. As the viewpoint changed, large volumes of the dataset would need to be communicated. Desire to minimize this led us to invent a novel parallel partitioning method: the *rotation-invariant partition*, which is intended to minimize communication caused by head rotations of the observer. [Palmer94b]

This partition is illustrated in Figure A.1. The figure shows three representations of the same distribution of partitioning lines. On the left of the figure is a view of the display screen. The screen is recursively divided by three sets of lines labeled A1,

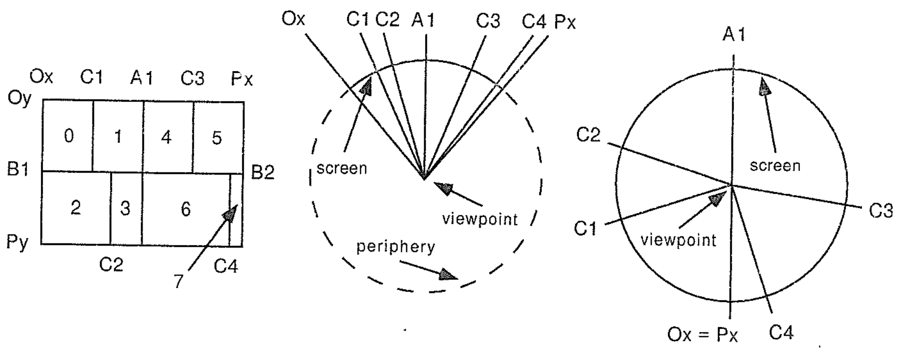


Figure A.1: Three views of a set of partitioning lines

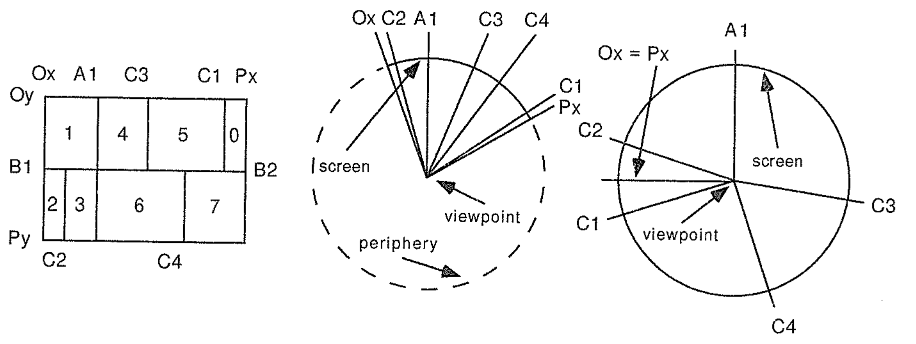


Figure A.2: Three views of the lines after a head rotation

B1, B2, and C1, C2, C3, C4. Ox and Px are the left and right boundaries of the screen in the X direction; Oy and Py are the boundaries in the Y direction. Each of the vertical lines, i.e., A1, C1, C2, C3, C4, Ox, Px, corresponds to a plane cutting through object space. In the center of the figure is a view from above the observer. The display screen appears as a solid arc. The viewpoint of the observer is in the center of the circle. Each of the cutting planes previously mentioned is shown here as a line radiating from the viewpoint. The dashed arc passes through the volume of image space which is currently in the periphery of the observer's field of view. On the right side of the figure, Ox and Px have been stretched around the circle until they become a common point, and the periphery has been spliced out; the screen has been mapped onto a torus (the mapping of the Y direction is not illustrated in the figure).

Figure A.2 shows what happens after a head rotation of 20 degrees to the right. Note in the center of the figure that Ox and Px have moved 20 degrees clockwise around the circle; Lines A1, C2, C3, and C4 have remained stationary, but line C1 has scrolled off the left side of the screen and wrapped around to the other side. The left and right sides of the figure show alternate views of the same set of line positions. Instead of moving the voxels, we have rotated the entire image partition beneath them on the torus; this eliminates the need to move many of the voxels.

If the screen fills a V degree field of view (in Figures A.1 and A.2, the angle between Ox and Px is 80 degrees), then, during a head rotation of 360 degrees, each processor will scroll across the screen from left to right $\frac{360}{V}$ times.

To make the load balancing lines follow the bulk movement of the data at each frame, the partitioning lines are translated on the screen some vector T. The T we use approximates how much the center of mass of the currently visible data elements has moved in between frames. We find the center of mass of all data elements for the last frame, matrix-multiply it by the difference between the current and last view matrices, and discard the Z coordinate. We then translate the origin of the image partition by this vector on the torus.

These partitions are designed to eliminate communications costs due to head rotations about the X and Y axes. Communication incurred by panning (X or Y pure translation) is not completely eliminated by this technique, but is reduced, especially for elements that are far from the observer.

This partitioning method did reduce overall communication compared to the ordinary image partition. However, this unconventional method still required the communication of many volume blocks as an integral part of ray casting. Without the benefits of caching, this method was still communication-bound.

Bibliography

[Ackerman95] The Visible Human dataset is publicly available from the Visible Human Project, c/o Dr. Michael J. Ackerman, National Library of Medicine.

[Amin95] M. Amin, A. Grama, and V. Singh. "Fast Volume Rendering Using an Efficient, Scalable Parallel Formulation of the Shear-Warp Algorithm." *Proceedings of the 1995 Parallel Rendering Symposium*, pp. 7-14. ACM Press, October, 1995

[Arvo87] J. Arvo, D. Kirk. "Fast Ray Tracing by Ray Classification." *ACM Computer Graphics* 21(**4**):55-64, July, 1987.

[Arvo88] J. Arvo, D. Kirk. "A Survey of Ray Tracing Acceleration Techniques.", in *An Introduction to Ray Tracing*, Andrew Glassner, Editor. Academic Press, Harcourt Brace Jovanovich, Publishers, New York, 1989.

[Badt88] S. Badt, Jr. "Two Algorithms for Taking Advantage of Temporal Coherence in Ray Tracing." *The Visual Computer* (1988), 3:123-131, Springer-Verlag, 1988.

[Blinn82] J. Blinn. "Light Reflection Functions for Simulation of Clouds and Dusty Surfaces." *ACM Computer Graphics*, 16(**3**):21-29. 1982.

[Cabral94] B. Cabral, N. Cam, J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," in *Proc. 1994 Symp. Volume Visualization*, Washington, D.C., pp. 91–98, Oct. 1994.

[Challinger91] J. Challinger. "Parallel Volume Rendering on a Shared-Memory Multiprocessor." Technical Report UCSC-CRL-91-23, Board of Studies in Computer and Information Sciences, University of California at Santa Cruz, March 1992.

[Cook84] R. Cook, R. Porter, L. Carpenter. "Distributed Ray Tracing." *ACM Computer Graphics*, 18(**3**):137-145. 1988.

[Corrie92] B. Corrie, P. Mackerras. "Parallel Volume Rendering and Data Coherence on the Fujitsu AP1000." *Department of Computer Science, The Australian National University*, Tech Report TR-CS-92-11, August,1991.

[Cleary86] J. Cleary, B. Wyvill, G. Birtwistle, R. Vatti. "Multiprocessor Ray Tracing." *Computer Graphics Forum*, 5:3-12. North-Holland. 1986.

[Cox93] M. Cox, P. Hanrahan. "Pixel Merging for Object-Parallel Rendering: a Distributed Snooping Algorithm." *Proceedings of the 1993 Parallel Rendering Symposium*, pp. 49-56. ACM Press, October 1993.

[Danskin92] J. Danskin, P. Hanrahan "Fast Algorithms for Volume Ray Tracing." *Proceedings of the Workshop on Volume Visualization*, pp. 91-95. ACM Press. October, 1992.

[Dippé84] M. Dippé, J. Swensen. "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis." *ACM Computer Graphics* 18(**3**):149-158, 1984.

[Drebin88] R.A. Drebin, L. Carpenter, P. Hanrahan. "Volume Rendering." *ACM Computer Graphics* 22(**4**):65-74, 1988.

[Fuchs89] H. Fuchs, et al. "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories." *ACM Computer Graphics* 23(**3**):79-88. 1989.

[Funkhouser92] T. Funkhouser, C. Séquin, S. Teller. "Management of Large Amounts of Data in Interactive Building Walkthroughs." *Proc. ACM SIGGRAPH '92*, pp. 11-20. Boston, March, 1992.

[Glassner84] A. Glassner. "Space Subdivision for Fast Ray Tracing." *IEEE Computer Graphics and Applications,* 4(**10**):15-22, October, 1984.

[Hanrahan90] P. Hanrahan. "Three-Pass Affine Transforms for Volume Rendering". *Computer Graphics*, 4(**5**):71-77, November 1990.

[Kajiya84] J. Kajiya, B. Von Herzen. "Ray tracing volume densities." *ACM Computer Graphics* 18(**3**):165-174. 1984.

[Karia94] R.J. Karia. "Load Balancing of Parallel Volume Rendering with Scattered Decomposition." *Proceedings of the 1994 Scalable High-Performance Computing Conference*, pp. 252-258.

[Kay86] T. Kay, J. Kajiya. "Ray Tracing Complex Scenes." *ACM Computer Graphics* 20(**4**):269-278. August, 1984.

[Kobayashi87] H. Kobayashi, T. Nakmura, Y. Shigei. "Parallel processing of an object space for image synthesis using ray tracing." *The Visual Computer* (1987), 3:13-22, Springer-Verlag 1987.

[Kobayashi88] H. Kobayashi, S. Nishimura, K. Kubota, T. Nakamura, Y. Shigei. "Load balancing strategies for a parallel ray-tracing system based on constant subdivision." *The Visual Computer* (1988), 4:197-209, Springer-Verlag 1988.

[Lacroute94] P. Lacroute, M. Levoy. "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation." *Proc. ACM SIGGRAPH '94*, pp 451-445. Orlando, Florida, July 24-29, 1994.

[Lacroute95] P. Lacroute. "Real-time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization." *Proceedings of the*

1995 Parallel Rendering Symposium, pp. 15-22. ACM Press, October, 1995.

[Laur91] D. Laur, P. Hanrahan. "Hierarchical splatting: A progressive refinement algorithm for volume rendering." *ACM Computer Graphics*, 25(**4**):285-288. 1991.

[Law96a] A. Law, R. Yagel. "The Active-Ray Approach to Rendering on Distributed Memory Multiprocessors." *Proc. IEEE Symposium on Parallel an Distributed Architectures*, 1996, pp. 441-421.

[Law96b] A. Law, R. Yagel. "Multi-Frame Thrashless Ray Casting with Advancing Ray-Front." *Proc Graphics Interface '96, Canadian Information Processing Society*, pp. 70-77.

[Law96c] A. Law, R. Yagel. "An Optimal Ray Traversal Scheme for Visualizing Colossal Medical Volumes." *Proceedings of Visualization in Biomedical Computing*, VBC '96, pp.33-43. Hamburg, Germany, September, 1996.

[Levoy88] M. Levoy. "Display of surfaces from volume data." *IEEE Computer Graphics and Applications*, May 1988:29-37.

[Levoy90] M. Levoy. "Efficient ray tracing of volume data." *ACM Transactions on Graphics*, 9(**3**), July 1990.

[Lorensen87] W.E. Lorensen, H.E. Cline. "Marching cubes: A high resolution 3d surface construction algorithm." *ACM Computer Graphics*, 21(**4**):163-169. 1988.

[Ma93] K. Ma, J. Painter, C. Hansen, and M. Krogh. "A Data Distributed Parallel Algorithm for Ray-Traced Volume Rendering." *Proceedings of the 1993 Parallel Rendering Symposium*, pp. 15-22. ACM Press, October 1993.

[Max86] N. Max. "Light Diffusion through Clouds and Haze." *Computer Vision, Graphics, and Image Processing*, **33**:280-292, 1986.

[Max90] N. Max, P. Hanrahan, R. Crawfis. "Area and volume coherence for efficient visualization of 3D Scalar functions." *ACM Computer Graphics*, 24(**5**):27-33. 1990.

[Mackerras94] P. Mackerras, B. Corrie. "Exploiting Data Coherence to Improve Parallel Volume Rendering. *IEEE Parallel and Distributed Technology*, 2(**2**), Summer 1994, pp. 8-16.

[Nemoto86] K. Nemoto, T. Omachie. "An adaptive subdivision by sliding boundary surfaces for fast ray tracing." *Proc Graphics Interface '86, Canadian Information Processing Society*, pp. 43-48.

[Neumann93] U. Neumann. "Parallel Volume-Rendering Algorithm Performance on Mesh-Connected Multicomputers." *Proceedings of the 1993 Parallel Rendering Symposium*, pp. 97-104. ACM Press, October 1993.

[Nieh92] J. Nieh, M. Levoy. "Volume rendering on scalable shared-memory MIMD architectures." *ACM SIGGRAPH 1992 Workshop on Volume Visualization*, pp. 17-24.

[Palmer94a] M.E. Palmer. *Immersing the Scientist in Data: Interactive Visualization of Unstructured Scientific Data on Concurrent Architectures.* Master's Thesis, Dept. of Computer Science, California Institute of Technology, Pasadena, CA. Caltech-CS-TR-94-06. 1994.

[Palmer94b] M.E. Palmer, S. Taylor, "Rotation Invariant Partitioning for Concurrent Scientific Visualization." *Parallel Computational Fluid Dynamics '94*, Kyoto, Japan. Elsevier Science Publishers B.V., 1994.

[Palmer95] M.E. Palmer, S. Taylor, B. Totty. "Interactive Volume Rendering on Clusters of Shared-Memory Multiprocessors." *Parallel Computa-*

tional Fluid Dynamics '95, Pasadena, CA. Elsevier Science Publishers B.V., 1995.

[Palmer97a] M.E. Palmer, B. Totty, S. Taylor. "Ray Casting Volume Rendering on Shared-Memory Architectures: Efficient Exploitation of the Memory Hierarchy." To appear in *IEEE Concurrency*, fall 1997.

[Palmer97b] M.E. Palmer, S. Taylor, B. Totty. "Exploiting Deep Parallel Memory Hierarchies for Ray Casting Volume Rendering." Submitted to *Proceeedings of the 1997 Parallel Rendering Symposium*. ACM Press, November, 1997.

[Porter84] T. Porter, T. Duff. "Compositing Digital Images." *ACM Computer Graphics* 18(**3**):253-259, July, 1984.

[Priol89] T. Priol, K. Bouatouch. "Static load balancing for a parallel ray tracing on a MIMD hypercube." *The Visual Computer* (1989), 5:109-119, Springer-Verlag 1989.

[Sabella88] P. Sabella. "A rendering algorithm for visualizing 3d scalar fields." *ACM Computer Graphics*, 22(**4**):51-58. 1988.

[Shirley90] Shirley P., and Tuchman, A. "A Polygonal Approximation to Direct Scalar Volume Rendering." *ACM Computer Graphics*, 24(**5**):63-70. 1990.

[Singh94] J.P. Singh, A. Gupta, M. Levoy, "Parallel Visualization Algorithms: Performance and Architectural Implications." *Computer* 27(**7**), July 1994, pp. 44-55.

[Sutherland74] I.E. Sutherland, R.F. Sproull, R.A. Schumacker. "A characterization of ten hidden-surface algorithms." *Comput. Surv.* 6(**1**), 1-55, March 1974.

[Schröder91] P. Schröder, J. Salem. "Fast Rotation of Volume Data on Data Parallel Architectures." *Visualization '91*, San Diego, October, 1991.

[Schröder92] P. Schröder, G. Stoll. "Data Parallel Volume Rendering as Line Drawing." *Proceedings of the Workshop on Volume Visualization*, pp. 25-31. ACM Press. October, 1992.

[Skibo95] Skibo, T. Results of HIPPI benchmark test performed by Thomas Skiboof Silicon Graphics, skibo@sgi.com, August, 1995. http://reality.sgi.com/employees/skibo/hippiTcpPerf.html

[Upson88] C. Upson, M. Keeler. "V-BUFFER: Visible volume rendering." *ACM Computer Graphics*, 22(**4**):59-65. 1988.

[Watts97] SCPlib was created by Jerrell Watts of the Scalable Concurrent Programming Laboratory, http://www.scp.syr.edu.

[Westermann95] R. Westermann. "Parallel Volume Rendering." *Proceedings of the 1995 International Parallel Processing Symposium (IPPS '95)*, pp. 693-699.

[Westover90] L. Westover. "Footprint evaluation for volume rendering." *ACM Computer Graphics*, 24(**4**):367-376. 1990.

[Whitman93] S. Whitman. "A Task Adaptive Parallel Graphics Renderer." *Proceedings of the 1993 Parallel Rendering Symposium*, San Jose, California, 1993, pp. 27-34, color plate page 107. ACM Press, October, 1993.

[Wilhelms91] J. Wilhelms, A. Van Gelder. "A Coherent Projection Approach for Direct Volume Rendering." *ACM Computer Graphics*, 25(**4**):275-284. 1991.

[Wittenbrink93] C. Wittenbrink, A. Somani. "Permutation warping for volume rendering." *Proceedings of the 1993 Parallel Rendering Symposium*, San

Jose, California, 1993, pp. 57-60, color plate page 110. ACM Press, October, 1993.

[Wittenbrink94] C. Wittenbrink. "A Scalable MIMD Volume Rendering Algorithm." *Proceedings of the 1994 International Parallel Processing Symposium (IPPS '94)*, pp. 916-919.

[Woodward95] The vorticity dataset was provided courtesy of Paul Woodward of the Laboratory for Computational Science and Engineering (LCSE), University of Minnesota.