A HYPERGRAPH-PARTITIONING BASED REMAPPING MODEL FOR IMAGE-SPACE PARALLEL VOLUME RENDERING

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

by
Berkant Barla Cambazoglu
February, 2000

Thesis
T
385
.C36
2000

A HYPERGRAPH-PARTITIONING BASED REMAPPING MODEL FOR IMAGE-SPACE PARALLEL VOLUME RENDERING

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

by

Berkant Barla Cambazoğlu

February, 2000

T
385
.C36
2000

B051146

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Cevdet Aykanat (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Uğur Güdükbay

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Atilla Gürsoy

Approved for the Institute of Engineering and Science:

Prof. Mehmet Baray
Director of Institute of Engineering and Science

ABSTRACT

A HYPERGRAPH-PARTITIONING BASED REMAPPING MODEL FOR IMAGE-SPACE PARALLEL VOLUME RENDERING

Berkant Barla Cambazoğlu
M.S. in Computer Engineering
Supervisor: Assoc. Prof. Cevdet Aykanat
February, 2000

Ray-casting is a popular direct volume rendering technique, used to explore the content of 3D data. Although this technique is capable of producing high quality visualizations, its slowness prevents the interactive use. The major method to overcome this speed limitation is parallelization. In this work, we investigate the image-space parallelization of ray-casting for distributed memory architectures. The most important issues in image-space parallelization are load balancing and minimization of the data redistribution overhead introduced at successive visualization instances. Load balancing in volume rendering requires the estimation of screen work load correctly. For this purpose, we tested three different load assignment schemes. Since the data used in this work is made up of unstructured tetrahedral grids, clusters of data were used instead of individual cells, for efficiency purposes. Two different cluster-processor distribution schemes are employed to see the effects of initial data distribution. The major contribution of the thesis comes at the hypergraph partitioning model proposed as a solution to the remapping problem. For this purpose, existing hypergraph partitioning tool PaToH is modified and used as a one-phase remapping tool. The model is tested on a Parsytec CC system and satisfactory results are obtained. Compared to the two-phase jagged partitioning model, our work incurs less preprocessing overhead. At comparable load imbalance values, our hypergraph partitioning model requires 25% less total volume of communication than jagged partitioning on the average.

Keywords: image-space parallelization, ray-casting, unstructured grids, work load assignment, hypergraph partitioning, load balancing, remapping.

ÖZET

GÖRÜNTÜ-UZAYI PARALEL HACİM GÖRÜNTÜLEME İÇİN HİPERÇİZGE BÖLÜMLEMEYE DAYALI YENİDEN EŞLEME MODELİ

Berkant Barla Cambazoğlu
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Doç Dr. Cevdet Aykanat
Şubat, 2000

Işın izleme, üç boyutlu verilerin incelenmesinde kullanılan, popüler bir doğrudan hacim görüntüleme tekniğidir. Bu teknik yüksek kalitede görüntüler üretebilecek kapasitede olmasına rağmen yavaşlığı birebir etkileşimli kullanımını engellemektedir. Bu hız sınırlamasını aşmanın en önemli yolu paralelleştirmedir. Bu çalışmada, ışın izlemenin dağıtık bellekli mimarilerdeki görüntü-uzayı paralelleştirmesi araştırılmıştır. Görüntü-uzayı paralelleştirmedeki en önemli konular yük dengeleme ve takip eden görüntüleme örneklerinde ortaya çıkan veri yeniden dağıtım yükünün en aza indirilmesidir. Hacim görüntülemedeki yük dengeleme, ekran iş yükünün doğru olarak tahminini gerektirmektedir. Bu amaçla, üç değişik yük tahsis etme planı denenmiştir. Bu çalışmada kullanılan veriler düzensiz tetrahedral ızgaralardan oluştuğu için, verimlilik amacıyla bireysel veri hücreleri yerine veri grupları kullanılmıştır. İlk veri dağılımının etkilerini görmek için iki farklı veri grubu-işlemci dağılım planı kullanılmıştır. Çalışmanın en önemli katkısı yeniden eşleme problemine bir çözüm olarak önerilen hiperçizge bölümleme modelidir. Bu amaçla, var olan hiperçizge parçalama aracı PaToH değiştirilerek tek safhalı yeniden eşleme aracı olarak kullanılmıştır. Model Parsytec CC sisteminde denenmiş ve tatmin edici sonuçlar elde edilmiştir. Önerilen yöntem iki safhalı kesikli bölümleme modeline göre, daha az ön hazırlık yükü yaratmaktadır. Kıyaslanabilir yük dengesizliklerinde, önerilen hiperçizge modeli kesikli bölümleme modelinden ortalama %25 daha az toplam iletişim hacmi gerektirmektedir.

Anahtar Kelimeler: görüntü-uzayı paralelleştirme, ışın izleme, düzensiz ızgaralar, iş yükü tahsisi, hiperçizge bölümleme, yük dengeleme, yeniden eşleme.

ACKNOWLEDGMENTS

I thank to my advisor Assoc. Prof. Cevdet Aykanat for his ideas, suggestions and help on this thesis. I also thank Asst. Prof. Uğur Güdükbay and Asst. Prof. Atilla Gürsoy for reading my thesis. I express my endless thanks to my family and my friend Ayça Özger for their support and patience.

Contents

1 Introduction **1**

1.1 Terminology and Classification 1

1.2 Previous Work 5

1.3 Proposed Work 6

2 Ray-Casting **9**

2.1 Basic Ray-Casting Algorithm 9

2.2 Data Structures for Unstructured Grids 11

2.3 Koyamada's Algorithm 13

2.3.1 Intersection Test 14

2.3.2 Resampling 16

2.4 Optimizations and Performance 18

3 Image-Space Parallelization **20**

3.1 OS versus IS Parallelization 20

3.2 Clusterization 22

3.2.1 Graph Partitioning 23

3.2.2 Weighting Scheme 24

3.2.3 Additional Data Structures 25

3.3 Load Balancing 25

3.3.1 Screen Subdivision 26

3.3.2 Work Load Calculation 28

3.4 Remapping 30

4 Hypergraph Partitioning 32

4.1 Introduction 32

4.2 Partitioning Methods 34

4.2.1 Iterative Improvement Methods 34

4.2.2 Multilevel Methods 36

5 A Remapping Model 40

5.1 Remapping by Hypergraph Partitioning 40

5.1.1 Two-Phase Hypergraph Partitioning Model 42

5.1.2 One-Phase Hypergraph Partitioning Model 43

5.2 Data Distribution 45

5.3 Hypergraph versus Jagged Partitioning 47

6 Implementation Details 52

6.1 View Independent Preprocessing 52

6.2 View Dependent Preprocessing 54

6.2.1 Work Load Calculation 54

6.2.2 Local Hypergraph Creation 55

6.2.3 Global Hypergraph Creation 55

6.2.4 Hypergraph Partitioning 55

6.3 Cluster Migration 55

6.4 Local Rendering 56

7 Experimental Results 57

7.1 Implementation Platform and Data Sets Used 57

7.2 View Independent Preprocessing 59

7.3 View Dependent Preprocessing 61

7.4 Performance 63

7.5 Comparison with Jagged Partitioning 64

8 Conclusion 69

8.1 Work Done 69

8.2 Future Work 70

A Calculation of Granularity Formula 77

List of Figures

1.1 Grid types in 2D. 2

1.2 Rendering pipelines for distributed memory architectures. 5

2.1 Ray casting. 10

2.2 Ray-casting for unstructured grids with mid-point sampling. . . 12

2.3 Data structures for tetrahedral unstructured data. 12

2.4 A case where face sorting fails. 13

2.5 Ray buffers contain the ray-segments generated for each pixel. . 14

2.6 Intersection test. 15

2.7 Example transfer functions. 17

3.1 Data-processor assignments in OS and IS parallelization. 21

3.2 Cell clusterization using graph partitioning. 23

3.3 Additional data structures used after clusterization. 26

3.4 Screen subdivision techniques. 27

3.5 Effect of projection area on grid granularity. 28

3.6 Work load assignment schemes. 31

4.1 Multilevel hypergraph bisection. 37

5.1 Representing the interaction between OS and IS by an hypergraph. 41

5.2 The partitioning cost calculated by two-phase method may be incorrect. 43

5.3 Special vertices are introduced into the hypergraph in one-phase model. 44

5.4 Cluster distribution schemes. 46

5.5 Rendered image of CC data set. 49

5.6 Example region-processor assignment in jagged partitioning. . . 50

5.7 Example region-processor assignment in hypergraph partitioning. 51

7.1 Load imbalances in HP and JP. 66

7.2 Preprocessing overhead incurred in HP and JP. 67

7.3 A comparison of communication volumes in HP and JP. 68

A.1 Imposing g by g screen cells onto an n by n area. 77

List of Tables

1.1 Grid classification. 2

1.2 Parallelization of DVR algortihms. 6

3.1 Possible weighting schemes for the clusterization graph. 24

7.1 Some features of the data sets used. 58

7.2 Abbreviations used in tables. 59

7.3 Results obtained by assigning different cluster counts per processor. 60

7.4 Effects of all possible weighting schemes used in the clusterization graph. 60

7.5 Effects of different work load calculation schemes. 61

7.6 Imbalance values and communication amounts observed. 62

7.7 Dissection of view dependent preprocessing time. 63

7.8 Speedup and efficiency values for different data sets and processor numbers. 64

A.1 Adaptive granularity calculation. 78

List of Symbols and Abbreviations

DVR	:	Direct Volume Rendering.
ISP	:	Image-Space Parallelization.
OSP	:	Object-Space Parallelization.
HP	:	Hypergraph Partitioning
JP	:	Jagged Partitioning
ff	:	front-facing.
bf	:	back-facing.
eff	:	external front-facing.
ebf	:	external back-facing.
KL	:	Kernighan-Lin hypergraph partitioning heuristic.
FM	:	Fiduccia-Mattheyses hypergraph partitioning heuristic.
α, β, γ	:	Scaling coefficients in intersection test.
R_i	:	Red color component of ith resampling point.
G_i	:	Green color component of ith resampling point.
B_i	:	Blue color component of ith resampling point.
O_i	:	Opacity component of ith resampling point.
λ_x	:	Transfer function for component x, where $x \in \{r, g, b, o\}$.
$\mathcal{F}_{\mathrm{ff}}$	:	Set of *ff* faces.
$\mathcal{F}_{\mathrm{eff}}$	:	Set of *eff* faces.
f	:	A tetrahedral face.
$Area(f)$	:	Function which returns the area of face f.
N	:	Number of nodes in the data.
T_{tr}	:	Total time spent on node transformation.
t_{tr}	:	Average time spent to transform a single node.
T_{sc}	:	Total time spent on scan conversion of *eff* faces.
t_{sc}	:	Average scan conversion cost for a pixel.
I_{xy}	:	Number of intersections made by a ray shot from pixel (x, y).
T_{it}	:	Total time spent on intersection test.
t_{it}	:	Average time spent on intersection test for an intersection.
T_{rs}	:	Total time spent on resampling.
t_{rs}	:	Average resampling time of a point.
T	:	Total time spent on rendering.

$\mathcal{G}$	: Graph used in clusterization.
$\mathcal{V}$	: Set of vertices in $\mathcal{G}$.
$\mathcal{E}$	: Set of edges in $\mathcal{G}$.
v_i	: ith vertex in set $\mathcal{V}$.
e_{ij}	: Edge connecting vertices v_i and v_j.
w_i	: Weight of the vertex v_i in set $\mathcal{V}$.
w_{ij}	: Weight of the edge between vertices v_i and v_j.
$\mathcal{C}$	: Set of data clusters.
$\mathcal{C}_i$	: ith cluster in set $\mathcal{C}$.
$\mathcal{S}$	: Set of screen parts.
$\mathcal{S}_i$	: ith screen part in set $\mathcal{S}$.
r_i	: ith screen cell.
K	: Number of processors used.
P	: Set of processors used.
P'	: Set of processors to which a cluster will migrate.
P_i	: ith processor in set P.
$\mathcal{H}$	: Remapping hypergraph in two-phase model.
$\mathcal{H}'$	: Remapping hypergraph in one-phase model.
$\mathcal{V}$	: Set of vertices in $\mathcal{H}$.
$\mathcal{V}'$	: Set of vertices in $\mathcal{H}'$.
$\mathcal{N}$	: Set of nets in $\mathcal{H}$.
v_i	: ith vertex in a hypergraph.
n_i	: ith net in a hypergraph.
p_i	: ith special vertex, representing P_i in $\mathcal{H}'$.
Π	: Set of screen parts returned by hypergraph partitioning in two-phase model.
Π'	: Set of screen parts returned by hypergraph partitioning in one-phase model.
ϵ	: Overall load imbalance value for a partition.
$\mathcal{N}_E(\mathcal{V}')$	: Set of external nets which has a pin on a vertex in set $\mathcal{V}'$.
$\gamma(v)$	: Gain obtained by moving vertex v.
$\gamma_r(v)$	: rth level Gain obtained by moving vertex v.
$B_{\mathcal{P}_i}(n)$	: Binding number of net n.
$\mathcal{M}(\mathcal{S}_i)$	: Mapping funcion which maps $\mathcal{S}_i$ to P_j.
d_i	: Degree of ith vertex in a hypergraph.
s_i	: Size of ith net in a hypergraph.

Λ_i : Connectivity set of a net n_i.

λ_i : Connectivity of a net n_i.

$\chi(\Pi)$: Cost of the partition Π.

$Pins(n_i)$: Operator which returns the set of vertices on which n_i has a pin.

$Nets(v_i)$: Operator which returns the set of nets which has a pin attached on v_i.

$\mathcal{B}$: Bipartite graph used in the second phase of two-phase model.

$\mathcal{X}$: Set of processors in $\mathcal{B}$.

$\mathcal{Y}$: Set of screen parts in $\mathcal{B}$.

$\mathcal{Z}$: Set of edges in $\mathcal{B}$.

x_i : vertex representing ith processor, in set $\mathcal{X}$.

y_i : vertex representing ith screen part, in set $\mathcal{Y}$.

$Load(\mathcal{C}_i)$: Function that returns the rendering load of the cluster $\mathcal{C}_i$.

$Load(r_i)$: Function that returns the rendering load of the screen cell r_i.

$Cost(\mathcal{C}_i)$: Function that returns the migration cost of the cluster $\mathcal{C}_i$.

Chapter 1

Introduction

The huge improvements in computing capabilities of hardware, and developments on the visualization software allowed researchers, students and people from many different work areas to study the interiors of 3-dimensional data on their desktops. Today, *volume visualization* stands as a science discipline, which is commonly used as a tool to aid the research by letting the scientists to get a visual grasp of the problem under investigation. The main method for scientific volume visualization [36] is *volume rendering*. It finds application in various areas such as hydrodynamics, molecular biology, sysmology and meteorology.

1.1 Terminology and Classification

Volume rendering can be simply defined as the process of mapping a set of scalar, tensor or vector values defined in 3D to a 2D image screen. In order to represent these volumetric data sets, different kinds of grids are used. Grids, according to their structural properties, can be classified as in Table 1.1.

Irregular grids are the most interesting type of grids with the ability to represent disparate field data effectively. In rectilinear grids, non-uniform, axis-aligned rectangular prisms are used as volumetric primitives (*voxels*). Curvilinear grids have the same topological structure with rectilinear grids, but they are

Table 1.1. Grid classification.

<table>
<tr><th colspan="3">Grids</th><th>Primitives used</th></tr>
<tr><td colspan="3">Cartesian</td><td>axis-aligned,
uniform cubes</td></tr>
<tr><td rowspan="4">Non-Cartesian</td><td colspan="2">Regular</td><td>axis-aligned,
uniform rectangular prisms</td></tr>
<tr><td rowspan="3">Irregular</td><td>Rectilinear</td><td>axis-aligned,
non-uniform rectangular prisms</td></tr>
<tr><td>Curvilinear</td><td>non-axis-aligned,
non-uniform hexahedra</td></tr>
<tr><td>Unstructured</td><td>no implicit connectivity,
polyhedra</td></tr>
</table>

warped from computational space to physical space. With the increase in the number of tools and methods for generating high quality adaptive meshes, unstructured grids are also gaining more popularity. Unstructured grids contain polyhedra with no implicit connectivity information. Volumetric primitives (*cells*) such as tetrahedra, hexahedra, and prisms can be used in these grids. However, because any volume can be decomposed into tetrahedra, and they are easy to work with, in most cases tetrahedral cells are used to form unstructured grids. Figure 1.1 shows 2D equivalents of these grid types.

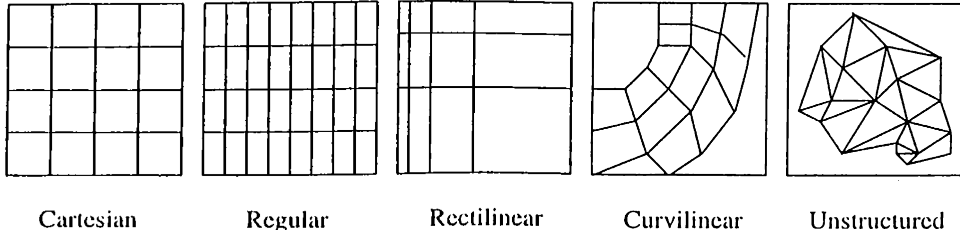


Figure 1.1. Grid types in 2D.

Two basic categories can be considered for volume rendering algorithms [7]: *Surface-based* algorithms, which compute different levels of surfaces within a given volume, and *direct volume rendering* (DVR) algorithms, which display the integral densities along imaginary rays passed between the viewers eyes and the volume data. Surface based methods are sometimes referred as indirect methods, since they try to extract an intermediate representation for the data set. Their main idea is to construct some level surfaces using the sample points with close density values, and represent them with a set of contiguous polygons, which will later be rendered in the standard graphics pipeline.

Despite the fact that surface-based algorithms are fairly well suited for areas such as medical imaging, which requires specific tissue boundaries to be displayed, there are many other areas in which they cannot be utilized due to the problems associated with the computation of surface levels. In most data sets, using artificial surfaces may result in highly non-linear discontinuities in the data and introduces artifacts in the rendered image. In other words, a surface may not always represent the actual data structure correctly. To overcome this problem, direct volume rendering techniques, which treat the volume data as a whole, are employed. DVR is a powerful tool for visualizing data sets with complex structures defined on 3D grids.

The main DVR methods are *ray-casting* and *data projection*, which are sometimes referred as *image-space* and *object-space* methods, respectively. Cell projection [12] and splatting [45] are the two examples of object-space methods. In cell projection based DVR algorithms, the projection primitives (triangle, tetrahedra, or cube) must be sorted with respect to the viewing point, due to the usage of the composition formula based on color and opacity accumulation at sampling points. However, because of the ambiguities in the visibility ordering [42] of projection primitives, the visualization process may yield a poor final rendering. Similar problems, which affect the image quality, arise in other object-space methods, too. For example, in splatting algorithms, effect extends of the resampling points should be approximated correctly.

Ray-casting [18, 41], without any hesitation, can be said to be a very good candidate to produce high-quality, realistic images. This method works by shooting rays from the image plane into the volume data, and combining the color and opacity values calculated at resampling points throughout the data. Because each ray's contribution to a pixel color is independent of all other rays, ray-casting algorithms cannot utilize the object-space coherency well. As a result, their elegancy comes at a cost.

The computational cost of DVR algorithms is affected by the huge amount of information to be processed in the data sets, and it prevents their widespread use. Although many optimization techniques are known, speeds of DVR algorithms are still far from interactive response times. The CPU speeds at which the current processors operate is not the only limitation before direct

volume rendering. Limited amount of physical memory in workstations can also be a bottleneck. Since some portions of the data may not fit into main memory, it may be necessary to access the data from virtual memory resulting in a much slower data access rate.

Parallelization of the existing DVR algorithms [46] is the main technique to overcome the speed limitations mentioned above. Considerable speedups can be gained through parallelization without trading the image quality for rendering speed. *Shared memory* or *distributed memory* architectures can be used for parallelization. In shared memory parallel machines, each processor has access to a global memory via some interconnect or bus. The global memory can be a single module, or can be divided equally among processors. Processors communicate by using the bus, through read and write operations performed over memory locations in the global memory. Although it presents ease of programming and flexibility, shared memory architectures does not scale well, due to the bottlenecks occurred during memory access.

In distributed memory architectures, each processor is given its own memory, which is not directly accessible by other processors. If a processor needs data contained in the memory of a remote processor, it sends a message asking for the data and retrieves it from that processor through an interconnection network. As a result, the data access is not always uniform, and issues such as data distribution, communication bandwidth and network topology gain importance.

Parallelization process can be carried out in 3D object domain or 2D screen domain, resulting in *object-space parallelization* (OSP) and *image space parallelization* (ISP), respectively. In OSP, each processor is assigned a sub-volume of the data, and produces the partial color information for the final image by rendering its volumetric primitives. Later, these partial results are merged at appropriate processors by a pixel merging step. Communication is needed to send the sub-results to their destination processors, where they will be combined to determine the final pixel colors. Hence, OSP is known as a pixel-flow method.

ISP, on the other hand, is a data-flow method. Instead of sub-volumes,

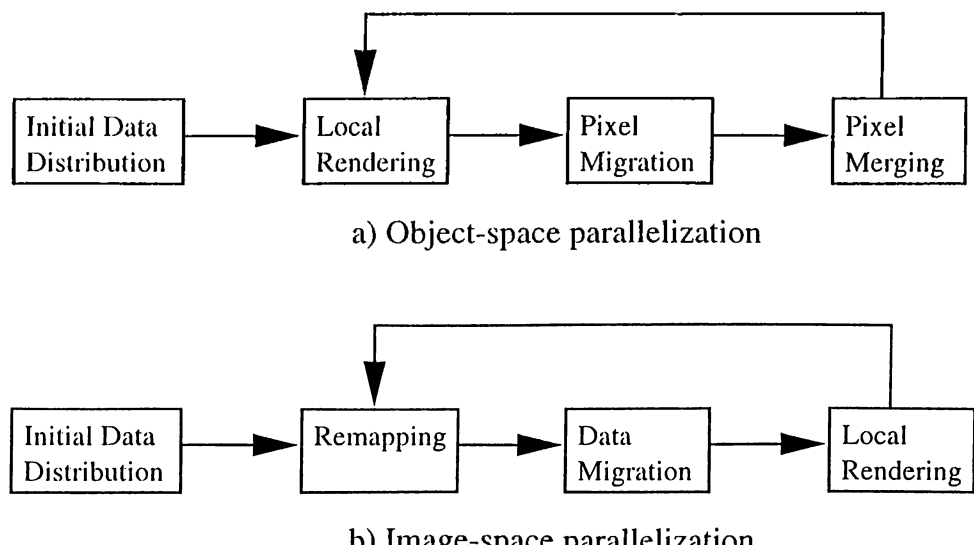


Figure 1.2. Rendering pipelines for distributed memory architectures.

each processor is assigned a sub-region over the screen and is responsible for rendering that particular region. However, since a processor may not possess all the primitives necessary for rendering its region, required sub-volumes are retrieved from processors owning them, before the rendering process starts. Figure 1.2 displays the general rendering pipelines for OSP and ISP on distributed memory architectures.

1.2 Previous Work

With the increase in their availability and decrease in their prices, massively parallel computers are becoming more popular. In the last decade, many attempts were done to parallelize the existing DVR algorithms. However, most of these work dealt with the structured kind of data. Work on unstructured data took less attention. Table 1.2 displays the references to latest work on parallelization of DVR methods. The classification in that table is done according to the type of the architecture and the parallelization method used.

Although lots of research is carried out on volume visualization, it is still very difficult to establish the standards to compare the quality of the works done on volume rendering. Unfortunately, this becomes more apparent in

Table 1.2. Parallelization of DVR algortihms.

	OS Parallelization	IS Parallelization
Shared Memory	[44]	[35]
Distributed Memory	[8] [32] [33] [37]	[4] [29]
Special Hardware	[47]	[48]

parallelization of DVR methods. There are many criteria such as data set size, execution speed, image quality, load balance, speedup, scalability that can be used to compare a work with the previous, similar works. This makes their comparison harder.

Considering the classification made in Section 1.1, our work can said to be the parallelization of ray-casting technique for distributed memory architectures. In our work, image-space decomposition was chosen for parallelization and the data used is of type tetrahedral unstructured grids.

There is little research done in this area. Hence, we compared our work with a similar algorithm, which uses jagged partitioning to divide the screen. Jagged partitioning is one of the best screen space subdivision algorithms. However, since it is mainly concentrated on the load balance between partitions, it lacks the power to minimize the communication between partitions. At same load balance, and preprocessing overheads we observed that the communication cost during the remapping phase, incurred by jagged partitioning can be up to 30% higher than the cost observed in our work.

1.3 Proposed Work

The ray-casting code we used is a slightly modified implementation of Koyamada's ray-casting algorithm [26]. This algorithm is a rather efficient algorithm, which makes use of both object-space coherency and image-space coherency. Our main modification to this algorithm is the use of a higher level of volumetric data abstraction.

In some cases, instead of working on individual tetrahedral cells we deal with connected clusters of tetrahedral cells. Although introducing clusters may result in data replication during parallelization, this effect may be negligible if the number of clusters used is kept high enough. Clusterization [9] simplifies the housekeeping work, and decreases the number iterations in some loops. Its main use comes at the computation of the screen work load, i.e., the distribution of the cell rendering costs over their projection areas on the screen. In addition, this clusterization process is necessary to obtain the condensed hypergraph which will be used during the remapping step.

The fundamental problem in image-space based DVR methods is that if a visualization parameter such as the viewpoint location or the viewing direction changes, the image on the screen should be wholly recomputed. For image-space parallelization, this creates a problem known as the *remapping* problem. During successive visualizations, the rendering costs of volumetric primitives distributed over screen pixels can largely change, resulting in severe load imbalances among the processors. This necessitates the migration of some volume data to other processors in order to balance the load distribution. The aim of remapping step is both to obtain a good load balance by shifting data from heavily loaded processors to lightly loaded processors, and to minimize the communication overhead incurred by this data migration.

Our main contribution is at the proposed remapping model. In this work, remapping problem is formulated as a hypergraph partitioning problem, so that the interaction between the object and image domains is represented by an hypergraph. A net in this hypergraph stands for a cell cluster in the volume data, and its weight shows the migration cost of the cluster. Cells of the hypergraph represent the pixels over the screen. Each pixel's rendering cost is assigned to its representative cell.

The tool used for hypergraph partitioning is PaToH. In order to obtain a *one-phase* remapping model, some modifications were done on the hypergraph model and PaToH, giving them the ability to treat some cells differently than the others. Some special cells are placed in the hypergraph to represent the processors used during execution. A special vertex is connected to a net by a pin, if net's cluster resided in the local memory of the processor. Details of

this one-phase model can be found in Chapter 5.

The implementation of our parallel ray-casting algorithm is composed of four consecutive phases: View independent preprocessing, view dependent preprocessing, cluster migration and rendering. View independent preprocessing is performed just once at the beginning of each run. It includes some steps such as cell clusterization, initial data distribution, and disconnected cluster elimination. In view dependent preprocessing, some transformations are applied on the data, and also clusters are mapped to new processors through hypergraph partitioning. In cluster migration step, communication is performed to send clusters to their new locations. An important fact is that the overhead incurred by these last two steps should be minimal, since they are executed at the beginning of every visualization instance. The final step is the rendering of the clusters locally by the processors. Because processors have all the clusters needed over their assigned screen regions, no global pixel merging is necessary.

The organization of the thesis is as follows: In Chapter 2, ray-casting and our implementation of Koyamada's algorithm were explained. Chapter 3 discusses image-space parallelization issues. Chapter 4 is an introduction to hypergraph partitioning. Our remapping model and parallel ray-casting implementation were presented in Chapter 5 and Chapter 6, respectively. Chapter 7 gives some experimental results, and Chapter 8 concludes the thesis.

Chapter 2

Ray-Casting

Ray-casting is an image-space rendering method, in which some rays are shot from the observers eye (*viewpoint*) into the volume data through the pixels over the image screen, and final pixel colors are calculated using the color contributions of the sample points over the ray. Sometimes, in the literature, the term ray-casting is used to refer to ray-tracing, although they are not the same. In ray-casting, only the shadow rays are considered, ignoring reflection rays and transmission rays which are important in ray-tracing. Thus, ray-casting can be said to be a simple form of traditional ray-tracing.

The next section describes the basic ray-casting algorithm. In the section following it, Koyamada's ray-casting algorithm which works on unstructured grids is explained. This algorithm forms the basis for our ray-casting code. Finally, some optimizations done in ray-casting and the performance of our rendering algorithm is discussed.

2.1 Basic Ray-Casting Algorithm

Before the ray-casting algorithm has started, it is assumed that the scalar values on grid vertices and the viewing orientation were already determined by the user. The viewing orientation is specified by the following three parameters: *view-reference point, view-direction vector, and view-up vector*. Together with

these three parameters, image screen resolution parameter is used to transform the grid vertices, which are originally in *world space coordinate* (WSC) system, into *normalized projection coordinate* (NPC) system, which will be used in ray-casting. Also, some transfer functions, which will map the scalar values at resampling points into an RGB color tuple and an opacity value, were assigned previously.

In ray-casting we perform an image-order traversal over the screen pixels, and try to assign a final color value to each pixel. To find a pixel's color, first, a ray is shot from the viewpoint into the volume data passing through that pixel (Figure 2.1). This ray is followed within the volume, and some sample values are calculated at the resampling points along the ray at some regular intervals. If the resampling point is not exactly on a grid vertex, its value is approximated by interpolating the scalar values at some close grid vertices. Different sampling methods and interpolation techniques are discussed in the following sections, in more detail.

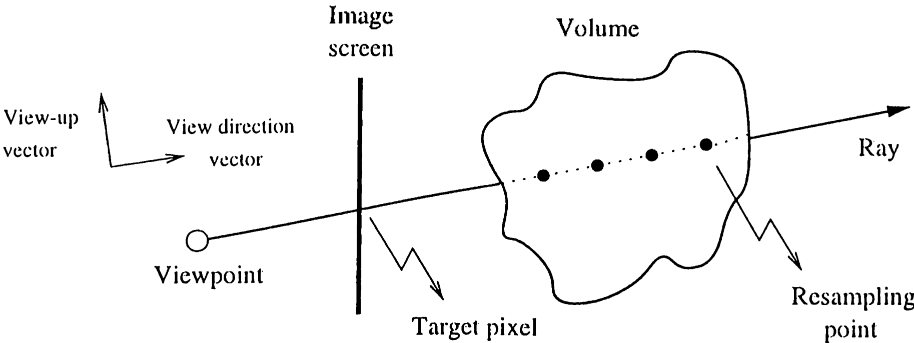


Figure 2.1. Ray casting.

At each resampling point, the transfer functions are applied to the sample values found, and color and opacity contributions of resampling points are calculated. Then, using a weighting formula, these color and opacity values are accumulated into the final color value of the pixel, from which the ray was shot. The weighting formula is such that, the points closer to the pixel contribute more than the points far from the screen.

This sampling step repeats until the ray reaches the end of the volume or the accumulated opacity reaches unity. At the end, the accumulated color tuple is multiplied by the partial opacity and the final color is stored for that

pixel. The algorithm continues by moving onto the next pixel, and performing all the steps above for this newly selected pixel.

2.2 Data Structures for Unstructured Grids

Grids representing volumetric data can be constructed from different primitives such as rectangular prisms, hexahedron, tetrahedron, polyhedron or a mixture of these. As we did in this work, mostly tetrahedral primitives are used in unstructured grids. We refer to tetrahedral volume elements as *cells* here. All types of polyhedra can be converted into a set of tetrahedral cells through a process called tetrahedralization. A points value inside a tetrahedral cell can be interpolated directly, and the data distribution is linear in any direction inside the cell. Also, for this type of cells explicit connectivity structure is easier to be established.

A cell is made up of four planar, triangular faces and four corner points, called its *vertices*. Each vertex of a cell is actually a sample point with WSC values and an associated scalar value. The cell faces are classified as either internal or external. A common face shared by two different cells is an internal face. If a face is not shared by a neighbor cell, it is an external face. We call a cell with no external faces as an internal cell. Otherwise, the cell has at least one external face, and it is called an external cell.

Cell faces can also be classified according to the angle between their normal vectors and the view direction vector. If those vectors are perpendicular to each other, then the face is parallel to the ray casted. In case the angle is less than 90°, we call the face a front-facing (*ff*) face. Otherwise, it is a back-facing (*bf*) face. An external *ff* face is named as an *eff* face. Similarly, an external *ff* face is named as an *ebf* face. Finally, we use the sets $\mathcal{F}_{ff}$ and $\mathcal{F}_{eff}$ to denote the sets of *ff* and *eff* faces, respectively (Figure 2.2).

Our tetrahedral cell data structure mainly contains two arrays: *Nodes* array, and *Cells* array. Size of the *Nodes* array is equal to the number of sampling points in the data. Each item of this array represents a single sampling point

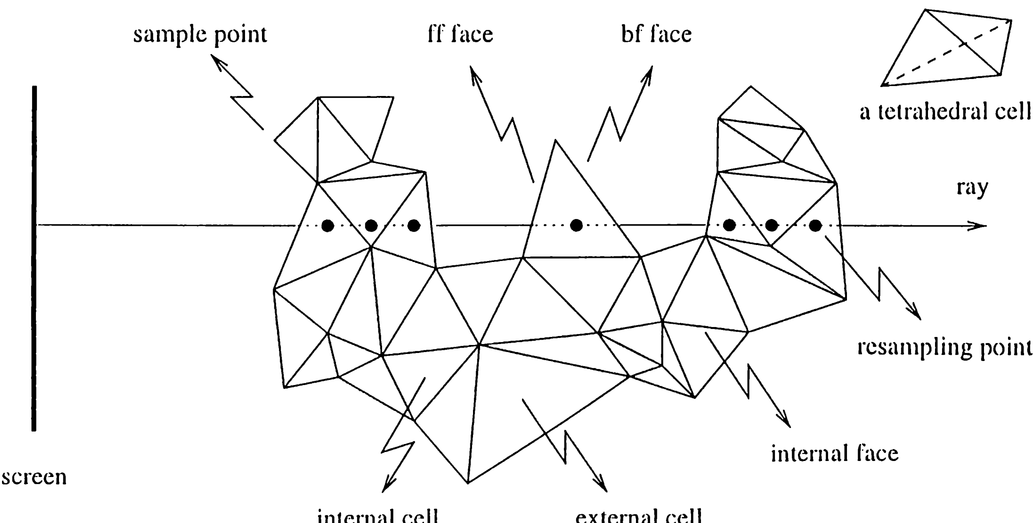


Figure 2.2. Ray-casting for unstructured grids with mid-point sampling.

and stores that points WSC, NPC values as well as the scalar value at that point. The scalar and coordinate values are stored as float numbers.

The second major array, namely the *Cells* array, is used to establish the connectivity between the cells. The number of items in this array equals to the tetrahedral cell count in the data. In an array item, for each face, the following information is stored: *Nodes* array indices of the four vertices forming the cell, *Cells* array indices of the four neighbor cells, and a number ranging from 0 to 3 to distinguish the shared faces of the neighbor cells. For non-shared faces of external cells a sentinel value of -1 is used. The data structures used can be seen in Figure 2.3.

```
struct Node {              struct Point {       struct Cell {
   struct Point WSC;          float x;             int vertices[4];
   struct Point NPC;          float y;             int neighborCells[4];
   float scalar;              float z;             int neighborFaces[4];
}                          }                    }
```

Figure 2.3. Data structures for tetrahedral unstructured data.

2.3 Koyamada's Algorithm

Koyamada's algorithm is a ray-casting algorithm that works on unstructured grids and is a rather efficient algorithm. It both tries to use the image-space coherency existing on the screen and the object-space coherency within the data. Image-space coherency is exploited during the scan conversion of *eff* faces, in order to determine the first ray-cell intersections. Object-space coherency is utilized by means of the connectivity information between the cells. Moreover, the results obtained from ray-face intersection is used for interpolation of scalar values making the interpolation operation very fast. Finally, due to the linear sampling method used, the resampling operations are very efficient.

Initial step taken in the algorithm is to scan convert the *eff* faces and find the first ray-volume intersections. Because the volume or the sub-volumes used during parallelization may be non-convex, more than one ray-segment may be generated for the same pixel. Due to the nature of the color composition formula, this ray-segments should be merged in a sorted order. In the original algorithm, *eff* faces are sorted with respect to the z coordinate of their centroids, and scan converted in that order.

However, this is an approximate ordering and may be wrong in some cases as shown in Figure 2.4. In that figure, although R_1 should be traversed before R_2, R_2 is processed and composited first; since the centroid of $\overline{KL}$, C_1, is sorted before the centroid of $\overline{MN}$, C_2. Our implementation overcomes this problem, which may lower the image quality, by means of a ray buffer data structure. This data structure keeps a linked list for each pixel, and the list elements

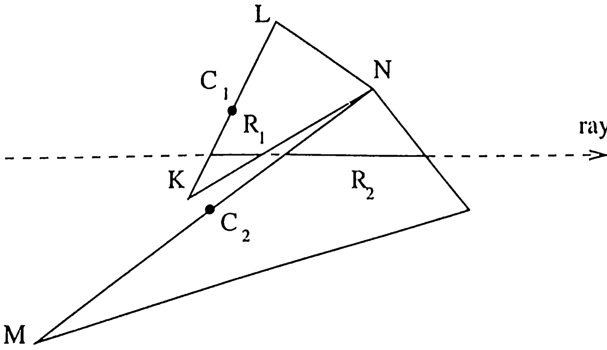


Figure 2.4. A case where face sorting fails.

contain composition information (i.e. accumulated color and opacity) about the ray-segments fired from that pixel (Figure 2.5). After all ray-segments are calculated and inserted into their respective lists in sorted order, they are merged by a pixel merging step in correct order.

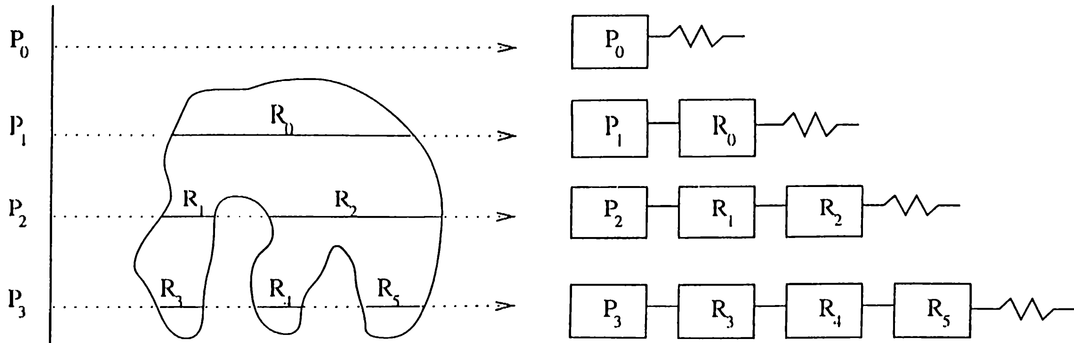


Figure 2.5. Ray buffers contain the ray-segments generated for each pixel.

The ray segments are traversed until they exit at some point from an *cbf* face of the volume. When a ray enters a cell from a face it exits the cell through one of that cells remaining three faces. This particular exit face, and the exit point coordinates are found by the intersection test. An exit point from a cell constitutes the entry point to another cell, therefore just one intersection is performed per cell.

The following subsections discuss the two important steps of the algorithm, that is, the intersection test, and the resampling steps, which are performed in an interleaved manner.

2.3.1 Intersection Test

In Figure 2.6, we consider the intersection test of a ray with the ABC face of a cell. Since x and y coordinates for the exit point Q are already known ($Q_x = R_x$ and $Q_y = R_y$), the problem is to find the Q_z coordinate value. This is done by expressing $\overrightarrow{AQ}$ as the summation of the two vectors whose directions are same with $\overrightarrow{AB}$ and $\overrightarrow{AC}$:

$$\overrightarrow{AQ} = \alpha\overrightarrow{AB} + \beta\overrightarrow{AC} \tag{2.1}$$

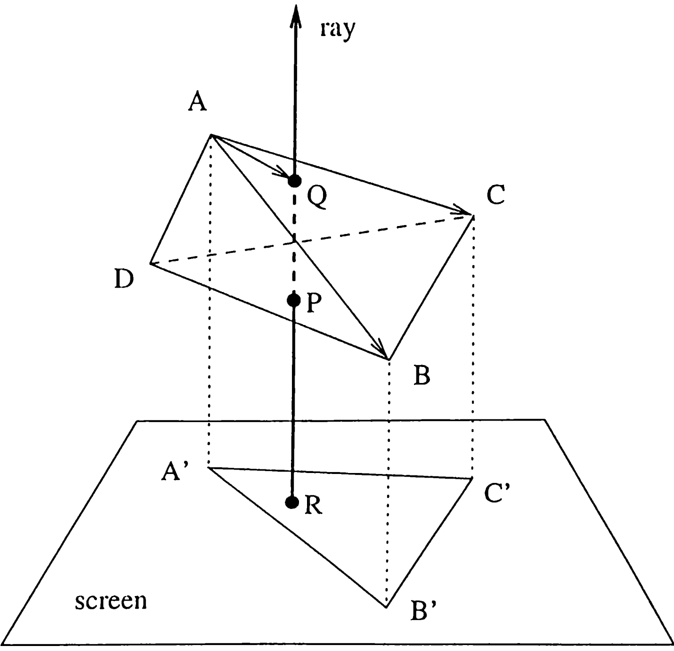


Figure 2.6. Intersection test.

Here, α and β are the coefficients used for scaling. The equation above can be rewritten in a more useful matrix form as follows:

$$\begin{bmatrix} B_x - A_x & C_x - A_x & 0 \\ B_y - A_y & C_y - A_y & 0 \\ B_z - A_z & C_z - A_z & 0 \end{bmatrix} \times \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} R_x - A_x \\ R_y - A_y \\ Q_z - A_z \end{bmatrix} \qquad (2.2)$$

In Equation 2.2, (A_x, A_y, A_z), (B_x, B_y, B_z), and (C_x, C_y, C_z) represent the coordinate values of A, B and C points, respectively. By solving the equations in the first and second lines of Equation 2.2, unknown α and β coefficients in Equation 2.1 can be calculated. If one of the $\alpha \geq 0$, $\beta \geq 0$, and $1 - \alpha - \beta \leq 0$ conditions does not hold, then the ray does not intersect the face an another face is tested in the same manner. The z coordinate of the exit point Q can be calculated by substituting the values found, into the Equation 2.3.

$$Q_z = A_z + \alpha(B_z - A_z) + \beta(C_z - A_z) \qquad (2.3)$$

2.3.2 Resampling

Next step in the algorithm is to find the scalar value at the exit point. That value is approximated by the interpolation of the scalar values at some sampling points. Koyamada's algorithm employs 2D inverse distance interpolation to calculate the scalar Q_s at point Q.In our example, the scalar values A_s, B_s, and C_s at face corners A, B and C are interpolated for this purpose. The α and β coefficients found during the intersection test is used in the following interpolation formula:

$$Q_s = \alpha B_s + \beta C_s + (1 - \alpha - \beta) A_s \tag{2.4}$$

Using (P_z, P_s) and (Q_z, Q_s) tuples, new scalars are calculated within the cell, by 1D inverse distance interpolation, along the ray segment bounded by the P entry, and the Q exit points. The number of resampling points depends on the method used. Equidistance sampling, adaptive sampling, and mid-point sampling are the most common resampling methods.

In equidistance sampling, the distance between successive resampling points is constant and scalars are calculated by 1D inverse distance interpolation using entry and exit points. Adaptive sampling takes the cell size variation into consideration and determines a different resampling distance for each cell. In mid-point sampling resampling is done at the middle of the ray-segment. The method used in this work is mid-point sampling. It is both fast, since the resampling is done just once for each cell, and is unaffected by the cell size variation in the volume. The scalar at the resampling point is calculated by the following simple formula, where M represents the resampling point:

$$M_s = \frac{P_s + Q_s}{2} \tag{2.5}$$

The scalar values obtained, are mapped into color and opacity values by transfer functions [24] λ_i, where $i \in \{r, g, b, o\}$. RGB color tuple determines the appearance of an object, and opacity is a property of the material which determines how much of the light is allowed through the object. By setting the transfer functions properly, some important features in the data and changes in the scalar values can be highlighted. Figure 2.7 shows two example transfer

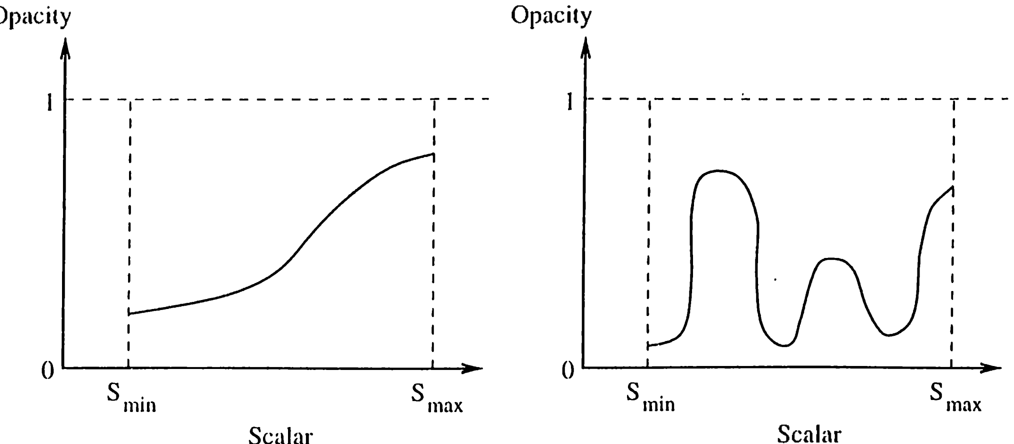


Figure 2.7. Example transfer functions.

functions. In the first graph, a smooth passage was supplied between opacities of close scalar values. In the second graph, some features at specific scalar value ranges were obscured by mapping those scalars to low opacity values.

The last step, after the colors and opacities for particular resampling points are found is to composite them using the color and opacity composition formulas:

$$O_{i+1} = O_i + \lambda_o(M_s)(1 - O_i) \tag{2.6}$$

$$R_{i+1} = (R_i O_i + \lambda_r(M_s)\lambda_o(M_s)(1 - O_i))/O_{i+1} \tag{2.7}$$

$$G_{i+1} = (G_i O_i + \lambda_g(M_s)\lambda_o(M_s)(1 - O_i))/O_{i+1} \tag{2.8}$$

$$B_{i+1} = (B_i O_i + \lambda_b(M_s)\lambda_o(M_s)(1 - O_i))/O_{i+1} \tag{2.9}$$

In the equations above, (R_i, G_i, B_i, O_i) and $(R_{i+1}, G_{i+1}, B_{i+1}, O_{i+1})$ values represent the color and opacity values composited before and after the resampling point M is reached, respectively. Also, the initial color and opacity values should be set as $O_0 = 0$, $R_0 = 0$, $G_0 = 0$, and $B_0 = 0$.

Finally, for each pixel the composited ray segments are collected in the ray buffers. If the ray shot from a pixel does not enter and exit the volume more than once, then there is just one ray segment in the buffer and its color is used to paint the pixel. Otherwise, the colors of the ray segments need compositing as it is done in the resampling points. In case no ray is fired from a pixel, a predetermined background color value is assigned to that pixel.

2.4 Optimizations and Performance

One of the optimizations of Koyamada's algorithm is on the scalar value calculation at the resampling points. Instead of making expensive 3D interpolations using the vertices of the tetrahedra, it performs a 2D inverse distance interpolation followed by a 1D inverse distance interpolation, which is much faster. Moreover, it makes use of the vector scaling coefficients found in the intersection test, during the data interpolation, and decreases the processing amount for this step. Our approach of using mid-point sampling decreases the number of resampling points taken and also prevents resampling some parts of the data unnecessarily.

Conventional techniques perform three intersection tests per cell. However, Koyamada's approach performs two tests per cell, on the average. This is because the exit point from a cell is used as the entry point to another cell. Hence object-space coherency is utilized.

In DVR methods, composition of the color and opacity can be done in back-to-front or front-to-back order. In object-space methods, which uses back-to-front order composition, scientists have the chance to view the image forming on the screen in an animation-like manner. That is, intermediate steps of final image appearance can be watched. On the other hand, front-to-back order composition can make use of the early ray termination. Early ray termination is an optimization technique, which stops the resampling operation when the accumulated opacity along a ray reaches unity.

There are some other optimization techniques such as pixel color interpolation over the image screen, but most of these techniques degrade the image quality. Since one of our aims is to produce high quality images, we preferred not to employ such optimization techniques in this work.

Performance of this algorithm is affected by four factors, that is, the times spent on node transformation, scan conversion, intersection test, and resampling. Node transformation is necessary to bring the volume from WSC to NPC. If N is the total number of nodes in the data, and the time to transform

a single point is t_{tr}, it can be formulated as:

$$T_{tr} = N t_{tr} \tag{2.10}$$

Note that, T_{tr} is independent of the visualization instance. Scan conversion cost on the other hand, is proportional to the sum of the areas of the *eff* faces, and can be affected by the viewing parameters. If *Area* is a function which returns the triangular area of a given face, and t_{sc} is the average time spent on the scan conversion of a pixel, then T_{sc} can be expressed as:

$$T_{sc} = \sum_{f \in \mathcal{F}_{eff}} Area(f) t_{sc} \tag{2.11}$$

Two other important costs are the times spent on intersection tests (T_{it}), and resampling operations (T_{rs}). Assuming W, H, I_{xy}, t_{it}, t_{rs} variables represent screen width, screen height, intersection count for a ray shot from (x, y) coordinate, average intersection time, and average resampling time, respectively; T_{it} and T_{rs} can be calculated as follows:

$$T_{it} = \sum_{x=0}^{x=W} \sum_{y=0}^{y=H} I_{xy} t_{it} \tag{2.12}$$

$$T_{rs} = \sum_{x=0}^{x=W} \sum_{y=0}^{y=H} I_{xy} t_{rs} \tag{2.13}$$

Note that, since we use mid-point sampling, the number of resampling points is equal to the number of intersection tests made. Therefore, I_{xy} can be used as the number of resamplings made along a ray. Considering our experiments about the weights of these four factors in the total execution time, the first two factors can simply be ignored. As a result, the computation time for the algorithm can be expressed by the following formula:

$$T = T_{it} + T_{rs} = \sum_{x=0}^{x=W} \sum_{y=0}^{y=H} I_{xy}(t_{it} + t_{rs}) \tag{2.14}$$

Chapter 3

Image-Space Parallelization

Parallelization of ray-casting can be done in object-space or in image-space. The focus of this work is on image-space parallelization. Load balance and remapping of data primitives to new processors gain importance in image-space parallelization. Therefore, it is important to handle this problems accurately and efficiently.

Next section gives a brief comparison of OS and IS parallelization. The section following it introduces our approach of using clusters instead of individual data primitives. Last two sections describe the load balance and remapping problems in IS parallelization.

3.1 OS versus IS Parallelization

In OS parallelization, decomposition is done in the object-space and some portions of the volume data are assigned to processors. The processors are responsible from rendering their own sub-volume. To obtain a load balance among the processors, the sub-volumes are determined such that their computational costs are nearly equal. The number of sub-volumes assigned to a processor can be more than one. Previously, some techniques such as *octree* [16, 17], *k-D tree* [2, 31, 33, 37], and *graph partitioning* [9, 32] were employed to find the appropriate data-processor assignments.

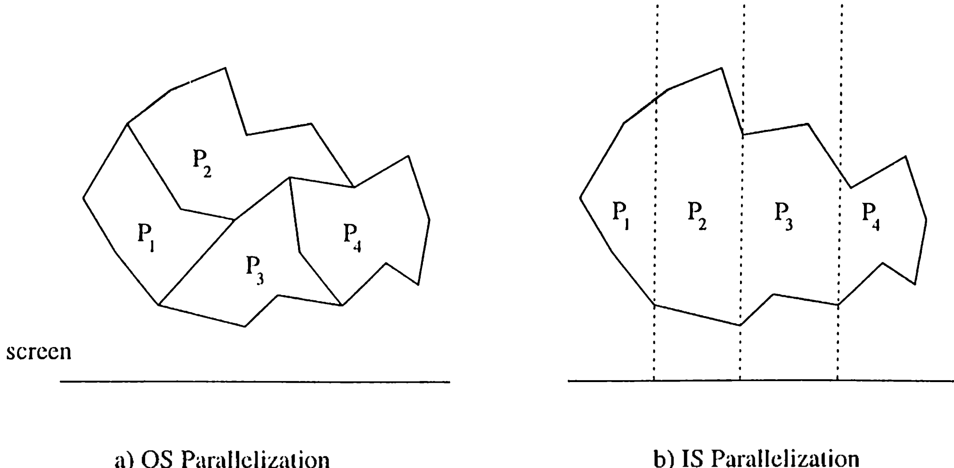


Figure 3.1. Data-processor assignments in OS and IS parallelization.

OS parallelization, since decomposition is done in the object-space, has the ability to establish a load balance among the processors. Changing viewing parameters do not disturb the existing load balance much. On the other hand, the need for compositing the ray-segments produced by the processors during their local rendering phase appears to be the major disadvantage of this method. Especially in unstructured grids, the number of ray-segments produced can be quite high in number, and may cause excessive communication costs.

The other option for parallelization is IS parallelization. In this method, instead of creating chunks of data and assigning them, each processor is given a screen region and works only on the data whose projection fall onto that screen region. A processor is given all the primitives needed to render its region beforehand, and no global pixel merging operation is necessary. To divide the screen into sub-regions, techniques such as *quad trees*, *jagged partitioning*, *recursive subdivision* had been used in the literature.

Load imbalance and communication costs during the data migration are the two important problems in IS parallelization, that will be discussed in more detail in the following sections. In DVR methods using unstructured data, the cell size variation is the basic reason for load imbalance, and remapping models can be used to ensure an acceptable load balance as well as minimizing the communication costs. Figure 3.1 displays data-processor assignments in

OSP and ISP, in a simplistic manner.

Compared to the OS parallelization, IS parallelization produces faster code execution. It is shown in [43] that the communication required by IS parallelization is usually higher than the one in OS parallelization. This makes remapping more important for IS parallelization.

3.2 Clusterization

To obtain a good load balance in IS parallelization, it is necessary to know the work load distribution over the screen pixels. In other words, I_{xy} should be known at each pixel prior to rendering. If individual tetrahedral cells are used during screen work load calculations, the amount of preprocessing overhead incurred at each visualization instance makes the model impractical to use. Hence, in this work, a clusterization step was employed to decrease this preprocessing overhead.

In this approach, each cluster contained a number of cells. The basic aim was to create clusters with equal cell rendering costs and with minimal surface area. Minimizing surface area leads to sphere-like cell clusters which in turn minimizes the interaction of the clusters with the screen. Therefore, both less scan-conversion is performed during work load calculation and a more contracted hypergraph can be obtained and used during remapping.

Since volumetric data is mostly produced by engineering simulations on parallel computers, we simply assumed that each processor acquired some chunk of the volume data previously. Instead of using a global clustering scheme, we employed a local clustering scheme. Every processor worked on its initially assigned data in parallel to produce the cell clusters. This eliminated the cost that will be incurred by global clustering.

3.2.1 Graph Partitioning

Graph partitioning is the method and state of the art METIS graph partitioning tool is the tool we used to form the cell clusters. Graph partitioning is a technique for assigning some tasks to partitions so as to balance the load of partitions and minimize the interaction between partitions. It arises in a variety of computing problems, such as VLSI design, telephone network design, and sparse gaussian elimination.

In our clusterization model, clusters correspond to partitions and cells correspond to tasks. We consider an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, W_{\mathcal{V}}, W_{\mathcal{E}})$ without loops and multiple edges. Here, vertex set $\mathcal{V}$ is the set of tetrahedral cells, and edge set $\mathcal{E}$ is the set of faces connecting these cells. A cell v_i is said to be connected to another cell v_j by an edge e_{ij}, if they share a common face. Vertex weights, $W_{\mathcal{V}}$, represent the cell rendering costs, and edge weights, $W_{\mathcal{E}}$, corresponds to the amount of interaction between the cells.

Graph partitioning a set $\mathcal{V}$ means dividing it into P disjoint, non-empty subsets whose unions form $\mathcal{V}$:

$$\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \ldots \cup \mathcal{C}_P \tag{3.1}$$

This partitioning is done considering a partitioning constraint and an optimality condition. Firstly, the sums of the weights $W_{\mathcal{C}_i}$ of nodes v_i in each $\mathcal{C}_i$ should be approximately equal. This means that the rendering costs of the clusters are nearly equal. Secondly, the sum of the weights $W_{\mathcal{E}'}$ of edges

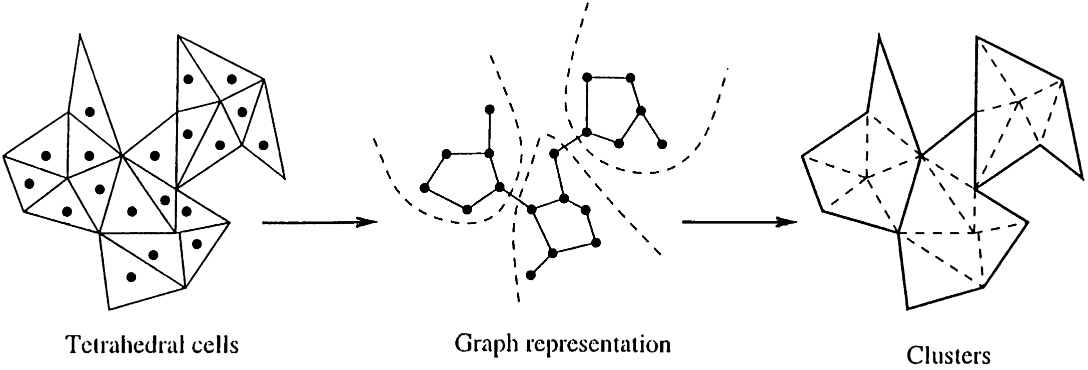


Figure 3.2. Cell clusterization using graph partitioning.

e_{kl} connecting the nodes in different partitions C_i and C_j should be minimized. This means that the total amount of interaction between clusters is minimized. Figure 3.2 shows this clusterization process.

The total number of clusters in the whole system can be chosen from the numbers between the number of processors, K, and the number of cells in the data, $|\mathcal{V}|$. Choosing the total cluster number near $|\mathcal{V}|$ may degrade the performance causing a huge hypergraph, and using a number near K may degrade the quality of the load balance in rendering phase. Hence, we prefer to determine this number empirically. Chapter 7 discusses some results found on this problem.

3.2.2 Weighting Scheme

There are six possible weighting scheme combinations that can be used to determine the vertex and edge weights of the clusterization graph. These six possibilities are shown in Table 3.1. The symbols CV, CA, FA denote the cell volume, cell area, and face area, respectively. Unit cost scheme is represented by 1.

For the edge weights it is more intuitive to use the FA scheme, since the total face area of a cluster better represents that clusters interaction with the other clusters and the screen. A cluster with large face areas has a higher chance to be hit by a ray. Using FA as edge weight produces spherical clusters which does not change their rendering loads by an important amount at different

Table 3.1. Possible weighting schemes for the clusterization graph.

Vertex weight	Edge weight
1	1
1	FA
CA	1
CA	FA
CV	1
CV	FA

visualizations.

Also, we set all vertex weights to 1 in the clusterization graph. This scheme produces clusters with equal cell size, and hence communication cost. The variation in the number of cells in clusters may be huge in CV and CA weighting schemes. Experimental results in Chapter 7 verifies our selection.

3.2.3 Additional Data Structures

Clustering the volume data requires the use of additional data structures. Each cluster is given a global identifier and a global *ClusterMap* array is created in every processor. This array is used to obtain the cluster-processor mapping and also to reach the data contained within a cluster. Each element in this array maintains a pointer to the clusters local data, and a processor id showing the processor in which the cluster resides.

Processors keep only the data contained in their assigned clusters. Since each cluster stands as an entity that can be rendered independently, the data in the clusters are treated as local data. The *Cluster* data structure contains two arrays, namely local *Nodes* and *Cells* arrays. Local indexing is utilized within these arrays.

Furthermore, the *Cell* data structure introduced in Chapter 2 is modified such that now it includes information about the cluster identities, showing the neighbor cluster for each face of a cell. Since a ray can leave a cluster from an *cbf* face and enter into another cluster, it is necessary to know this new clusters identifier as a connectivity information. For internal faces the identifier of the cluster in which the face is located is assigned as the neighbor cluster identifier. These new and modified data structures are shown in Figure 3.3.

3.3 Load Balancing

Load balancing is one of the primary concerns in parallel applications. Without proper arrangement, an idle processor may drag the performance of the system

downward. In parallel volume rendering, the rendering load shared among the processors should be balanced.

In structured grids, load balancing is a relatively simple task. However, the lack of a simple indexing scheme in unstructured grids makes visualization calculations on such grids very complex. Furthermore, unstructured grids contain data cells which are highly irregular in both size and shape. In a distributed computing environment, irregularities in cell size and shape make balanced load distribution very difficult.

3.3.1 Screen Subdivision

Since the aim in this work is image-space parallelization, the screen pixels and hence the load spread over them is tried to be equally shared among the processors. Previous work on screen space subdivision methods includes the use of quad-trees, recursive bisection and jagged partitioning. All these methods are common in that they try to divide the screen into rectangular pieces and distribute these screen regions to processors. However, since the division lines separating the regions are always parallel to the coordinate axis these subdivision techniques are not flexible enough and may not always produce perfect load balancing. Figure 3.4 displays these techniques on a screen with discrete load assignment.

The most flexible screen subdivision technique would be the one which makes the partition boundaries as flexible as possible, that is, sub-screen boundaries would be able to change to any shape. Unfortunately, this is not

```
struct Map {
 int proc_id;
 Cluster *Clusters;
}

struct Cluster {
 Cell *Cells;
 Node *Nodes;
 int CellCount;
 int NodeCount;
}

struct Cell {
 int vertices[4];
 int neighborCells[4];
 int neighborFaces[4];
 int neighborClusters[4];
}
```

Figure 3.3. Additional data structures used after clusterization.

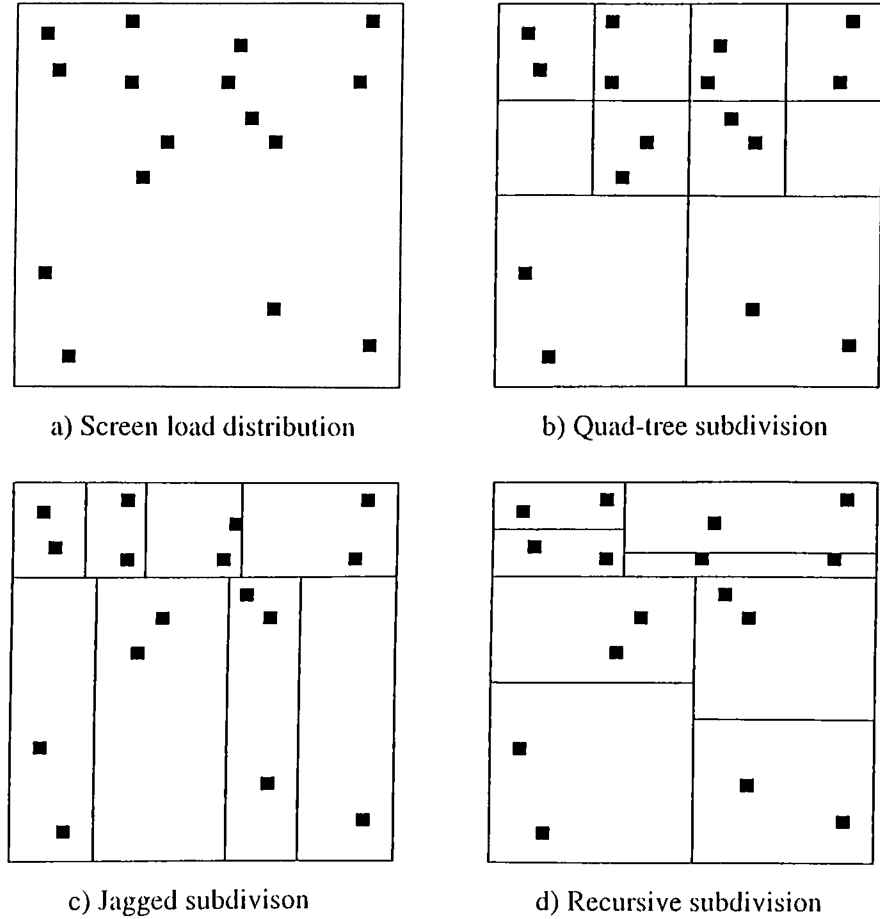
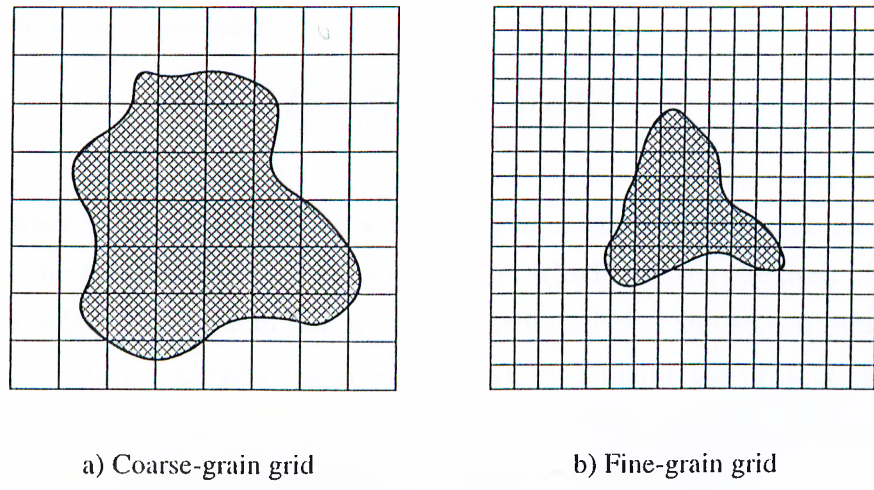


Figure 3.4. Screen subdivision techniques.

practical for two reasons: First, excessive amount of processing is needed to determine and manipulate the non-regular sub-screen boundaries, and second, the data structures to represent the boundaries would be too complex and might require too much storage.

In this work, the screen is subdivided by an n by n cartesian grid forming n^2 sub-regions, and this sub-regions are assigned to processors for rendering. From now on, we will refer to these screen sub-regions as *screen cells*. We represent the set of screen cells by $\mathcal{S}$. An individual screen cell in this set is represented by r_i.

Since the projection area of the volume occupying the screen changes with respect to the visualization parameters, our approach requires the estimation of the grid granularity. In our work, the number of pixels in screen cells is



a) Coarse-grain grid b) Fine-grain grid

Figure 3.5. Effect of projection area on grid granularity.

determined adaptively. This is done by keeping the number of occupied (having a load) screen cells constant at every visualization instance. This also prevents the variation in view dependent preprocessing time. Figure 3.5 shows the grid granularities for two different projection area size. Note that, the number of occupied regions is nearly the same in both cases in the figure.

Adjusting the grid granularity such that each screen cell will contain just one pixel results in the most flexible screen cell boundaries mentioned above. However, increasing view dependent preprocessing overhead makes this approach infeasible to use. On the contrary, if the number of screen cells is kept low, the solution space of the load balancing problem is reduced and satisfactory load balance values cannot be obtained. We used an engineering formula, which is explained in Chapter A, to determine the appropriate granularity of the grid in an adaptive manner.

3.3.2 Work Load Calculation

In image-space parallelization, screen subdivision is not enough by itself to obtain a good load balance. Also, the work load distributed over the pixels should be calculated correctly. The total work of rendering a cluster is approximated

by the following formula:

$$Load(\mathcal{C}) = \sum_{f \in \mathcal{F}_{ff}} Area(f) \tag{3.2}$$

In other words, the work for a cluster is equal to the sum of areas of the *ff* faces in the cluster. Here, *Area* is a function which returns the area of a given triangular face. The calculated work is distributed over the screen pixels (over the screen cells in case of a coarse grain grid), by utilizing the projection area of the cluster. In this work, we have tested three different work load assignment schemes, using different bounding boxes for the clusters.

1. *Cluster Bounding Box*: This is a rather rough estimation of the pixels affected by a cluster. The box surrounding the clusters projection area on the screen is found and all pixels in that box are assigned the same load value. The assigned load per pixel is equal to the ratio of $Load(\mathcal{C})$ to the bounding box area. In this method, many pixels with actual work load of zero is assigned a load. Especially, if the cluster is disconnected or has a low concavity, work load distribution on the screen can be very poor. On the other hand, since the calculation of cluster bounding box is a simple min-max operation performed over the vertices of a cluster, this method is very fast.

2. *Cell Bounding Box*: Instead of using the whole clusters bounding box, bounding boxes for all *eff* faces are calculated and a smaller affected region is found. The total work of the cluster is shared among those pixels in the affected region. Compared to the previous method, this method produces much better load distributions. Only problem arises with the faces having a thin, long shape. For cells with such properties, the bounding boxes are too large and this increases the amount of error made.

3. *Inside-Outside Test*: Among these three methods, this is the one which produces most accurate load distributions. The effect area of the cluster is calculated by performing inside-outside test on each *eff* face in the cluster. This test locates the pixels inside a triangular face in an exact manner.

Figure 3.6 displays these three work load assignment methods. The accuracy of inside-outside test in load distribution is not its only advantage. It also allows a correct topology for the hypergraph used in the remapping stage to be established. This means reduction in the hypergraph partitioning overhead, and prevention of unnecessary communication during cluster migration.

3.4 Remapping

Load balancing by itself is not enough for a good parallelization. As the visualization parameters change, there occurs differences in load distributions on the screen. This requires the remapping of screen regions to processors, since each processor, now, holds unequal amount of rendering work. Also, since in image-space parallelization a processor needs all the volume data above its assigned screen regions, it is necessary to exchange some clusters between processors. The amount of communication performed during cluster migration due to the remapping should be minimized.

This remapping problem is an NP-hard problem for our case. There are some heuristics giving suboptimal solutions, which are used to solve this problem. In this work, we offer a hypergraph partitioning model as a solution to this remapping problem. The details of our model can be found in a separate chapter, Chapter 5, which we reserved for the explanation of the model.

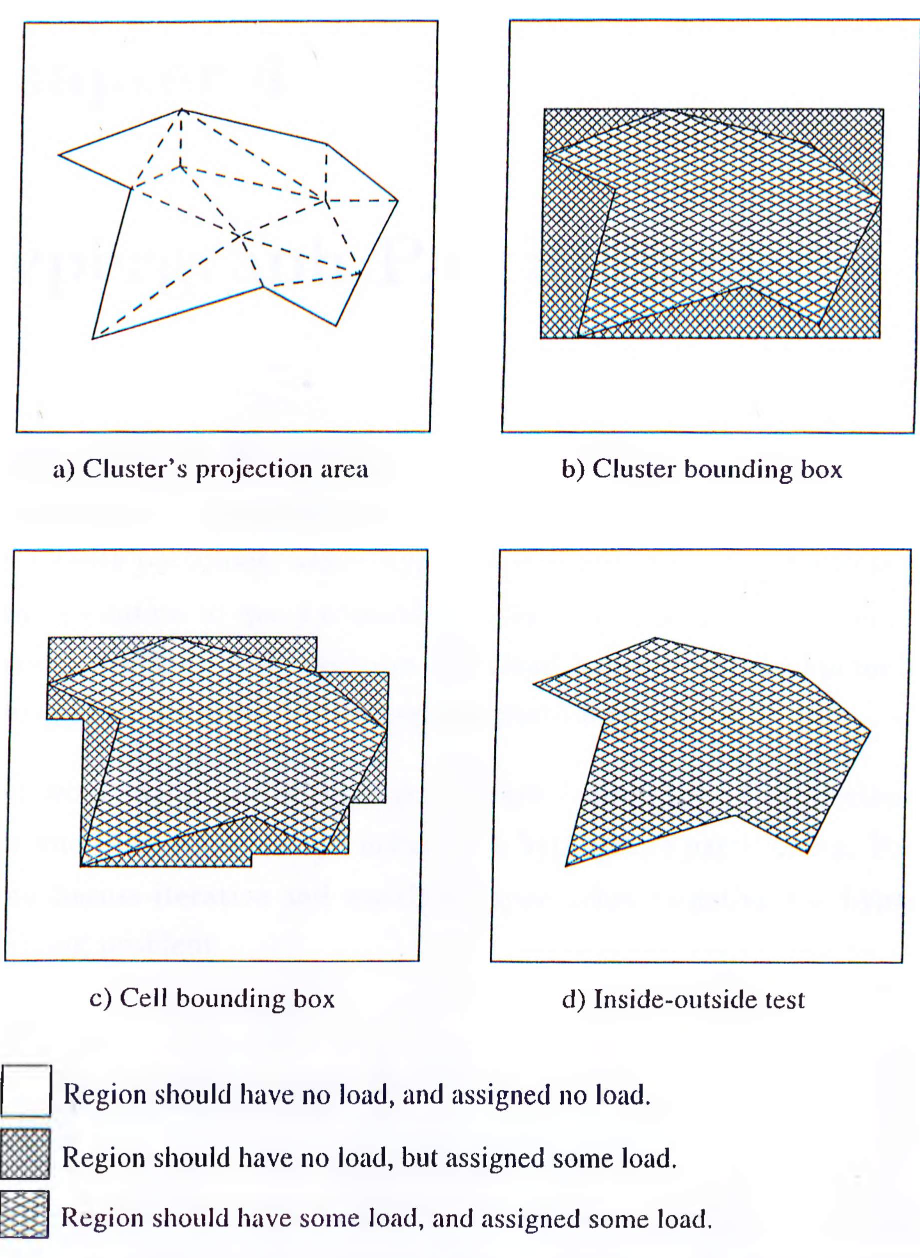


Figure 3.6. Work load assignment schemes.

Chapter 4

Hypergraph Partitioning

Hypergraph models lately began to attract interest in academia. With the development of hypergraph partitioning tools which run faster and produce higher quality partitions; today, hypergraph partitioning was begun to be considered as solution to many research problems. VLSI design, data mining, and in general, problems which require both load balance and remapping are the sample application areas for hypergraph partitioning.

Our work makes use of hypergraph partitioning, too. Next section introduces some basic concepts and notation in hypergraph partitioning. Following sections discuss iterative and multilevel approaches targeting the hypergraph partitioning problem.

4.1 Introduction

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ contains a set of vertices $\mathcal{V}$ and a set of nets $\mathcal{N}$. The nets n_i in set $\mathcal{N}$ are some subsets of the vertices in $\mathcal{V}$. The set of vertices forming a net n_i, called its pins, is denoted as $Pins(n_i)$. The same operator can be used to represent the pins of a set of nets $\mathcal{N}'$:

$$Pins(\mathcal{N}') = \bigcup_{n_i \in \mathcal{N}'} Pins(n_i) \tag{4.1}$$

The nets connected to a vertex is found by the *Nets* operator, that is, $Nets(v_i)$ returns the nets n_j such that $v_i \in Pins(n_j)$. The size of a net n_i is equal to the number of its pins, and the degree of a vertex v_j is equal to the number of nets it is connected to, that is, $s_i = |Pins(n_i)|$, and $d_j = |Nets(v_j)|$, respectively. Each vertex $v_i \in \mathcal{V}$ has an assigned weight w_i. Similarly, each net $n_i \in \mathcal{N}$ has a cost c_i.

For a partition $\Pi = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_K\}$ to be a K-way partition for a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, the following three conditions must hold:

1. $\mathcal{P}_k \subset \mathcal{V}$, $\mathcal{P}_k \neq \emptyset$, for $1 \leq k \leq K$
2. $\bigcup_{k=1}^{K} \mathcal{P}_k = \mathcal{V}$
3. $\mathcal{P}_k \cap \mathcal{P}_l = \emptyset$, for $1 \leq k < l \leq K$

A net is connected to a part, if it has at least one pin in that part of the partition. Connectivity set, Λ, of a net n_i is the set of parts to which n_i is connected. Connectivity of a net, n_i, is equal to the size of its connectivity set, that is, $\lambda_i = |\Lambda_i|$. A net with a connectivity of 1 is called as an internal net. If $\lambda_i > 1$, the net is an external net. An external net is said to be at cut. We denote the set of external nets which has a pin on a vertex set $\mathcal{V}'$ by $\mathcal{N}_E(\mathcal{V}')$.

The weight of a part $\mathcal{V}_i$ is denoted by W_i and is equal to the sum of the vertex weights in part $\mathcal{V}_i$. To determine the overall load imbalance between the parts a value of ϵ is used.

Given all these definitions, K-way hypergraph partitioning problem can be defined as finding a partition Π for a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, such that the weight of each part is bounded, and a function defined over the nets is optimized. The first condition is called partitioning constraint. If W is the sum of part weights, this condition can be formulated as follows:

$$W/(\epsilon K) \leq W_i \leq \epsilon W/K, \qquad \text{for } 1 \leq i \leq K \tag{4.2}$$

The requirement that a function is tried to be optimized, is referred as the partitioning objective. There are several objective functions developed and

used in the literature. One of the most popular objective functions uses the cutnet metric. According to this metric cost of a partition is equal to the sum of the weights of external nets:

$$\chi(\Pi) = \sum_{n_i \in \mathcal{N}_E(\mathcal{V})} c_i \tag{4.3}$$

Another widely used metric is the (λ-1) connectivity metric. In this metric each net contributes $c_i(\lambda_i - 1)$ to the cost:

$$\chi(\Pi) = \sum_{n_i \in \mathcal{N}_E(\mathcal{V})} c_i(\lambda_i - 1) \tag{4.4}$$

4.2 Partitioning Methods

A nice survey by Alpert and Kahng [1] classifies the partitioning methods under four main categories: Move-based approaches, geometric representations, combinatorial formulations, and clustering approaches. Among these, move-based approaches are the ones which attract the most attention in the literature. They are known to be the most successful ones in terms of both speed and solution quality. The same survey mentions iterative improvement [10, 23], simulated annealing [25], and tabu search [13] as the methods used in move-based approaches. Since the partitioning tool used in this work is a multilevel hypergraph partitioning tool which makes use of iterative improvement methods, in this section, we concentrated only on these types of work.

The term *bisection* is used to mean a two-way partitioning of a hypergraph with load constraints on part weights. We used the term *multi-way partitioning* to refer the partitioning where the number of parts produced is more than two.

4.2.1 Iterative Improvement Methods

Iterative improvement methods uses greedy strategy. Given an initial feasible solution, they try to reach to a better solution by making changes on the current solution iteratively. Search for a solution stops when all neighbor solutions are

worse than the current solution. Since the operations performed are simple vertex move or swap operations, these heuristics can easily get stuck in a local optima. Hence, some extended data structures can be used to empower them with the capability to climb out of local optima.

One of the earliest bisection heuristic is the KL heuristic which was proposed by Kernighan and Lin [23]. This heuristic was originally for graphs and later extended to hypergraphs by Schweikert and Kernighan [39]. In their algorithm, a series of passes is performed over the vertex set of the hypergraph, in which every vertex is unlocked initially. During a pass an unlocked vertex in part P_1 is swapped with an unlocked vertex belonging to part P_2. After a vertex is swapped it is locked and cannot again be swapped within the same pass. The vertices to be swapped are chosen such that the gain, e.g. the decrease in the bisection cost, is maximum. In order to climb local optima, swap gains with negative values are allowed. At each swap in a pass, cost of the current bisection is recorded. When all vertices are swapped the pass ends, and the bisection encountered with the lowest cost is returned as the solution of this pass. This bisection is used as the initial solution in the following pass. The whole algorithm terminates when a pass fails to find a better solution than its initial solution. KL heuristic often lasts in a few passes.

If n is the number of vertices, complexity of KL algorithm is $O(n^2 \lg n)$. Its usage may be limited, since it works on hypergraphs having vertices with no weights. An improvement performed over KL is the FM heuristic introduced by Fiduccia and Mattheyses [10]. Execution time of their algorithm is linearly proportional to the number of pins in the hypergraph, that is, it has complexity $O(p)$. In general, FM is very similar to KL. Main difference appears during the process performed to find a neighbor solution. In FM, instead of swapping with another one, a vertex is directly moved to the other part and gain calculations are done accordingly. The gain associated with moving a vertex v from P_i,that is, $\gamma(v)$ is the following:

$$\gamma(v) = |\mathcal{N}_E(P_i)| - |\mathcal{N}_E(P_i - \{v\})| \tag{4.5}$$

Since, vertices may have weights, care must be taken on the partitioning constraint. Moving a vertex to the other part can bring the heuristic to an

infeasible solution. As a precaution, the arrived solution is permitted to deviate from the exact bisection by the weight of the heaviest vertex. The remaining parts of FM are almost the same with KL. During a pass, the best solution observed is returned as an intermediate solution, and the algorithm terminates when a pass fails to find a better solution than its starting solution.

Hagen et al. [14] have shown that in some cases there may be many vertices with equal move gains during an FM pass. This decreases the chance to choose the best vertex to be moved. Hence, it is a good idea to include a tie breaking mechanism in the heuristic. In order to break the ties, Krishnamurthy [27] suggested an improvement over FM by adding some lookahead capability. He used a gain vector of size r for each vertex, and kept the potential gains from feature moves up to the rth move, in these vectors. When a tie occurs gain vectors are checked for higher gain levels until the tie is broken. The rth level gain of moving v from $\mathcal{P}_1$ is calculated as

$$\begin{aligned}\gamma_r(v) = & \quad \mid \{n \in \mathcal{N}_E(\{v\}),\ B_{\mathcal{P}_1}(n) = r,\ B_{\mathcal{P}_2}(n) > 0\} \mid \\ & - \mid \{n \in \mathcal{N}_E(\{v\}),\ B_{\mathcal{P}_1}(n) > 0,\ B_{\mathcal{P}_2}(n) = r-1\} \mid \end{aligned} \qquad (4.6)$$

Here, $B_{\mathcal{P}_i}(n)$ is the binding number defined by Krishnamurthy. In case there does not exists a locked vertex in this set, binding number is equal to the number of unlocked vertices in the set $\mathcal{P}_i \cap Pins(n)$. Otherwise, $B_{\mathcal{P}_i}(n)$ is assigned infinity. To be more clear, his approach counts the number of vertices that must be moved from P_i in order remove n from the cut. Note that, for $r = 1$, Equation 4.6 reduces to Equation 4.5. Krishnamurty's approach is later extended to multi-way hypergraph partitioning by Sanchis [38].

4.2.2 Multilevel Methods

Due to several reasons, the partitions produced by these iterative partitioning methods may be poor in terms of both partition quality and speed. Also, the quality of the partitions may be far from the best solution by a large margin, making the correct prediction of the resulting solution quality very difficult. Multilevel methods [6, 15, 21, 22] are proposed to enhance the existing iterative

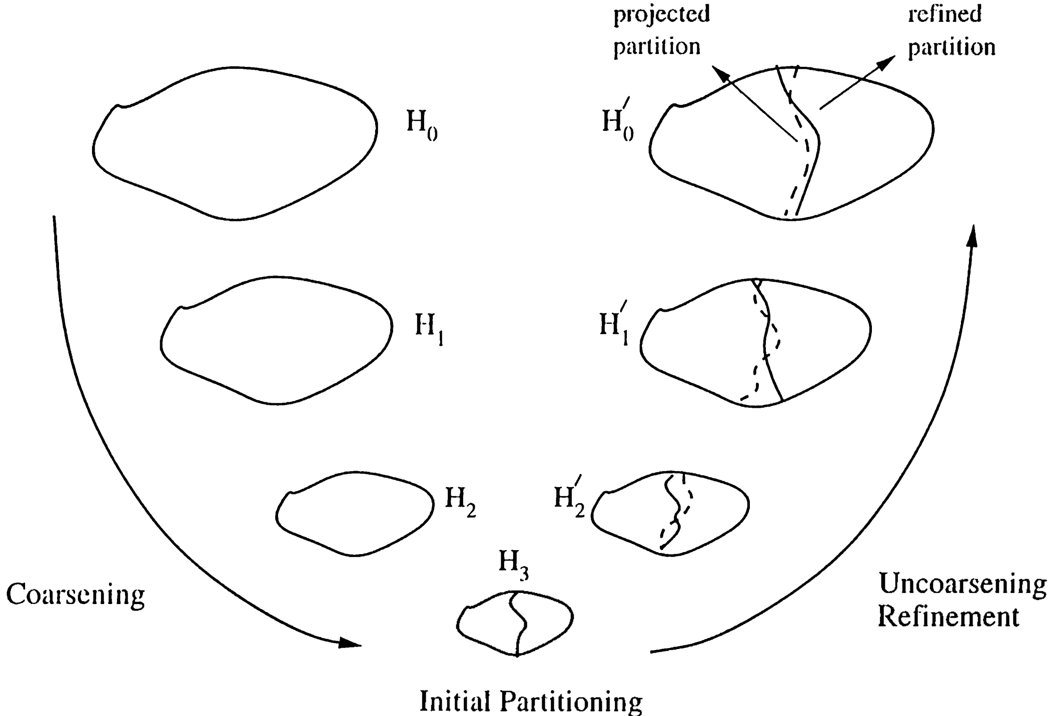


Figure 4.1. Multilevel hypergraph bisection.

methods. HMETIS [19] and PaToH [5] are the two example partitioning tools, using this multilevel paradigm.

A multilevel bisection scheme is composed of three consecutive steps: Coarsening, initial partitioning, and uncoarsening. In coarsening phase, highly interacting vertices of the hypergraph are grouped as multinodes, and a coarser hypergraph is constructed. Coarsening proceeds in a number of passes over the coarser hypergraphs, until the coarsest hypergraph with the desired vertex count is obtained. Initial partitioning phase tries to partition this coarsest hypergraph into two parts having equal sizes. In uncoarsening phase, these two parts are successively projected back on the previous finer hypergraphs. Figure 4.1 displays a sample multilevel bisection process, composed of three coarsening and uncoarsening levels.

For multi-way partitioning recursive bisection can be used. In recursive bisection scheme, a two-way bisection obtained from multilevel bisection is further partitioned in a recursive manner. It takes lg_2K bisection levels to partitioning a hypergraph into K parts. Our work uses recursive bisection

while partitioning the remapping hypergraph mentioned in Chapter 5.

4.2.2.1 Coarsening

Using the original hypergraph $\mathcal{H} = \mathcal{H}_0 = (\mathcal{V}_0, \mathcal{N}_0)$ a set of smaller hypergraphs $\mathcal{H}_1$ through $\mathcal{H}_m$ is constructed. Each level in coarsening takes the previous hypergraph as input and produces a coarser hypergraph. In each coarser hypergraph there are fewer number of vertices. Also, an increase in the vertex degrees is observed.

Coarsening of a hypergraph is performed by clustering two or more vertices together in the same multinode. Clustering can be done by matching based or agglomerative techniques. In matching based clustering, each vertex in the hypergraph is visited in a random or predetermined order and grouped with an unmatched vertex, marking the vertices in the group as matched. The multinode created forms a single vertex in the succeeding coarser hypergraph. In agglomerative clustering more than two vertices can involve in the same multinode, and a single vertex can join to a multinode. In the literature, there are several metrics used for determining the vertex visit order, and finding the highly interacting vertices that should be grouped.

4.2.2.2 Initial Partitioning

In this phase, the coarsest hypergraph $\mathcal{H}_m$ is bisected into two nearly equal-sized parts. For this purpose, PaToH hypergraph partitioning tool uses an algorithm called Greedy Hypergraph Growing. In this algorithm, a cluster is grown around a randomly selected vertex by moving some vertices into the cluster. Vertices are moved into the cluster according to their gains starting from the vertex with the highest gain. Growth of the cluster stops when a fixed load balance criteria between the two parts is satisfied. The vertices contained in the cluster are assigned to the first part, and the remaining vertices are assigned to the second part, giving a bisection of the coarsest hypergraph $\mathcal{H}_m$.

Another approach, here, could be to partition $\mathcal{H}_m$ into K parts directly,

leading to a direct K-way algorithm [20]. In this algorithm, K parts can be obtained by coarsening the original hypergraph until K vertices are left in the coarsest hypergraph. Also, a recursive bisection algorithm can be used to compute the K parts in the partition.

4.2.2.3 Uncoarsening

During the uncoarsening phase, each coarser hypergraph is projected back on the corresponding finer hypergraph in the previous level. Also, after finer hypergraphs are obtained, a refinement heuristic, similar to FM or KL is used to improve the partition quality of the current hypergraph. While minimizing the objective function, care is taken not to violate the load constraint on the parts.

Chapter 5

A Remapping Model

In IS parallelization, the load distribution and hence the computational structure of the problem may vary largely with changing visualization parameters. The existing screen-processor and data-processor assignments may turn into poor mappings, disturbing the load balance, and hence increasing the execution time.

This chapter is dedicated to a hypergraph partitioning model which is proposed as a solution to the remapping problem in IS parallelization. In the first section, some definitions are given and our two-phase and one-phase solutions to the problem are explained. Next section, discusses the effects of initial data distribution on remapping. Finally, a comparison of our model with jagged partitioning model is given.

5.1 Remapping by Hypergraph Partitioning

For our remapping model, we used the hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ to represent the computational structure of the problem, and to establish the interaction between the object-space and image-space. The vertex set $\mathcal{V}$ in the hypergraph corresponds to the set of screen cells, $\mathcal{S}$. These cells are found during screen sub-division, by imposing a coarse grid on the screen, as explained in Chapter 3, and they correspond to the atomic tasks which are to be individually processed

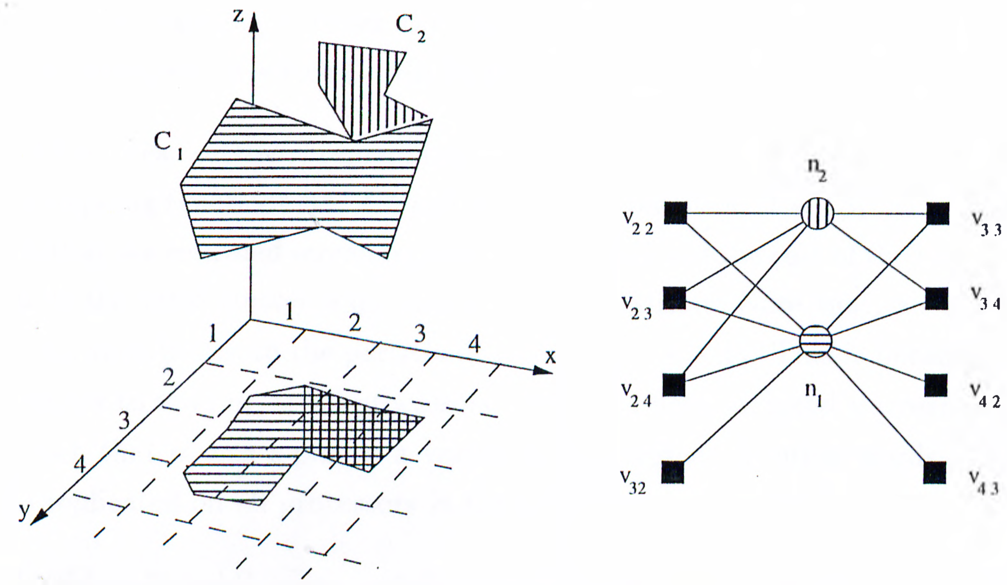


Figure 5.1. Representing the interaction between OS and IS by an hypergraph.

and completed by a processor. The weight of a vertex is assigned such that it is equal to the rendering load of the pixels in the screen cell, which the vertex is representing. In other words, for each vertex v_i, $w_i = Load(r_i)$.

Similarly, we use the nets in the hypergraph to represent the data clusters. From now on, we will use the words cluster and net interchangeably, to mean the same thing. As weight of a net, c_i, the communication cost for a cluster, $Cost(\mathcal{C}_i)$ is assigned. Here, $Cost$ is a function calculating the number of bytes transmitted, in order to send a clusters data and connectivity information from one processor to another. As a result, in this model, minimizing the cut of the hypergraph corresponds to minimization of the total volume of communication. Pins of a net is used to mark the screen cells occupied by the projection area of the cluster that the net represents. We call a net, n_i, with $s_i = 1$ a virtual net. Virtual nets appear in case a clusters area completely falls within the boundaries of a screen region. There may be a vertex, v_i, with $d_i = 1$, too. This is seen when just one clusters projection area occupies the screen cell. See Figure 5.1 for an example hypergraph.

Finally, two mapping functions $\mathcal{M}$, and $\dot{\mathcal{M}}$ are needed. $\mathcal{M}$ is used to obtain the mapping between screen regions and the processors. In other words, $\mathcal{M}(\mathcal{S}_i)$ returns the processor P_j to which the screen region $\mathcal{S}_i$ is assigned. $\dot{\mathcal{M}}$ is used

to find the mapping between the clusters and processors. $\tilde{\mathcal{M}}(\mathcal{C}_i)$ returns the set of processors in which $\mathcal{C}_i$ will be replicated.

After these settings, remapping problem reduces to the problem of finding two mapping functions $\mathcal{M}$, $\tilde{\mathcal{M}}$, and obtaining a partition $\Pi = \{\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_K\}$, such that the returned screen regions are balanced and the cost of the cut, $\chi(\Pi)$, that is the total cluster migration cost is minimized. Note that, an external net, n_i, contributes to the partitioning cost by $c_i(\lambda_i - 1)$. This is because, it is necessary to send n_i, to all processors which had been assigned a screen region $\mathcal{S}_i$, such that n_i has a pin on a vertex in that region. In other words, cluster $\mathcal{C}_i$ is replicated on all processors in the following set, P':

$$\tilde{\mathcal{M}}(\mathcal{C}_i) = P' = \{P_j : \exists v_k, v_k \in \mathcal{S}_l, v_k \in Pins(n_i), \mathcal{S}_l \in \Lambda_i, M(\mathcal{S}_l) = P_j\} \quad (5.1)$$

It is clearly seen from Equation 5.1 that $\lambda - 1$ metric should be as the partitioning objective function. Using the cutnet metric results in an incorrect estimation of the actual cut cost, since it may be necessary to replicate a cluster in more than one processor.

5.1.1 Two-Phase Hypergraph Partitioning Model

As the name suggests, the two-phase model hypergraph model proceeds in two phases. In the first phase, hypergraph partitioning is performed over the remapping hypergraph $\mathcal{H}$, and K screen regions are obtained. These screen regions are used as input into the second phase, which produces the mapping function $\mathcal{M}$. Without the second phase, that is, in the absence of a mapping function, each screen region $\mathcal{S}_i$ may be assigned to a processor P_j arbitrarily.

Since making this assignment arbitrarily may create a poor matching in terms of communication overhead, we apply a bipartite graph matching algorithm in this step to find better screen-processor assignments. The K processors used and the K screen regions produced by partitioning Π form the partite nodes $\mathcal{X}$ and $\mathcal{Y}$ of the bipartite graph $\mathcal{B} = (\mathcal{X}.\mathcal{Y}, \mathcal{Z})$. In this graph, vertices x_i and y_j denote processor P_i and screen region $\mathcal{S}_j$, respectively. An edge z_{ij} is placed between two vertices x_i and y_i, if there exists a cluster which is stored

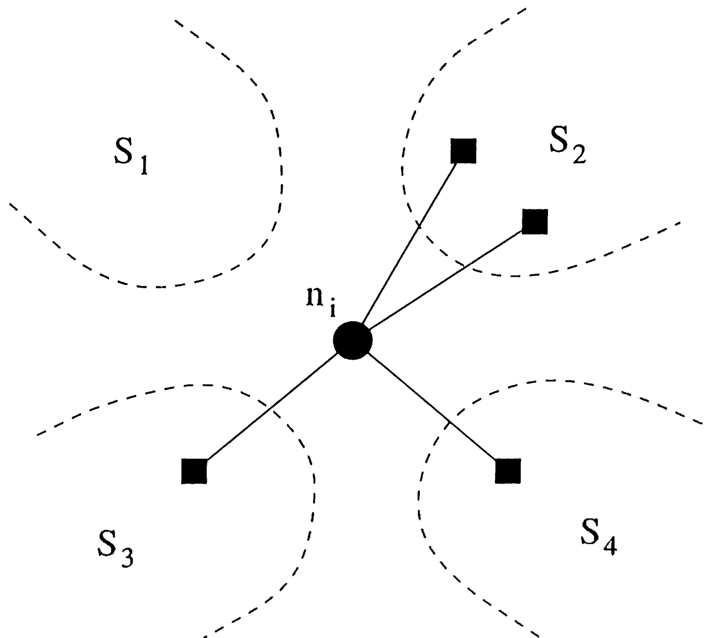


Figure 5.2. The partitioning cost calculated by two-phase method may be incorrect.

by processor P_i, and if that clusters projection area falls onto the screen region $\mathcal{S}_j$. The migration costs c_k of such clusters are summed and assigned as the weight of the edge z_{ij}, that is, W_{ij}.

After the bipartite graph is created, we use a maximum weight bipartite graph matching algorithm to find the region-processor mappings. The mapping found is an optimal solution, and using this mapping together with Equation 5.1, $\tilde{\mathcal{M}}$ can be calculated.

5.1.2 One-Phase Hypergraph Partitioning Model

The most important point ignored by the two-phase model is that each cluster is originally owned by a processor. We can prove that for some clusters, the cost added to the cut is in fact a wrong cost. Consider the cluster n_i shown in Figure 5.2. Assume that, n_i has pins on parts $\mathcal{S}_2$, $\mathcal{S}_3$, $\mathcal{S}_4$, and $M(\mathcal{S}_1) = P_1$, that is, $\mathcal{S}_1$ is assigned to P_1. In this situation, the cluster-processor assignment before the remapping would gain importance. If C_i is kept in the memory of a processor other than P_1, the cost contributed to the cut would be $2c_i$,

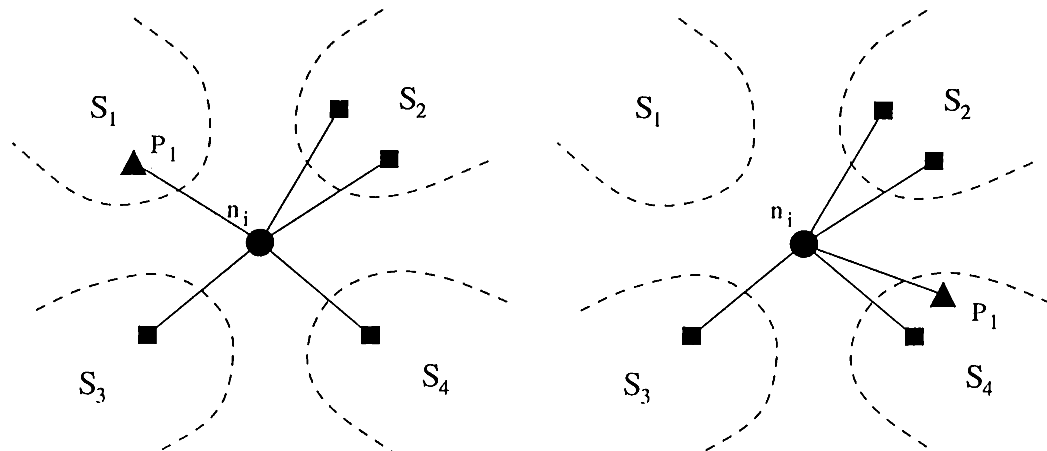


Figure 5.3. Special vertices are introduced into the hypergraph in one-phase model.

which is correct. However, if C_i was kept in the memory of P_1, then the correct cost added to the cut should have been $3c_i$. This is because none of the processors had the necessary cluster, and C_i should have been sent from P_1 to all other processors. Two-phase model simply ignores this fact, and in many cases calculates an incorrect $\chi(\Pi)$.

In general, two-phase approaches try to solve the assignment problem independent of the partitioning problem. Solution space provided by partitioning phase to the assignment phase is rather restricted. Hence, it may not always be possible to find a high quality mapping in two-phase models.

To convert the two-phase model into a one-phase model, it is necessary to supply the initial cluster-processor mapping to the model. In one-phase model we used a hypergraph $\mathcal{H}' = (\mathcal{V}', \mathcal{N})$ with some vertices and pins added to the previous hypergraph $\mathcal{H}$. Vertex set of new hypergraph, $\mathcal{V}'$ is formed by adding K special vertices, p_i, to the original vertex set, $\mathcal{V}$. These special vertices represent the K processors used, and they are assigned no weight. Also some new pins are added to the pins in the original hypergraph. We use a new pin between a special vertex and a net, if the cluster that the net is representing resides in the memory of the processor that the special vertex represents. The example in Figure 5.2 is redisplayed for this new model, and two different initial cluster-processor assignments are shown, in Figure 5.3. The triangular node in the figure represents the special vertex used for P_1. The other three special

vertices are not displayed for clarity.

This model is able to calculate the cost correctly for all possible partitionings. After introducing special vertices, it is necessary to add a constraint such that each special vertex is assigned to only one part. This constraint allows the model to find both the partitions and region-processor mapping in just one phase.

In order to obtain this model, some modifications are performed over the PaToH hypergraph partitioning tool. Throughout the partitioning process, special vertices are treated differently than the others. In coarsening phase, matching of two special vertices is prevented. They were able to be involved in a supernode with ordinary vertices, but they are never grouped with another special vertex in the same supernode. A supernode containing a special vertex gained the special vertex status. During initial partitioning phase, each special vertex is assigned to a part, and locked there. As uncoarsening and refinement phases progress, these special vertices were not able to move to other parts.

After partitioning the hypergraph $\mathcal{H}'$, a partition Π' is found. Using this partition, mapping function $\mathcal{M}$ is calculated according to Equation 5.2:

$$\mathcal{M}(\mathcal{S}_i) = P_j \Leftrightarrow p_j \in \mathcal{S}_i \tag{5.2}$$

At the end, Equation 5.1 is used to determine the communication pattern for sending data clusters between the processors, that is, $\check{\mathcal{M}}$ is calculated.

5.2 Data Distribution

Initial data distribution is an important factor, affecting the quality of the remapping process. Current cluster-processor mapping determines the topology of the hypergraph used during remapping. For different cluster-processor assignment schemes, this results in different partitionings, and hence variations on the data communication cost. Moreover, cluster distribution can affect the amount of view-dependent preprocessing overhead. This is because the time spent on work load calculations is different for each cluster.

After remapping, within the succeeding visualization instance, both the number of clusters in each processor and clusters' preprocessing costs change. Since the load balance constraint in remapping is only applied on the distribution of the screen load, there may be huge variations in view dependent preprocessing costs, making it impractical to use. As a result, it seems very difficult to include an adaptive screen subdivision scheme, and an adaptive data distribution scheme together, in image-space parallelization.

In this work, two different data distribution schemes were tested: Neighbor Cluster Assignment (NCA) scheme, and Scattered Cluster Assignment (SCA) scheme. Both of these schemes are static data distribution schemes, that is, data is distributed at the program startup. Although some clusters can be replicated in different processors, its first owner never changes.

- *NCA scheme*: In this scheme, all processors are given a set of neighbor clusters. The assignment of clusters to processors are static; that is, in each remapping step, this initial data distribution is used as the starting data-processor assignment. This scheme makes use of object-space coherency well. Also, two neighbor clusters have a higher chance of having a projection area on the same screen cells. Consequently, a better hypergraph topology can be constructed, helping the minimization of the communication cost during hypergraph partitioning. If the successive visualization instances contain many 1° rotation operations, NCA appears to be the best possible static data distribution scheme.

- *SCA scheme*: Generally, neighbor clusters contain cells with similar size

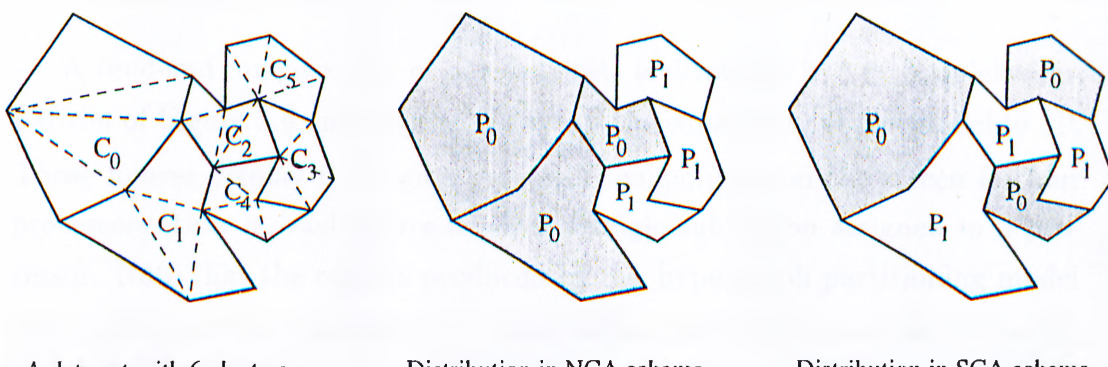


Figure 5.4. Cluster distribution schemes.

and shape. Hence, in NCA scheme, each processor carries a set of clusters with nearly equal preprocessing costs. This creates a load imbalance in preprocessing step, increasing the execution time of this scheme. SCA scheme, on the other hand, tries to distribute the clusters as scattered as possible. Since processors had cells of varying sizes, much less preprocessing overhead is observed in this scheme. In case the visualization contains operations such as zooming to a particular region on the screen, this scheme is the most beneficial static data distribution scheme. Its only disadvantage is observed at the slight increase in the communication amount, due to the usage of a topologically more complicated hypergraph during the remapping phase.

Figure 5.4 displays these distribution schemes on an example, for 2 processor case, using a simplistic data set containing six clusters. Note the load imbalance in the NCA scheme due to the big variation in the tetrahedral cell sizes. In that figure, although both P_1, and P_2 have 3 clusters, and 12 cells, the preprocessing performed by P_1 is much greater than the one performed by P_2.

5.3 Hypergraph versus Jagged Partitioning

Jagged partitioning (JP) is a two-phase partitioning model. For a $K = pq$ processor decomposition, this method first divides the screen into p stripes in one dimension. Then, each stripe is further divided into q regions, in the unused dimension, independent of each other (see. Figure 3.4). The details of JP can be found in [28].

A rendered image of one of our data sets is displayed in 5.5. A quick comparison of Figure 5.6 and Figure 5.7 reveals the superiority of our model to JP. Those figures display the region-processor assignments on the screen for four processors, that is, each figure denotes a single sub-region assigned to a processor. Note that the regions produced by our hypergraph partitioning model have much more flexible sub-screen boundaries than the the ones in JP model. Also, the assigned regions are not necessarily made up of a single connected component as in JP. In our particular example, some tiny regions are assigned

to each processor separately from a larger region. This is probably, because the clusters above those regions are owned by these processors. Especially, in SCA scheme, this allows a greater flexibility for remapping.

Moreover, JP model carries all characteristics of a two-phase model. As in our two-phase hypergraph model, bipartite graph matching is necessary in the second phase, to find the region-processor mappings so that better cluster-processor mappings will be obtained. In other words, JP, by itself, is not capable of directly minimizing the redistribution communication costs of the clusters. Without the second phase, it primarily attacks the load balancing problem. Although, the solution found by the second phase is an optimal solution, it may not be that good, since the problem given to that phase has a rather restricted solution space.

One-phase hypergraph model, on the other hand, both balances the load and minimizes the migration costs at one single step. The experiments we conducted verifies the validity of our model. At similar load balance values, the communication incurred in the one-phase hypergraph model is 25% less than the one in JP model, on the average.

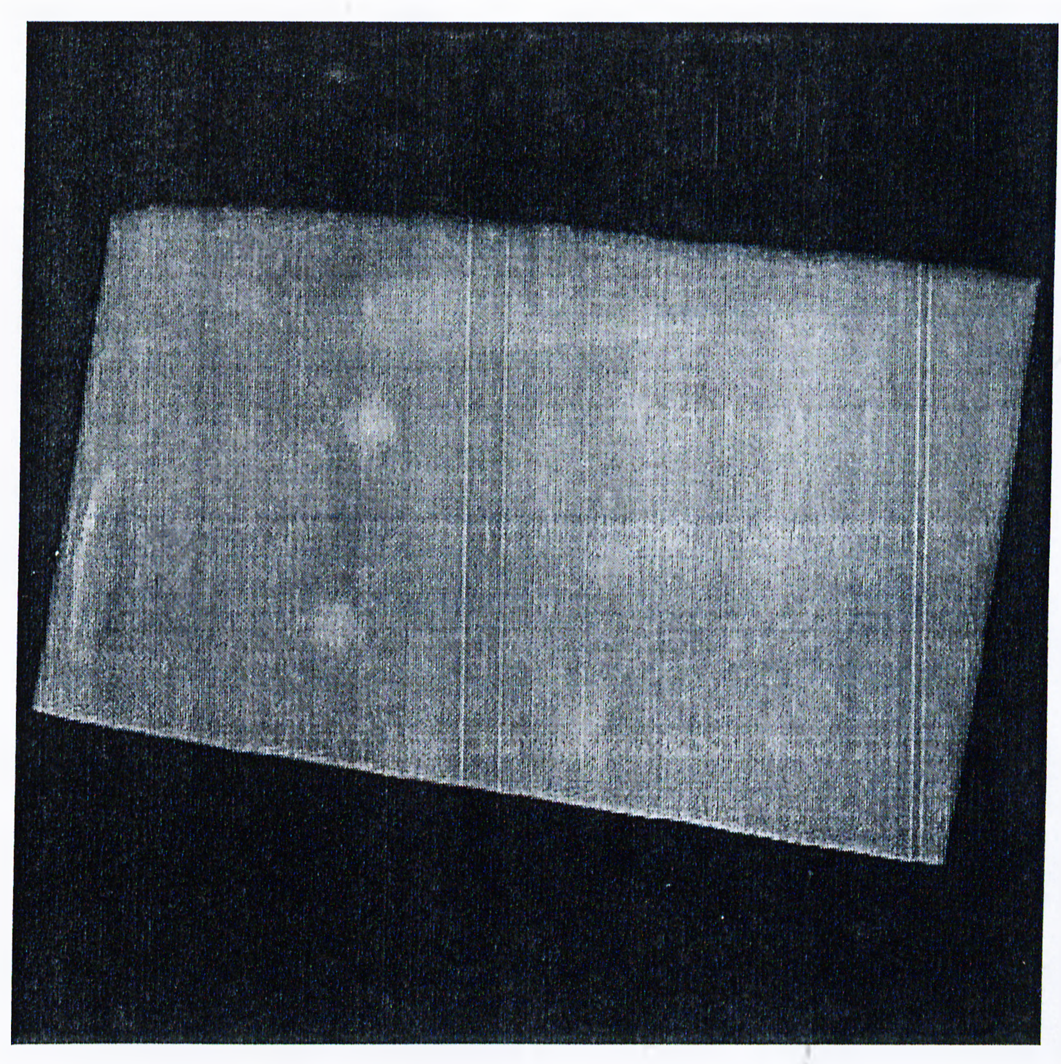


Figure 5.5. Rendered image of CC data set.

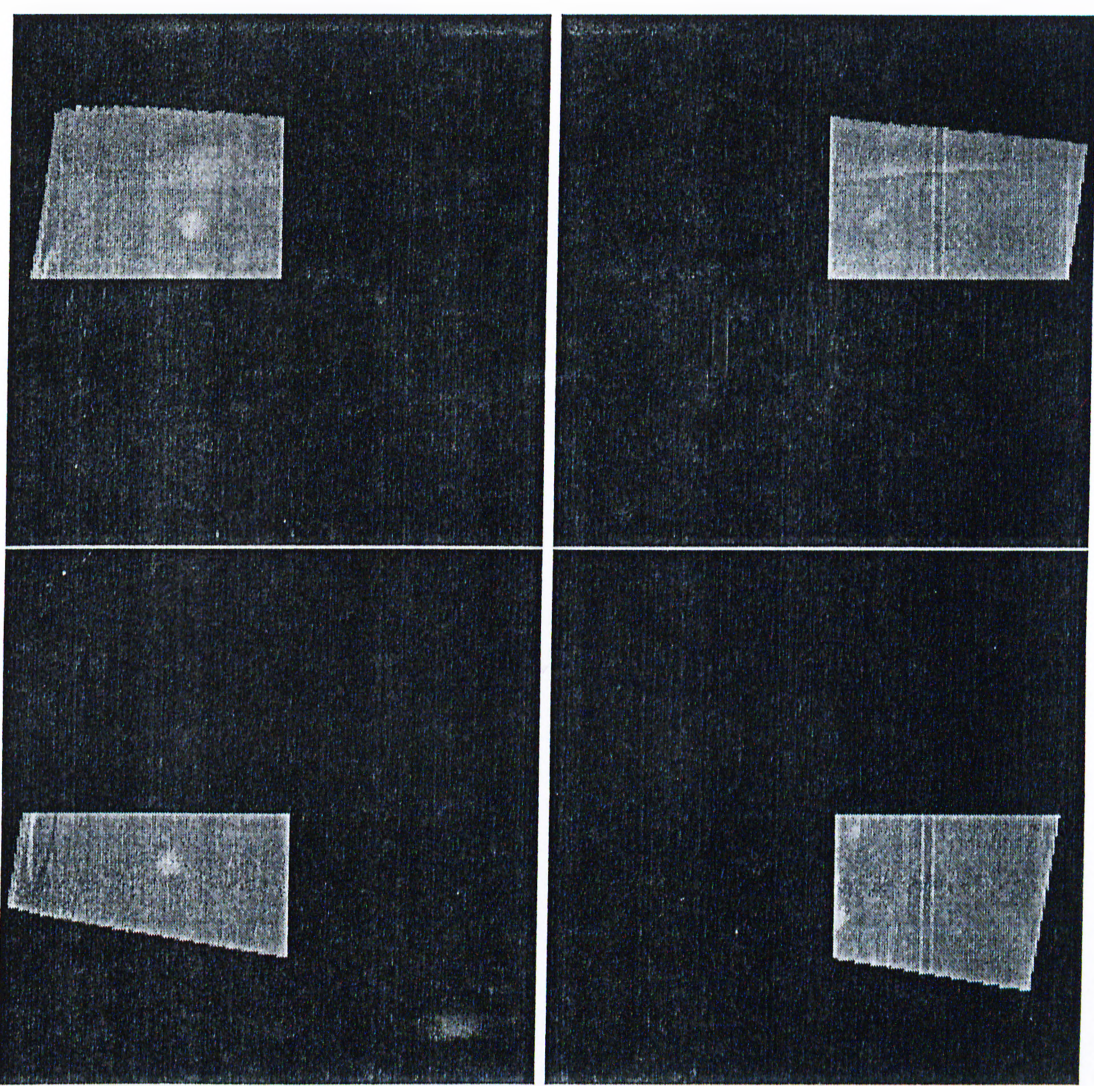


Figure 5.6. Example region-processor assignment in jagged partitioning.

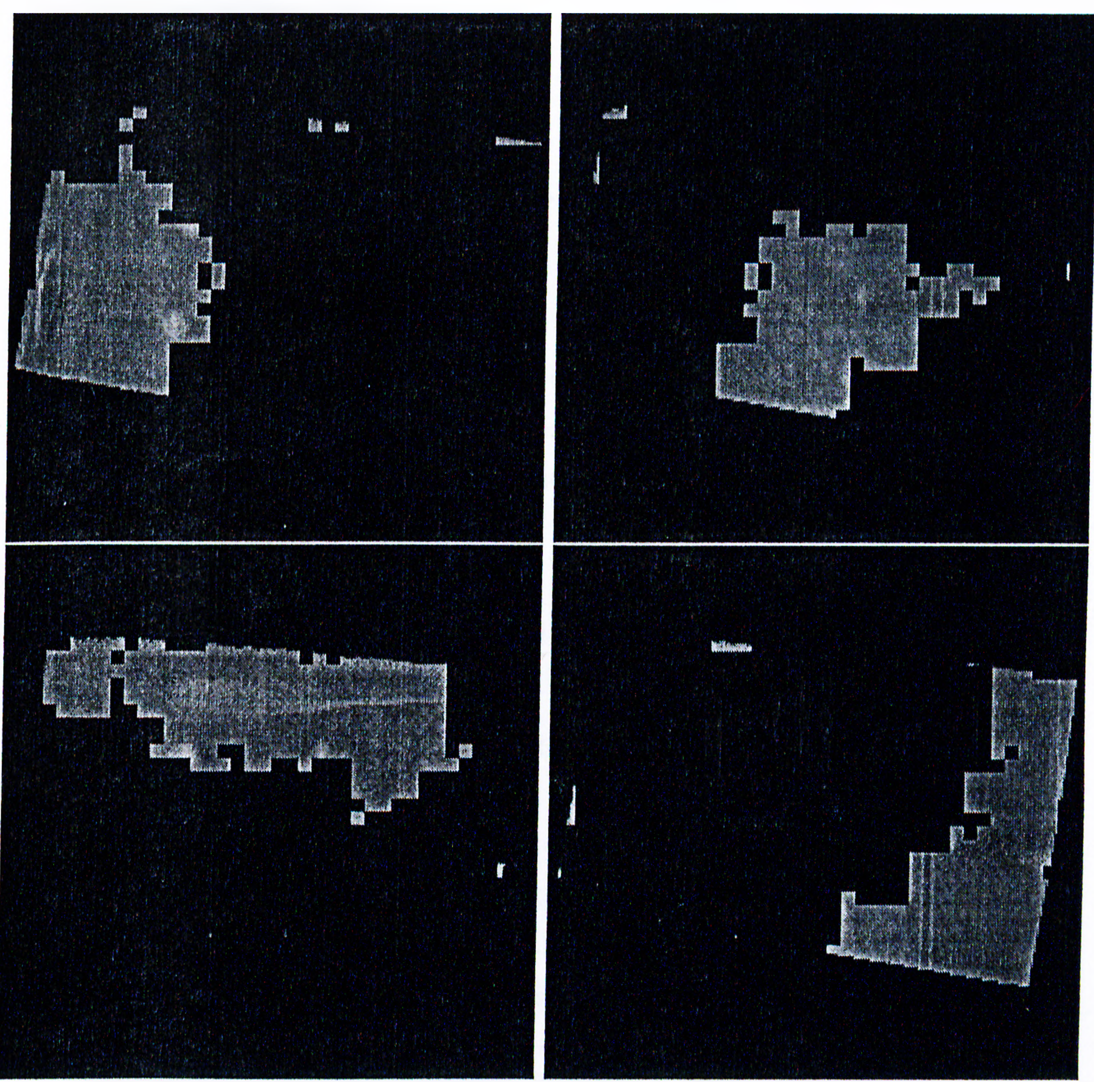


Figure 5.7. Example region-processor assignment in hypergraph partitioning.

Chapter 6

Implementation Details

This chapter explains some details in our implementation which is mainly composed of four consecutive phases: View independent preprocessing, view dependent preprocessing, cluster migration, and local rendering.

6.1 View Independent Preprocessing

As the name suggests, this step performs all the work that does not depend on the visualization parameters, and it is executed just once at program startup. Our implementation assumes that the 3D volume data is stored in several chunks, and the number of data chunks is equal to the processors used in that particular run of the program. Since the data we used is usually produced by simulations performed on parallel machines, this assumption seems reasonable.

At the program startup, a master processors reads the whole data set from disk and sends a single data chunk to each processor. Each chunk contains equal amount of tetrahedral cell information, and the data in each processor is treated as processors' local data. Every processor concurrently constructs the appropriate data structures mentioned in Chapter 2. For some external faces which are shared by two cells stored in different processors, communication is carried out among the processors in order to complete the connectivity information within the tetrahedral cells.

Then, using connectivity data structures a local clusterization graph is created in each processor. These graphs are given as input into METIS graph partitioning tool, and as output, the partitions showing the appropriate clusterization of the data are obtained. Later, processors fill the additional data structures used to store the clusterization information, by inspecting the returned partitions.

Since the Greedy Graph Growing algorithm is used within METIS, in most cases, clusters produced are made up of connected cells. However, there may be a few disconnected clusters. Especially, for work load assignment schemes using bounding boxes, these disconnected clusters may cause wrong load distributions due to the increase in the bounding boxes calculated. Moreover, they may cause an increase in the number of pins of the remapping hypergraph, and hence, an increase in the duration of the hypergraph partitioning step. As a result, we prefer to eliminate the existing disconnected clusters at the cost of some additional view independent preprocessing overhead. For this purpose, we perform breadth first search on every cluster to see whether they are connected, and if a disconnected cluster is found it is separated into its connected components. All the smaller components found in a processor are placed in a single, newly created cluster. This limits the error made in work load assignment step to a single cluster.

For NCA scheme, there is no need to a change on the current cluster-processor assignments, since the data chunks read for each processor from the disk is already made up of neighbor clusters. For SCA scheme we need to scatter the clusters among processors. To do that we follow a method similar to bin packing. First, we calculate the sum of face areas for every cluster, and enter the cluster numbers into a list in decreasing order of their area sums. Then, starting from the top of the list, we begin to distribute the clusters. Mapping decisions are made such that a cluster is given to the less heavily loaded processor in terms of face area sum of its stored clusters, at any time. Moreover, during this mapping a limit is applied on the number of clusters a processor can hold. Hence the number of cells in processors are also kept in balance.

6.2 View Dependent Preprocessing

In every visualization instance, this preprocessing phase is repeated for the sake of the efficiency of code sections running in parallel. Hence, it is important to complete this phase as quick as possible. It mainly contains work load calculations, hypergraph creation, and hypergraph partitioning steps.

6.2.1 Work Load Calculation

Since we used an adaptive screen sub-division technique and imposed a grid on the actual screen, it is very important to adjust the granularity of this grid correctly. As a too fine-grain grid may create lots of vertices in the remapping hypergraph, a too coarse-grain grid may result in a hypergraph with a few vertices. The first case results in a rather complicated hypergraph, which may decrease the efficiency of partitioning heuristics used in PaToH. The later case restricts the solution space of the problem, and proper load balance values cannot be obtained.

The granularity of the grid depends on the number of pixels occupied by the projection area of the volume, and the total screen cell count that we want to be produced. By screen cell we mean the screen cells with some associated rendering load. In our implementation we used numbers between 400 and 500 for the total screen cell count. These upper and lower bounds are found empirically. There are $g \times g$ pixels in a single screen cell, and we calculate g using the formula in Equation 6.1, where A, and C denote the projection area of the volume on the screen, and the total number of screen cells, respectively (see Appendix A).

$$g = \frac{\sqrt{A} - 2 + \sqrt{(C-1)A - 2C\sqrt{A} + 2C}}{C - 2} \qquad (6.1)$$

After the grid is imposed on the screen, we use one of the three methods mentioned in Chapter 3 to calculate the load distribution on the screen.

6.2.2 Local Hypergraph Creation

While the load distribution is calculated, the interaction between the data and the screen is also recorded. Some data structures are used to keep track of the connectivity between the clusters and the screen cells. In a sense, each processor constructs a hypergraph in an intermediate data structure. Since each processor has a subset of the whole volume, only the local hypergraphs can be constructed.

6.2.3 Global Hypergraph Creation

After local hypergraph creation, each processor sends its hypergraph to all other processors, that is, an all to all broadcast operation is performed among the processors in order to obtain the information necessary to construct a common global hypergraph. During this construction process, screen cells with no rendering load are discarded. This is because, vertices of the global hypergraph should be chosen from the loaded screen cells. Special processor vertices, necessary for the one-phase remapping model are also added in this step.

6.2.4 Hypergraph Partitioning

In this phase, each processor executes PaToH hypergraph partitioning tool sequentially, to partition the global hypergraph obtained in the previous step. In two-phase model, the resulting partition is given as input to our graph bipartitioning code, and there, the cluster-processor mappings are calculated. In one-phase model, this mapping is directly calculated by checking the special vertex locations in the partition, which was returned by PaToH.

6.3 Cluster Migration

The clusters, which are mapped to a different processor than their current processor, are send to the processors they are assigned to. This is done by

performing an all to all personalized communication among the processors, and replicating the data structures of the migrating clusters.

6.4 Local Rendering

After remapping, all processors receive the clusters necessary to render their assigned screen regions. Since the current cluster-processor assignment in a processor results in a highly interacting set of neighbor clusters, clusters do not need to be rendered individually. Instead, ray entry points of the clusters which are closest to the screen are determined, and the rays shot from these points are followed throughout the volume. The ray entry points can be found by scan converting the set of F_{eff} faces stored in a processor. A ray segment is generated for each pixel found during the scan conversion, excluding the pixels falling out of the processors' screen regions.

Even if the cells traversed by a ray belong to different local clusters, we can efficiently traverse the volume as in Koyamada's original algorithm, by utilizing the additional data structures which store the connectivity information between the clusters. Also, note that, although clusterization process can create non-convexities within some clusters, this will not cause an increase in the number of ray-segments generated. This is because a processor had all the necessary clusters and traverses them as if traversing a single convex sub-volume.

For non-convex volumes, on the other hand, there is a possibility of having more than one ray-segment for the same pixel, which necessitates the use of ray-buffers we mentioned in Chapter 2. In such volumes, the ray-segments generated for a screen pixel are inserted into the appropriate slots in ray-buffers, in sorted order of their their exit z coordinate values, including the color and opacity values associated with the ray-segments. When all ray-segments are traversed, the color and opacity values are retrieved from the ray-buffers and are composited using the standard composition formulas. At this point, since processors have created just a sub-image of the assigned screen regions, a final all-to-one communication step is carried out, and the full image is generated in a single processor.

Chapter 7

Experimental Results

This chapter presents the results obtained from our experiments on three different data sets, using various parameters for visualization, partitioning and parallelization.

7.1 Implementation Platform and Data Sets Used

The work done in this thesis is implemented on a Parsytec's CC-24 system. This machine is based on a distributed memory, MIMD, architecture. It has 24 nodes, each of which is containing a 133 MHz PowerPC 604 processor. The machine has 4 I/O nodes with 128 MB of RAM, and 20 compute nodes with 64 of MBytes of RAM. The two of the I/O nodes are also used as the entry nodes to the system. All nodes are connected to each other via a high speed communication link. The peak performance of this link is 40 MB/s [30].

AIX is the operating system used on each node of the Parsytec CC system. On top of it, Embedded Parix (EPX) is used. It provides a set of functions for management of the communication between the nodes [49]. For message passing purposes, we made use of EPX and Parallel Virtual Machine (PVM) libraries. The algorithms were coded in C programming language.

Table 7.1. Some features of the data sets used.

Data Set	# of Nodes	# of Cells	CSV
Blunt Fin	40,960	187,395	5.50
Combustion Chamber	47,025	215,040	0.42
Oxygen Post	109,744	513,375	4.26

As experimental volume data, we used three different data sets [50]: Blunt Fin (BF), Combustion Chamber (CC), and Oxygen Post (OP). These data sets are the results of some computational fluid dynamic simulations. They were originally curvilinear in structure. We converted the format of these data sets into unstructured data format, by applying a tetrahedralization [11, 40] process on the hexahedral cells of the original data, obtaining five tetrahedral cells per hexahedral cell.

Table 7.1 illustrates some features of the data sets used. In that table, the number of nodes and the number of cells in the data sets are given. Also, a CSV value is displayed per data set to represent the cell size variation within the cells of a data set. Higher CSV values imply a more irregular data set. Note that, the BF and OP data sets we used are rather irregular data sets.

The experiments are made using a wide range of changing parameters. Image screens of size 400×400, 800×800, and 1200×1200 are used in final images. In all visualizations, we leaved a thin margin between the final image and the screen boundaries. Images are tried to be fit into the screen as much as possible. For such details of the sequential visualization algorithm refer to [3]. The abbreviations used in the tables of this chapter are listed in Table 7.2. All timings are in seconds, and communication volume is given in KBytes. Load imbalance values are measured as the ratio of the maximum number of samplings done by a processor to the average sampling count. Load imbalance values found in terms of local rendering times give very similar results. By communication volume, we only mean the communication performed in the cluster migration step, and used this words to mean that, throughout this chapter.

Table 7.2. Abbreviations used in tables.

K	number of processors used
c	number of cells in the remapping hypergraph
p	number of pins in the remapping hypergraph
W_V	vertex weighting scheme
W_E	edge weighting scheme
LI	load imbalance in the number of samplings per processor
TVoC	total volume of communication performed in cluster migration
T_{par}	parallel rendering time
T_{pre}	view dependent preprocessing time
T_{wlc}	work load calculation time
T_{hf}	hypergraph formation time
T_{hp}	hypergraph partitioning time
T_{cm}	cluster migration time
T_{lr}	local rendering time
T_{seq}	sequential rendering time
S	speedup
E	efficiency

7.2 View Independent Preprocessing

The duration of processors' view dependent preprocessing phase, and the load imbalance in local rendering phase can be affected by the total number of clusters used. Table 7.3 displays the results obtained by creating different number of clusters ($C \in \{50, 100, 200\}$) per processor ($K \in \{4, 8, 12, 16, 20, 24\}$), initially. Results are obtained by executing our implementation on all data sets over a screen of size 400×400 pixels. The code is executed 12 times per (C, K) pair, with three different view points, and four different random seed values. The random seed used here and in the other runs is necessary to obtain more accurate average values, since the partitioning heuristics in METIS and PaToH both make use of some randomness at several places where a selection is necessary, and hence may produce partitions of differing quality over a wide range.

Data in Table 7.3 were calculated by averaging the results found. For easier comparison, T_{pre} values are normalized with respect to the lowest value in the

Table 7.3. Results obtained by assigning different cluster counts per processor.

K	# of clusters per processor					
	50		100		200	
	LI	T_p	LI	T_p	LI	T_p
4	3.367	1.742	1.940	1.947	1.345	2.360
8	4.589	1.158	3.689	1.327	3.201	1.567
12	5.891	1.027	4.258	1.271	4.299	1.417
16	6.255	1.009	5.291	1.243	5.260	1.404
20	8.738	1.005	7.938	1.150	7.203	1.395
24	10.220	1.000	9.590	1.144	9.116	1.391

table. It can be seen from the table that rising C values increase the preprocessing overhead. Hence, too high values cannot be used for C. Otherwise, we may observe severe decreases in our speedup values. On the other hand, using too low C values may increase the load imbalance among the processors. This is because, the lower C values cause larger cluster volumes, and hence the error made during the work load assignment phase is higher. As a result, we preferred using an average number of 100 clusters per processor, which has reasonable load imbalance and T_{pre} values. All the experiments following this are carried out with this fixed cluster per processor value.

Table 7.4 displays the effects of the various weighting schemes that can be used as edge and vertex weights on the clusterization graph. The experiments are performed over all data sets with a 400×400 screen, using 8 processors. Number of pins in the hypergraph, cut of the partition and the view dependent preprocessing time can be seen from the columns of the table, separately for

Table 7.4. Effects of all possible weighting schemes used in the clusterization graph.

W_V	W_E	BF			CC			OP		
		p	TVoC	T_{pre}	p	TVoC	T_{pre}	p	TVoC	T_{pre}
1	1	9,609	17,261	1.216	9,083	21,022	1.260	7,939	42,102	1.855
1	FA	6,929	15,969	1.070	8,278	20,775	1.196	5,279	38,244	1.604
CA	1	11,827	17,382	1.473	9,079	20,789	1.254	10,213	43,484	2.197
CA	FA	8,653	16,344	1.171	8,273	20,363	1.198	6,651	39,547	1.858
CV	1	11,526	17,131	1.363	9,099	20,601	1.248	9,166	42,254	2.046
CV	FA	8,120	16,037	1.126	8,290	20,907	1.977	6,228	38,314	1.731

Table 7.5. Effects of different work load calculation schemes.

	c	p	LI	TVoC	T_{wlc}	T_{hp}
I. O. Test	342	7,837	6.181	45,036	0.615	0.345
Cell B. B.	354	8,648	8.353	47,888	0.410	0.376
Cluster B. B	384	10,323	10.150	51,339	0.256	0.448
Exact	342	7,837	5.521	44,969	2.612	0.340

each data set.

We note that, for weighting schemes which assign unit cost to the edge weights, a more complicated hypergraph is created, that is, the number of pins in the remapping hypergraph is higher than the ones in schemes which consider the face areas for edge weighting. This results in heavier final partition cuts and also increases the duration of hypergraph partitioning step, meaning more preprocessing overhead. As seen in the table, schemes which assign FA as the edge weight can have almost 30% less pins in the remapping hypergraph than the unit cost schemes.

Among the remaining three schemes, (1,FA) scheme is seen to be better than the others. Especially for BF and OP data sets, which have big cell size variations, it both results in better partitions of the remapping hypergraph, and is faster. These observations validates our choice of using (1,FA) weighting scheme in the clusterization graph.

7.3 View Dependent Preprocessing

In this section, first, we compared four different work load calculation schemes. Three of these schemes are cluster bounding box, cell bounding box and inside-outside test schemes that we used in our implementation. The fourth one, that is, exact load scheme is capable of calculating the exact load distribution on the screen by scan converting every face in the data. In other words, it uses individual cells instead of clusters during preprocessing phase. We included it here just for comparison purposes, since its execution time is not affordable for a parallel application.

Table 7.6. Imbalance values and communication amounts observed.

		400 × 400		800 × 800	
		LI	TVoC	LI	TVoC
BF	4	3.427	13506	1.646	12947
	12	6.205	21337	7.626	21606
	20	12.825	24811	9.904	25370
CC	4	1.005	17002	1.383	17141
	12	5.441	28314	5.739	29562
	20	11.447	35142	11.678	34568
OP	4	2.347	30443	2.223	30088
	12	5.480	43196	5.794	44468
	20	11.556	48212	10.191	49876

Columns of Table 7.5, from left to right, contain the average values found, that is, cell and pin numbers in the remapping hypergraph, load imbalance, total volume of communication, work load calculation time, and hypergraph partitioning time. Runs are made on 16 processors, using OP data over a 400×400 image screen, with SCA distribution scheme. In order to see the effects of work load calculation schemes on the topology of the remapping hypergraph, we used a fixed screen granularity of 20 pixels per screen cell.

The results verifies the importance of correctly establishing the remapping hypergraph topology. Notice the excess cells and pins introduced in the bounding box approximations. When these schemes are used, those additional screen cells cause extra overhead in the total volume of communication, and cause an increase in the hypergraph partitioning time due to the calculations performed for these miscalculated cells and pins. Also, the difference in load imbalance values produced by the exact scheme and inside outside test scheme should be noted. Exact scheme seems to be better at calculating the work load. This is basically because, in inside outside test scheme, each pixel under a clusters projection area is assigned the same cost, although the clusters may have unbalanced cell distributions along the viewing direction. On the other hand, inside outside test runs approximately 4 times faster than the exact scheme, and produces nearly the same timing result in hypergraph partitioning. Moreover, it provides a pretty good minimization of the total communication volume.

Table 7.6 displays the load imbalance values and total volume of communication amount observed in our model for 400×400 and 800×800 screen sizes. As the number of processors increase, we note higher values in the table

Table 7.7. Dissection of view dependent preprocessing time.

	400 × 400				800 × 800				1200 × 1200			
	T_{wlc}	T_{hf}	T_{hp}	T_{cm}	T_{wlc}	T_{hf}	T_{hp}	T_{cm}	T_{wlc}	T_{hf}	T_{hp}	T_{cm}
BF	1.527	0.354	0.192	0.815	4.048	1.284	0.189	0.777	8.022	2.731	0.187	0.765
	0.947	0.327	0.348	0.743	2.573	1.080	0.366	0.718	5.172	2.297	0.346	0.686
	0.738	0.270	0.493	0.688	2.032	0.769	0.508	0.674	4.099	1.590	0.504	0.630
	0.625	0.273	0.639	0.686	1.720	0.708	0.669	0.633	3.468	1.434	0.645	0.609
	0.560	0.272	0.767	0.697	1.514	0.651	0.785	0.629	3.064	1.249	0.758	0.614
	0.516	0.294	0.876	0.646	1.376	0.756	0.901	0.619	2.784	1.278	0.869	0.620
CC	1.476	0.297	0.238	0.787	3.631	1.021	0.209	0.753	7.086	2.209	0.223	0.752
	0.879	0.232	0.454	0.635	1.757	0.554	0.596	0.593	4.399	1.456	0.415	0.606
	0.677	0.226	0.628	0.620	1.493	0.513	0.787	0.531	3.470	1.183	0.586	0.590
	0.570	0.227	0.788	0.592	1.493	0.513	0.787	0.531	2.980	1.035	0.775	0.532
	0.518	0.240	0.890	0.555	1.345	0.494	0.961	0.495	2.665	0.911	0.916	0.514
	0.477	0.315	1.033	0.519	1.232	0.490	1.138	0.489	2.450	0.915	1.156	0.449
OP	2.052	0.213	0.106	1.972	3.798	0.706	0.099	2.123	6.575	1.649	0.106	2.022
	1.196	0.175	0.198	1.839	2.340	0.518	0.188	1.911	4.107	1.114	0.194	1.881
	0.903	0.187	0.291	1.773	1.872	0.905	0.288	1.925	3.232	1.157	0.290	1.804
	0.742	0.193	0.389	1.703	1.506	0.448	0.378	1.838	2.742	1.061	0.383	1.775
	0.645	0.206	0.462	1.656	1.320	0.442	0.450	1.819	2.432	1.000	0.469	1.779
	0.588	0.226	0.545	1.582	1.201	0.489	0.543	1.747	2.212	0.990	0.558	1.759

as expected. On the other hand, increasing image sizes do not affect LI and TVoC values much. This is because we keep the size of our hypergraph at the same level by decreasing the granularity of the grid imposed on the screen when the screen sizes got bigger. Hence the problem size remains the same for different screen sizes.

7.4 Performance

In this section we analyze the execution times and performance of our algorithm. Table 7.7 presents dissection of T_{pre} into T_{wlc}, T_{hf}, T_{hp}, and T_{cm}. Note that, since the cluster migration step does not exist in sequential code, we consider T_{cm} as a preprocessing cost here. We note that T_{wlc} decreases as the number of processors increase. This is because work load calculation step is carried out by all processor on a part of the data in parallel. That is when the processor number increases the number of faces scan converted by a processor decreases. In BF data set, there occurs an interesting increase in T_{hf} for $K = 24$. The reason for this increase is the communication overhead during the construction of the the global remapping hypergraph. Also, a noticeable increase occurs in the duration of the hypergraph partitioning phase. The additional nodes in the one-phase model's hypergraph cause this increase.

Table 7.8. Speedup and efficiency values for different data sets and processor numbers.

		400 × 400				800 × 800				1200 × 1200			
	K	T_{seq}	T_{par}	S	E	T_{seq}	T_{par}	S	E	T_{seq}	T_{par}	S	E
BF	4	109.43	28.95	3.78	94.5	430.54	112.54	3.82	95.5	980.24	254.22	3.85	96.2
	8		16.32	6.70	83.7		58.24	7.39	92.3		132.01	7.43	92.8
	12		11.82	9.26	77.2		39.92	10.78	89.8		88.99	11.01	91.7
	16		9.80	11.17	69.8		31.67	13.59	84.9		70.92	13.82	86.3
	20		7.89	13.86	65.2		27.19	15.83	79.1		60.60	16.17	80.8
	24		7.64	14.43	60.1		25.12	17.14	71.4		53.15	18.44	76.8
CC	4	124.68	32.23	3.85	96.3	497.79	126.57	3.93	98.5	1120.41	281.67	3.97	99.2
	8		17.73	7.03	87.8		65.82	7.56	94.5		145.73	7.69	96.1
	12		12.72	9.80	81.6		45.63	10.91	90.8		100.39	11.16	93.0
	16		10.26	12.14	75.8		35.59	13.98	87.3		77.86	14.39	89.9
	20		8.80	14.16	70.8		29.66	16.78	83.9		64.31	17.42	87.1
	24		7.95	15.68	65.3		25.81	19.28	80.3		55.40	20.22	84.2
OP	4	170.57	44.52	3.83	95.7	653.77	166.56	3.92	98.0	1579.97	398.645	3.96	99.0
	8		24.06	7.08	88.5		86.84	7.52	94.0		204.15	7.73	96.7
	12		16.91	10.08	84.0		61.47	10.63	88.5		141.35	11.17	93.0
	16		13.47	12.66	79.1		47.89	13.64	85.2		113.21	13.95	87.2
	20		11.84	14.40	72.0		39.87	16.39	81.9		92.58	17.06	85.3
	24		10.39	16.40	68.3		35.62	18.35	76.4		80.89	19.53	81.3

Moreover, since this part is run sequentially, it prevents our parallelization from having higher speedup values.

Table 7.8 displays the speedup and efficiency values obtained for $K \in \{4, 8, 12, 16, 20, 24\}$, for all image sizes and data sets. Obviously, as the screen sizes increase all speedups and efficiencies in the table also increase. This is because sequential parts of the code constitute a lesser portion of the total work for large image sizes. For 400×400 image size, low efficiency values are observed due to the sequentially running hypergraph partitioning code. For very small image sizes, T_{hp} can even approach T_{lr} resulting in rather poor speedup and efficiency values.

7.5 Comparison with Jagged Partitioning

Figures 7.1, 7.3 and 7.2 gives a brief comparison of our one-phase hypergraph partitioning model (HP)with jagged partitioning model (JP) model. The data were collected using the OP data set over a screen of size 1200×1200. First graph displays a comparison of load imbalance values in these models for different number of processors. For low processor numbers, HP has a good load imbalance. On the other hand, as the number of processors increase, it quickly begins to produce unbalanced partitions. JP has a lower imbalance increase

rate for high number of processors.

As we can see from Figure 7.2, HP incurs slightly less preprocessing overhead than JP. The major difference is seen in the total volume of communication performed in both schemes. HP is much better at minimizing the communication volume. Especially, for increasing number of processors this becomes more apparent. At 24 processors, HP performs approximately 30% less communication than JP.

In general, at large image and data sizes HP outperforms JP, both in terms of speed and the minimization of communication volume. For small scale data sets and image sizes, both models are preferable. In such problems, HP produces 10% less communication overhead on the average at comparable load imbalance values.

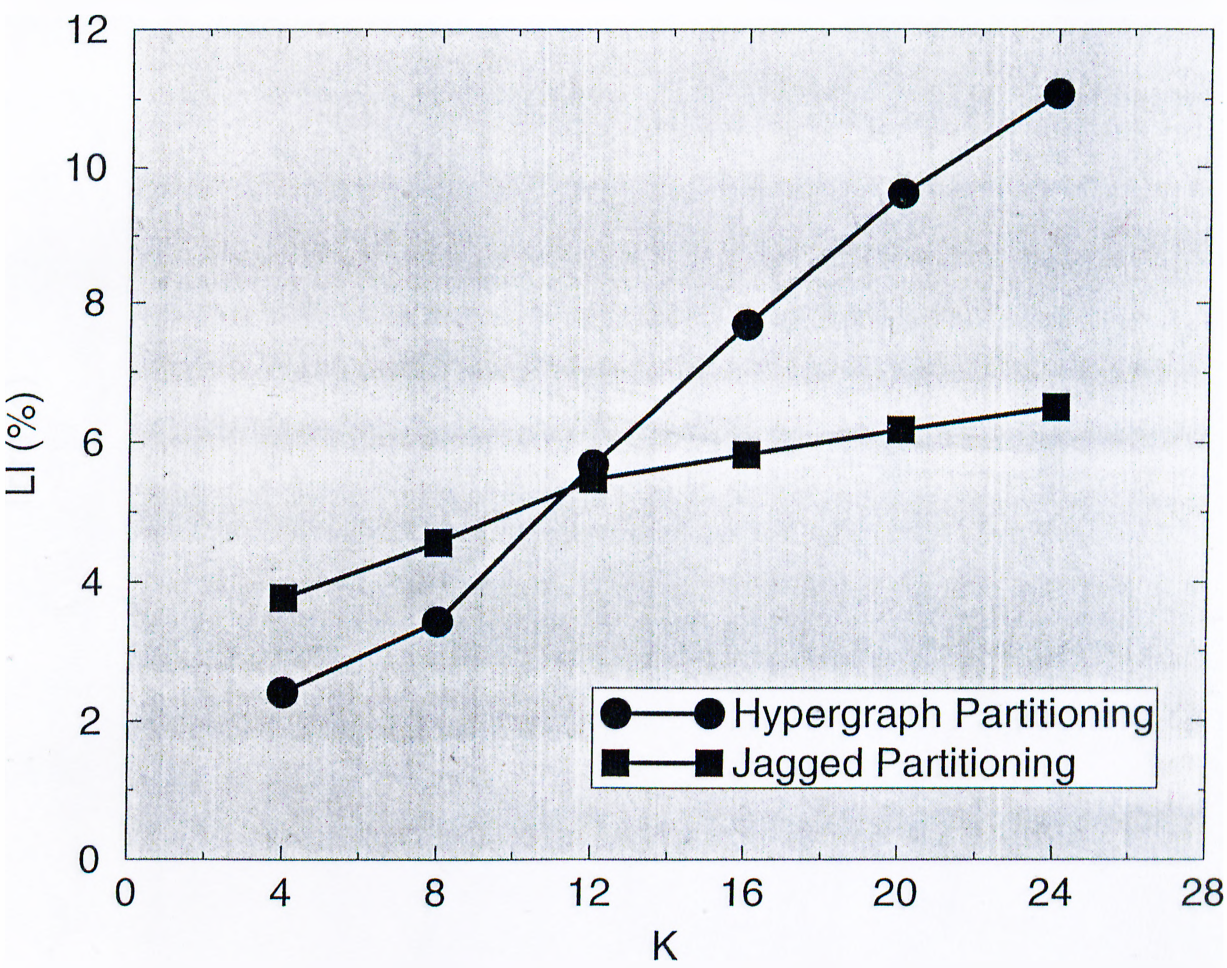


Figure 7.1. Load imbalances in HP and JP.

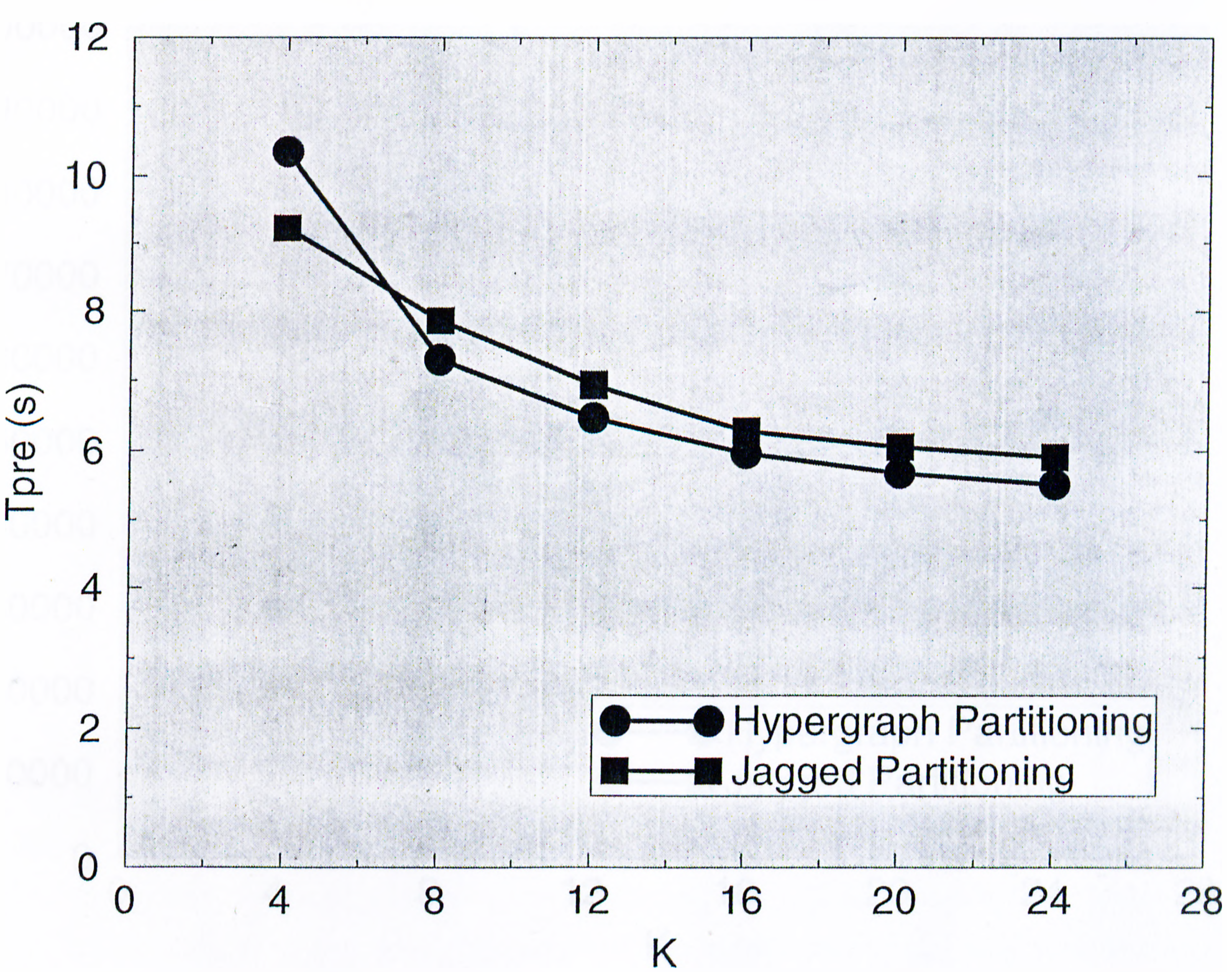


Figure 7.2. Preprocessing overhead incurred in HP and JP.

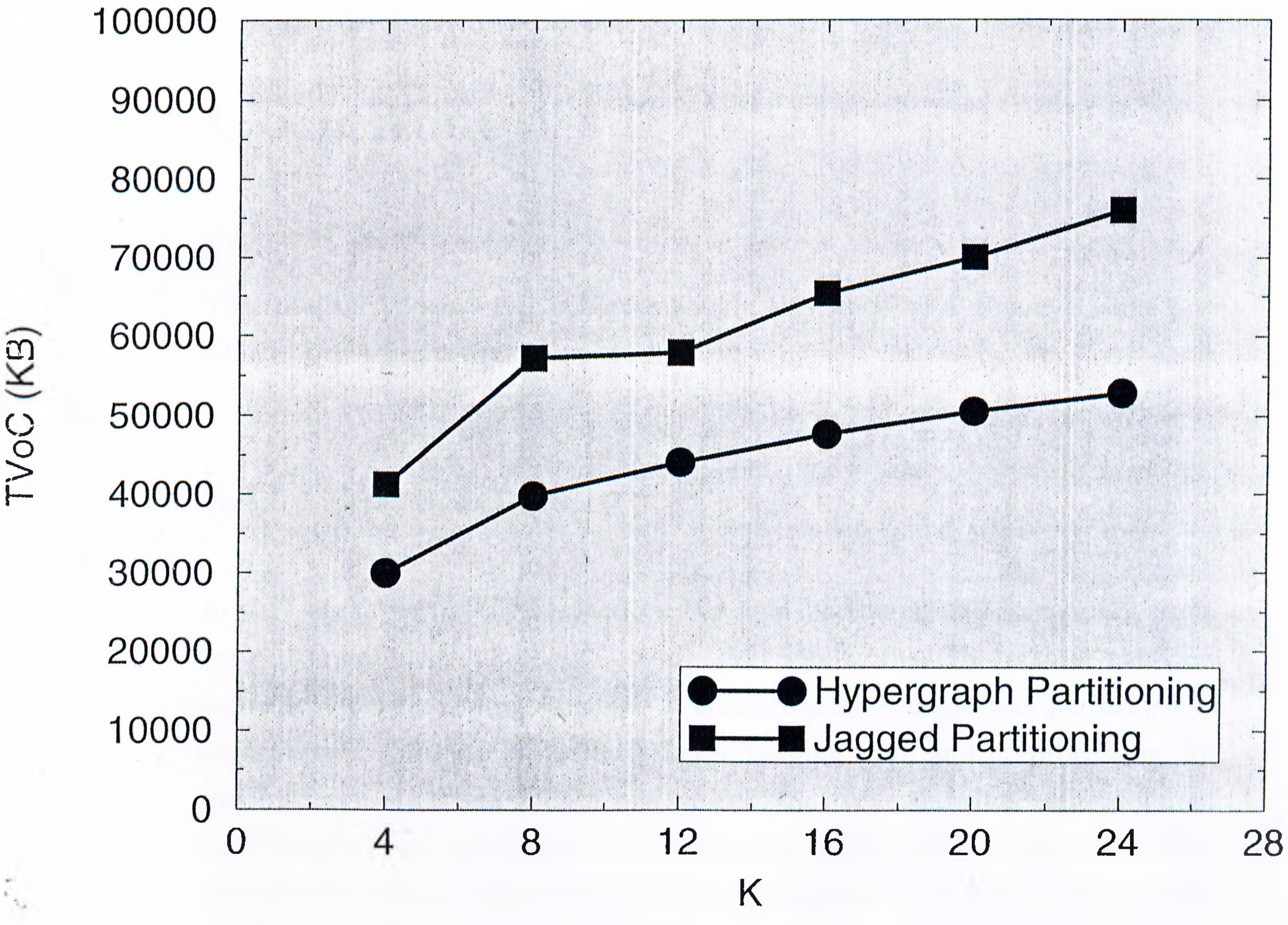


Figure 7.3. A comparison of communication volumes in HP and JP.

Chapter 8

Conclusion

This chapter presents our achievements in this work, and discusses some possible improvements for future studies.

8.1 Work Done

In this work, we mainly focused on the load balancing and remapping problems in image-space parallelization of DVR algorithms. To decrease the view dependent preprocessing overhead a clusterization scheme was performed in object-space. This simplified both house-keeping work and preprocessing, for the exchange of increased data replication. For load balancing, three different work load assignment schemes tested. Among those, inside-outside test which calculates the cluster projection area exactly, appeared to be the most valuable choice.

Furthermore, two different data distribution schemes tested. Compared to the SCA scheme, less communication overhead is observed in NCA scheme. However, since the neighbor clusters contained cells with similar sizes, there occurred load variations during the work load assignment phase. This increased the preprocessing overhead for the NCA scheme.

As a solution to the remapping problem, we proposed a one-phase hypergraph partitioning model. In this model, we represented the interaction between the object-space and image-space by a hypergraph. Partitioning this hypergraph produced screen-processor and data-processor mappings. For hypergraph partitioning, a modified version of PaToH hypergraph partitioning tool was used. Satisfactory values obtained for load imbalance, and remapping costs. At comparable load balance values, the total volume of communication performed in our model is 25% less than the total volume of communication in jagged partitioning, on the average.

8.2 Future Work

A nice feature of our work is that it is open to further improvements. As new heuristics found for hypergraph partitioning or existing hypergraph partitioning tools are improved, the solution quality of our work will also improve. We believe that a possible improvement can be done on the execution time of our hypergraph partitioning code. Since we used the functions in PaToH as external library functions, some code unrelated to our code may be executed, increasing the execution time of our implementation. A more specific hypergraph partitioning code could produce superior timing results.

At the time this work carried out, due to the lack of a K-way partitioning tool, we used a recursive bisection scheme in our partitionings. It is publicly accepted that, direct K-way partitioning approaches are better at optimizing the global objective functions. Hence, we believe that using a direct K-way partitioning scheme for partitioning the remapping hypergraph in our work, we can produce better partitions, in terms of minimization of the communication cost.

Furthermore, we note that, hypergraph partitioning phase, which is run sequentially by each processor, is the limiting factor on the speed-up values. So, a parallel hypergraph partitioning tool, which will probably be implemented in the feature, can eliminate this drawback of our implementation, resulting in much better speed-up values.

Finally, an interesting feature work would be to produce a similar work for object-space parallelization. Our hypergraph partitioning model, with some minor changes can be applied to object-space decomposition. In such a work, instead of minimizing the cluster migration overhead, pixel migration overhead can be tried to be minimized.

Bibliography

[1] C. J. Alpert, A. B. Kahng, Recent Directions in Netlist Partitioning: A Survey, *VLSI Journal*, 19(1-2):1-81, 1995.

[2] J. L. Bentley, Multidimensional Binary Search Trees Used for Associative Searching, *Communications of the ACM*, 18(8):509-517, 1975.

[3] H. Berk, Fast Direct Volume Rendering of Unstrucured Grids, MSc Thesis, Bilkent University, Department of Computer Engineering, 1997.

[4] B. Corrie, P. Mackerras, Parallel Volume Rendering and Data Coherence, *Proceedings of 1994 Symposium on Volume Visualization*, 23-26, 1994.

[5] Ü. V. Çatalyürek, C. Aykanat, PaToH: Partitioning Tool for Hypergraphs, technical report, Department of Computer Engineering, Bilkent University, 1999.

[6] Ü. V. Çatalyürek, C. Aykanat, Hypergraph-Partitioning Based Decomposition for Parallel Sparse-Matrix Vector Multiplication, *IEEE Transactions on Parallel and Distributed Systems*, 10:673-693, 1999.

[7] T. T. Elvins, A Survey of Algorithms for Volume Visualization, *Computer Graphics (ACM Siggraph Quarterly)*, 26(3):194-201, 1992.

[8] T. T. Elvins, Volume Rendering on a Distributed Memory Parallel Computer, *Proceedings of IEEE Visualization '92*, 93-98, 1992.

[9] F. Fındık, Parallel Direct Volume Rendering of Unstructured Grids Based on Object-Space Decomposition, MSc Thesis, Bilkent University, Department of Computer Engineering, 1997.

[10] C. M. Fiduccia, R. M. Mattheyses. A Linear Time Heuristic for Improving Network Partitions. *Proceedings of ACM/IEEE Design Automation Conferance*, 175–181, 1982.

[11] M. P. Garrity, Ray-Tracing Irregular Volume Data, *Computer Graphics*, 24(5):35–40, 1990.

[12] A. V. Gelder, J. Wilhelms, Rapid Exploration of Curvilinear Grids Using Direct Volume Rendering, *Proceedings of IEEE Visualization '93*, 70–77, 1993.

[13] F. Glover, Tabu search - part I, *ORSA Journal on Computing*, 1:190–206, 1989.

[14] L. W. Hagen, D. J. Huang, A. B. Kahng, On Implementation Choices for Iterative Improvement Partitioning Algorithms, *IEEE Transactions on Computer-Aided Design*, 16(10):1199-1205, 1997.

[15] B. Hendrickson, R. Leland, A Multilevel Algorithm For Partitioning Graphs, technical report, Sandia National Laboratories, 1993.

[16] C. L. Jackins, S. L. Tanimoto, Octrees and Their Use in Representing Three-Dimensional Objects, *Computer Graphics and Image Processing*, 14(3):249–270, 1980.

[17] M. Levoy, Efficient Ray Tracing of Volume Data, *ACM Transactions on Graphics*, 9(3):245–261, 1990.

[18] M. Levoy, Display of Surfaces from Volume Data, *IEEE Computer Graphics and Implementations*, 8(3):29–37, 1988.

[19] G. Karypis, V. Kumar, R. Aggarwal, S. Shekhar, hMETIS: A Hypergraph Partitioning Package, technical report, Department of Computer Science and Engineering, University of Minnesota, Army HPC Research Center, Minneapolis, 1998.

[20] G. Karypis, B. Kumar, Multilevel K-way hypergraph partitioning, technical report, University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, 1998.

[21] G. Karypis, R. Aggarwal, V. Kumar, S. Shekhar, Multilevel Hypergraph Partitioning: Applications in VLSI domain, technical report, short version in 34th Design Automation Conference, University of Minnesota, Department of Computer Science and Engineering, 1997.

[22] G. Karypis, V. Kumar, A Fast and Hight Quality Multilevel Scheme for Partitioning Irregular Graphs, *SIAM Journal on Scientific Computing*, (submitted).

[23] B. W. Kernighan, S. Lin, An Efficient Heuristic Procedure for Partitioning Graphs, *Bell Systems Technical Journal*, 49(2):291–307, 1970.

[24] G. Kindlmann, J. W. Durkin, Semi-Automatic Generation of Transfer Functions for Direct Volume Rendering, *Proceedings of 1998 Symposium on Volume Visualization*, 79–86, 1998.

[25] S. Kirkpatrick, C. D. Gelatt, Jr., M. P. Vecchi, Optimization by Simulated Annealing, *Science*, 220:671–680, 1983.

[26] K. Koyamada, Fast Traversal of Irregular Volumes, *Visual Computing, Integrating Computer Graphics with Computer Vision*, 295–312, 1992.

[27] B. Krishnamurthy, An Improved Min-Cut Algorithm for Partitioning VLSI Networks, *IEEE Transactions on Computers*, 33(5):438–446, 1984.

[28] H. Kutluca, Image-Space Decomposition Algorithms for Sort-First Parallel Volume Rendering of Unstructured Grids, MSc Thesis, Bilkent University, Department of Computer Engineering, 1997.

[29] H. Kutluca, T. M. Kurç, C. Aykanat, Image-Space Decomposition Algorithms for Sort-First Parallel Volume Rendering of Unstructured Grids, *Journal of Supercomputing*, 15:51–93, 2000.

[30] H. Kutluca, T. M. Kurç, C. Aykanat, Some Experiments with Communication on a Parsytec CC System, technical report, Department of Computer Engineering, Bilkent University, 1997.

[31] K. Ma, J. Painter, C. Hansen, M. Krogh, A Data Distributed Parallel Algorithm for Ray-Traced Volume Rendering, *Proceedings of 1993 Parallel Rendering Symposium*, 15–22, 1993.

[32] K. Ma, Parallel Volume Ray-Casting for Unstructured-Grid Data on Distributed-Memory Multicomputers, *Proceedings of 1995 Parallel Rendering Symposium*, 23–30, 1995.

[33] K. Ma, T. W. Crockett, A Scalable Parallel Cell-Projection Volume Rendering Algorithm for Three-Dimensional Unstructured Data, *Proceedings of 1997 Parallel Rendering Symposium*, 95–104, 1997.

[34] X. Mao, L. Hong, A. Kaufman, Splatting of Curvilinear Volumes, *Proceedings of IEEE Visualization '95*, 61–68, 1995.

[35] J. Nieh, M. Levoy, Volume Rendering on Scalable Shared-Memory MIMD architectures, *Proceedings of 1992 Workshop on Volume Visualization*, 17–24, 1992.

[36] G. M. Nielson, K. Voegele, An Annotated Bibliography of Scientific Visualization. Part 2, *The Journal of Visualization and Computer Animation*, 2:2–8, 1991.

[37] J. S. Rowlan, G. E. Lent, N. Gokhale, S. Bradshaw, A distributed, Parallel, Interactive Volume Rendering Package, *Proceedings of IEEE Visualization '94*, 21–30, 1994.

[38] L. A. Sanchis, Multiple-Way Network Partitioning, *IEEE Transactions on Computers*, 38(1):62–81, 1989.

[39] D. G. Schweikert, B. W. Kernighan, A Proper Model for the Partitioning of Electrical Circuits, *Proceedings of ACM/IEEE Design Automation Conferance*, 57–62, 1972.

[40] P. Shirley, A. Tuchman, A Polygonal Approximation to Direct Scalar Volume Rendering, *Computer Graphics*, 24(5):63–70, 1990.

[41] R. Shu, A Fast Ray Casting Algorithm Using Adaptive Isotriangular Subdivision, *Proceedings of IEEE Visualization '91*, 232–237, 1991.

[42] C. T. Silva, J. S. B. Mitchell, P. L. Williams, An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes, *Proceedings of 1998 Symposium on Volume Visualization*, 87–94, 1998.

[43] C. T. Silva, A. E. Kaufman, Parallel Performance Measures for Volume Ray Casting, *Proceedings of IEEE Visualization '94*, 196–203, 1994.

[44] J. Wilhelms, A. V. Gelder, P. Tarantino, J. Gibbs, Hierarchical and Parallelizable Direct Volume Rendering for Irregular and Multiple Grids, *Proceedings of IEEE Visualization '96*, 57–64, 1996.

[45] P. L. Williams, Interactive Splatting of Nonrectilinear Volumes, *Proceedings of IEEE Visualization '92*, 37–44, 1992.

[46] C. M. Wittenbrink, Survey of Parallel Volume Rendering Algorithms, *Parallel and Distributed Processing Techniques and Applications*, 1329–1336, Las Vegas, NV, 1998.

[47] C. M. Wittenbrink, Irregular Grid Volume Rendering with Composition Networks, *Proceedings of SPIE Visual Data Exploration and Analysis V, SPIE's Electronic Imaging '98*, 250–260, 1998.

[48] T. S. Yoo, U. Neumann, H. Fuchs, S. M. Pizer, T. Cullip, J. Rhoades, R. Whitaker, Achieving Direct Volume Visualization with Interactive Semantic Region Selection, *Proceedings of IEEE Visualization '91*, 58–65, 1991.

[49] Parsytec GmbH, Germany, *Embedded Parix (EPX) version 1.9.2 User's Guide and Programmers Reference Manual*, 1996.

[50] http://www.nas.nasa.gov/Software/DataSets/

Appendix A

Calculation of Granularity Formula

We approximate the projection area of a 3D volume on the screen by a square containing $n \times n$ pixels, and try to divide the projection area into screen cells containing $g \times g$ pixels. This can be done using $\frac{n^2}{g^2}$ screen cells, at the best case. At the worst case, it requires $\frac{(n-2)^2}{g^2} + \frac{4n-8}{g} + 4$ screen cells (see Figure A.1).

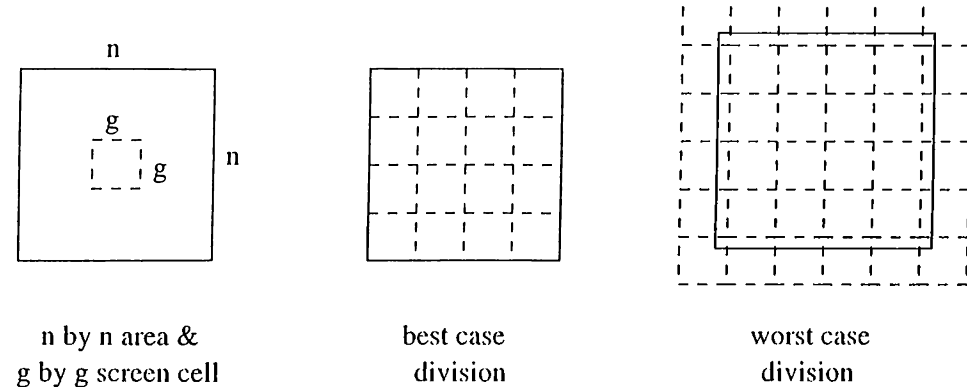


Figure A.1. Imposing g by g screen cells onto an n by n area.

We calculate the average of these two numbers, and equalize it to a fixed total screen cell number, C, that we want to produce. This results in the following second order equation:

$$(2 - C)g^2 + (4n - 8)g + 2n^2 - 4n + 4 = 0 \qquad \text{(A.1)}$$

Table A.1. Adaptive granularity calculation.

screen size	400 × 400	800 × 800	1200 × 1200
C	400	400	400
g	15	31	46
C'	401	375	388
error (%)	0.25	6.25	3.00

Solving this equation for g yields the root in Equation A.2. Note that, we have substituted, $n = \sqrt{A}$ and $n^2 = A$ into that equation.

$$g = \frac{\sqrt{A} - 2 + \sqrt{(C-1)A - 2C\sqrt{A} + 2C}}{C - 2} \tag{A.2}$$

Despite the fact that this formula is just an approximation, and the projection areas on the screen are usually non-square, complex regions, the formula produces pretty good results. Table A.1 displays these results for a fixed $C = 400$ value. C' corresponds to the actual number of screen cells found after the grid is imposed on the screen. Note the linear increase in g with increasing screen size.