IMAGE-SPACE DECOMPOSITION ALGORITHMS FOR SORT-FIRST PARALLEL VOLUME RENDERING OF UNSTRUCTURED GRIDS

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

By
Hüseyin Kutluca
August 1997

IMAGE-SPACE DECOMPOSITION ALGORITHMS FOR SORT-FIRST PARALLEL VOLUME RENDERING OF UNSTRUCTURED GRIDS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND INFORMATION SCIENCE

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Hüseyin Kutluca

[illegible]

August, 1997

QA
166
.K88
1997

BC38383

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.

C. Aykanat

Assoc. Prof. Cevdet Aykanat(Principal 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.

[illegible]

Asst. Prof. Tuğrul Dayar

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.

[illegible]

Asst. Prof. Uğur Güdükbay

Approved for the Institute of Engineering and Science:

M. Baray

Prof. Dr. Mehmet Baray, Director of Institute of Engineering and Science

ABSTRACT

IMAGE-SPACE DECOMPOSITION ALGORITHMS FOR SORT-FIRST PARALLEL VOLUME RENDERING OF UNSTRUCTURED GRIDS

Hüseyin Kutluca
M. S. in Computer Engineering and Information Science
Supervisor: Assoc. Prof. Cevdet Aykanat
August, 1997

In this thesis, image-space decomposition algorithms are proposed and utilized for parallel implementation of a direct volume rendering algorithm. Screen space bounding box of a primitive is used to approximate the coverage of the primitive on the screen. Number of bounding boxes in a region is used as a workload of the region. Exact model is proposed as a new workload array scheme to find exact number of bounding boxes in a rectangular region in $O(1)$ time. Chains-on-chains partitioning algorithms are exploited for load balancing in some of the proposed decomposition schemes. Summed area table scheme is utilized to achieve more efficient optimal jagged decomposition and iterative rectilinear decomposition algorithms. These two 2D decomposition algorithms are utilized for image-space decomposition using the exact model. Also, new algorithms that use inverse area heuristic are implemented for image-space decomposition. Orthogonal recursive bisection algorithm with medians of medians scheme is applied on regular mesh and quadtree superimposed on the screen. Hilbert space filling curve is also exploited for image-space decomposition. 12 image-space decomposition algorithms are experimentally evaluated on a common framework with respect to the load balance performance, the number of shared primitives, and execution time of the decomposition algorithms.

Key words: parallel computer graphics application, volume rendering, sort-first rendering, image-space parallel volume rendering, image-space decomposition, load balancing.

ÖZET

DÜZENSİZ IZGARALARIN ÖNCE-SIRALA ALGORİTMASI KULLANARAK PARALEL HACİM GÖRÜNTÜLENMESİ İÇİN EKRAN UZAYI BÖLÜMLEME ALGORİTMALARI

Hüseyin Kutluca
Bilgisayar ve Enformatik Mühendisliği Bölümü Yüksek Lisans
Tez Yöneticisi: Assoc. Prof. Cevdet Aykanat
Ağustos, 1997

Bu tezde görüntü uzayı bölümleme algoritmaları önerilmiş ve bu algoritmalardan paralel doğrudan hacim görüntüleme algoritması için yararlanılmıştır. Hacim elemanlarının kapsama kutluları onların ekrandaki kapladığı alanı yaklaşık olarak belirlemek için kullanılır. Bir bölgedeki kapsama kutusu sayısı o bölgenin iş yükü olarak kullanılmıştır. Kesin model adında yeni bir iş yükü yöntemi önerilmiştir. Bu yöntem dikdörtkensel bir bölgedeki kapsama kutusu sayısını $O(1)$ zamanında bulmak için kullanılır. Zincir üzerinde zincir parçalama algoritmasından önerilen bazı bölumleme algoritmalarının yük denkliği için yararlanılmıştır. Toplanmış alan tablosu yönteminden daha etkin eniyi kesikli (jagged) bölümleme ve yineli doğrusal bölümleme algoritmaları için yararlanılmıştır. Bu iki 2-boyutlu bölümleme algoritmasından kesin model yöntemi kullanarak görüntü uzayı bölümlemesi için yararlanılmıştır. Aynı zamanda, ters alan sezgisel algoritması kulanan yeni ekran-uzayı bölümleme algoritmaları önerilmiştir. Ortancanın-ortancası yöntemini kullanan dikey özyineli bölme algoritması ekran üzerine yerleştirilmiş düzenli ızgara ve dörtlü ağaca uygulanmıştır. Hilbert uzay doldurma eğriside görüntü uzayı bölümleme için kullanılmıştır. 12 görüntü uzayı algoritması deneysel olarak aynı ortamda yük denkliği, paylaşılan hacim elemanları sayısı ve algoritmaların çalışma zamanı açısından irdelenmiştir.

Anahtar kelimeler: paralel bilgisayar grafiği uygulamaları, hacim görüntüleme, önce-sırala türü görüntüleme, görüntü uzayı paralel hacim görüntüleme, görüntü uzayı bölümleme, yük denkliği.

To my family

Acknowledgment

I would like to express my gratitude to Assoc. Prof. Cevdet Aykanat for his supervision, guidance, suggestions and invaluable encouragement throughout the development of this thesis. I would like to thank th committee members Asst. Prof. Tuğrul Dayar and Asst. Prof. Uğur Güdükbay for reading the thesis and their comments. I would like to thank my colleague Dr. Tahsin Kurç for his technical support, guidance and cooperation in this study.

I would like to thank my family for their moral support. I would like to thank my friends Ozan Özhan, Halil Kolsuz, İçtem Özkaya, Buket Oğuz, Esra Taner, Vedat Mermer and all others for their friendship and moral support.

Contents

List of Figures

List of Tables

1. INTRODUCTION

Rendering in computer graphics can be described as the process of generating a 2-dimensional (2D) representation of a data set defined in 3-dimensional (3D) space. Input to this process is a set of *primitives* defined in a 3D coordinate system, usually called *world coordinate system*, and a *viewing* position and orientation also defined in the same world coordinate system. The viewing position and orientation define the location and orientation of the image-plane, which represents the computer screen. The output of the rendering process is a 2D picture of the data set on the computer screen.

One popular application area of computer graphics rendering is the ray-casting based direct volume rendering (ray-casting DVR) [24, 29] of scalar data in 3D, unstructured grids. In many fields of science and engineering, computer simulations provide a cheap and controlled way of investigating physical phenomena. The output of these simulations is usually a large amount of numerical values. Large quantity of data makes it very difficult for the scientist and researcher to extract useful information from the data to derive some conclusions. Therefore, visualizing large quantities of numerical data as an image provides an indispensable tool for researchers. In many engineering simulations, data sets consist of numerical values which are obtained at points (sample points), with 3D coordinates, distributed in a volume that represents the physical entity or the physical environment. The sample points constitute a *volumetric grid* superimposed on the volume. Sample points are connected to some other nearby sample points to form *cells*. In unstructured grids, sample points in the volume data are distributed irregularly over 3D space and there may be voids in the volumetric grid. Spacing between sample points is variable and there exists no constraint on the cell shapes. Common cell shapes are tetrahedra and hexahedra shapes. Unstructured grids are common in engineering simulations such as computational fluid dynamics (CFD). Unstructured grids are also called *cell oriented* grids. They are represented as a list of cells with pointers to sample points that form the respective cells. Due to cell oriented nature and irregular distribution of sample points, the connectivity information between cells are provided explicitly if it exists. In some applications, simulations do not require a connectivity information. In such cases, the connectivity between cells may not be provided at all. Because of these properties of unstructured grids, algorithms for rendering such grids consume a lot of computer time (usually from tens of seconds to tens or hundreds of minutes). In addition, huge amount of data obtained in scientific and engineering applications, like CFD, requires large memory space. Thus, rendering of unstructured grids is a good candidate for parallelization on distributed-memory multicomputers. Furthermore, many engineering simulations are carried out on parallel machines. Rendering the results on these machines saves the time to transfer vast amounts of data from parallel machines to sequential graphics workstations over possibly slow communication links.

Efficient parallelization of rendering algorithms on distributed-memory multicomputers necessitates decomposition and distribution of data and computations among processors of the machine. There are various classifications for parallelism in computer graphics rendering [10, 12, 34, 48]. Molnar et al. [34] classify and evaluate parallel rendering approaches for polygon rendering. In polygon rendering, the rendering process is a pipeline of operations applied to primitives in the scene. This pipeline is called *rendering pipeline* and has two major steps called *geometry processing* and *rasterization*. Molnar et al. [34] provides a classification of parallelism, based on the point of data redistribution step in the rendering pipeline, as *sort-first* (before geometry processing), *sort-middle* (between geometry processing and rasterization), and *sort-last* (after geometry processing).

Most of the previous work on parallel rendering of unstructured grids evolved on shared-memory multicomputers [6, 8, 31, 49]. Ma [32] presents a sort-last parallel algorithm for distributed-memory multicomputers. In this work, we take a different approach and investigate *sort-first* parallelism for volume rendering of unstructured grids. This type of parallelism was not previously utilized in volume rendering of unstructured grids on distributed-memory multicomputers. In sort-first parallel rendering, each processor is initially assigned a subset of primitives in the scene. A pre-transformation step is applied on the primitives in each processor to find their positions on the screen. This pre-transformation step typically produces screen-space bounding boxes of the primitives. Screen is decomposed into regions and each processor is assigned one or multiple regions of the screen to perform rendering operations. In this thesis, a region is referred to as a subset of pixels on the screen. This subset of pixels may form connected or disconnected regions on the screen. The primitives are then redistributed among the processors using the screen-space bounding boxes so that each processor has the primitives intersecting the region assigned to it. After the redistribution step, each processor performs rendering operations on its region independent of the other processors. Primitives intersecting more than one region. referred to here as *shared primitives*, are replicated in the processors assigned those regions. Thus, total number of primitives in the system may increase after redistribution.

In this thesis. we present algorithms to decompose the screen adaptively among the processors. We experimentally evaluate these heuristics on a common framework with respect to *load balancing performance*. the *number of shared primitives*, and *execution time of the decomposition algorithms*. In previous work on parallel polygon rendering [13, 36, 43, 48], the number of primitives in a region is used to represent the workload associated with that region. That is. the screen is divided into regions and/or screen regions are assigned to processors using the primitive distribution on the screen. In these work. screen-space bounding box of a primitive is used to approximate the coverage of the primitive on the screen. This is done to avoid expensive computations to determine the exact coverage. In the experimental evaluation of the algorithms, the same approximations are used. That is, the number of primitives with bounding box approximation is taken to be the workload of a region for evaluating load balancing performance of the algorithms. The second criteria used in the comparisons is the number of shared primitives after division of the screen. Reducing the number of shared primitives is desirable since they potentially introduce overheads and waste system resources [20]. The most obvious is the waste of memory in the overall machine since such primitives have to be replicated in different processors. They also introduce redundant computations such as geometry processing in polygon rendering, intersection tests in ray tracing [20], etc. Execution time of the decomposition algorithms is another important criteria. A long execution time may take away all the advantages of a particular algorithm.

Operations performed for image-space decomposition are parallelized as much as possible to reduce the preprocessing overhead. Initially, primitives are divided evenly among the processors. Each processor creates screen-space bounding boxes of its local primitives and then creates local workload array using these bounding boxes. A global sum operation is performed over local workload arrays to find the global distribution of primitives on the screen. In some of the decomposition schemes, the partitioning algorithms also run in parallel, whereas in the other schemes the partitioning algorithms are not parallelized because of either their sequential nature or their fine granularity. In the decomposition schemes using parallel partitioning, region-to-processor assignment array is constructed in a distributed manner. Hence, these schemes necessitate a final all-to-all broadcast operation so that each processor gets the all region-to-processor assignments to classify its local primitives for parallel primitive redistribution. In the decomposition schemes using sequential partitioning algorithm, the partitioning algorithm is redundantly and concurrently executed in each processor to avoid the global communication.

In this work, we propose a taxonomy for image-space decomposition algorithms. This taxonomy is based on the decomposition strategy and workload arrays used in the decomposition. We first classify the algorithms based on the dimension of the decomposition of the screen, which is a 2D space. *1D decomposition* algorithms divide the screen in one dimension by utilizing the workload distribution with respect to only one dimension. *2D decomposition* algorithms divide the screen in two dimensions by utilizing the workload distribution with respect to both dimensions of the screen.

Decomposition algorithms are further classified based on the workload arrays used in the decomposition step. Algorithms in the first group utilize *1D workload arrays*, while the algorithms in the second group use *2D workload arrays*. In the first group, *1D workload arrays* are used to find the distribution of workload in each of the dimensions of the screen. In the second group. a 2D coarse mesh is superimposed on the screen and distribution of workload over this mesh is used to divide the screen. We introduce two models. referred to here as *inverse area heuristic* (IAH) model and *exact* model, to create workload arrays and query the workload in a region. In the IAH model, an estimate of the workload in a region can be found, whereas in exact model, exact workload in a region can be found. IAH model allows rectangular or non-rectangular regions as well as regions consisting of non-adjacent cells when 2D arrays are used. However, exact model can only be used for rectangular regions consisting of adjacent mesh cells. Our formulation of IAH model needs only one 2D workload array, but exact model requires four 2D workload arrays.

Among the workload arrays described above, the exact model is a new model. It is proposed to find the exact number of bounding boxes in a rectangular region in $O(1)$ time.

In this work. chains-on-chains partitioning (CCP) problem is investigated and exploited for optimal 1D decomposition of image space. Optimal load balancing in 1D decomposition schemes can be directly modeled as the CCP problem. We have also investigated the usage of CCP algorithm in load balancing of two different 2D decomposition scheme. The first algorithm finds an optimal jagged decomposition. The second one is an iterative heuristic for rectilinear decomposition. Summed area table (SAT) is a well known data structure. especially in computer graphics area. to query the total value of a rectangular region in $O(1)$ time. We exploit SAT for more efficient optimal jagged decomposition and iterative rectilinear decomposition algorithms. An optimal jagged decomposition algorithm that uses efficient probe-based CCP algorithm is proposed.

For the implementation of optimal jagged decomposition (OJD-E) and iterative rectilinear decomposition (RD) for image space decomposition, the exact model is used for a workload array. Here, the exact model takes the role of SAT for a workload array of general type. The same jagged decomposition algorithm is implemented with workload array that is generated using IAH. Note that this scheme does not give optimal solution for rectangle distribution.

There are also some spatial decomposition algorithms implemented for image space decomposition. Orthogonal recursive bisection (ORB) algorithm with

IAH as a workload array was presented by Mueller [36] for adaptive image-space decomposition (referred as a MAHD in their work). In this work, two new algorithms are implemented to alleviate load imbalance problem due to the straight division line in ORB. One scheme is orthogonal recursive bisection with medians-of-medians on cartesian mesh (ORBMM-M) [41]. This scheme also divides the median division line to achieve better load balance. In the second scheme, a quadtree is superimposed on the mesh to reduce the errors due to IAH, and then this quadtree is decomposed using orthogonal recursive bisection with medians-of-medians (ORBMM-Q) [26, 41, 45]. Another image-space decomposition algorithm implemented in this work uses a class of space filling curve, hilbert curve, for decomposition of image space (HCD). In this scheme, the 2D coarse mesh is traversed with a space filling curve. Then, the mesh cells are assigned to processors such that each processor gets the cells that are consecutive in this traversal. Image-space decomposition algorithms proposed in [27] are also presented and experimentally evaluated for the sake of completeness of comparison. The first algorithm, decomposes the image space in one dimension with recursive bisection heuristic using 1D workload arrays (HHD). The second one is a heuristic version of the jagged decomposition algorithm (HJD) that uses 1D workload arrays. Similarly, the ORB decomposition scheme is also implemented using 1D workload arrays (ORB-1D). Finally image-space decomposition is modeled as a graph partitioning problem [26] and state-of-the-art graph partitioning tool MeTiS is used for partitioning the generated graph (GPD).

The algorithms proposed and presented in this work are utilized for parallel implementation of a volume rendering algorithm for visualizing unstructured grids. The sequential volume rendering algorithm is based on Challinger's work [6, 8]. This algorithm is a polygon rendering based algorithm. It requires volume elements composed of polygons and utilizes a scanline z-buffer approach for rendering. We discuss the application of the decomposition algorithms for this volume rendering algorithm. We present experimental speedup figures for rendering of benchmark volume data sets on a Parsytec CC[1] system. We observe that only the number of primitives in a region does not provide a good approximation to actual computational load. The number of spans and pixels generated

[1]Parsytec CC is a registered trademark of PARSYTEC GmbH

during the rendering of primitives were incorporated into the algorithms to approximate workload better. It has been experimentally observed that speedup values are almost doubled using these additional factors.

In the parallel algorithm, after the screen is decomposed into regions, the local primitives are redistributed according to region-to-processor assignment. Each processor needs to classify its local primitives. For decompositions that generates rectangular regions, rectangle intersection based algorithm is used for classification. For decompositions that generate non-rectangular regions, mesh based algorithm is used. In this thesis, a new classification scheme is proposed for horizontal, rectilinear and jagged decomposition schemes. The proposed algorithms exploit the structured decomposition of the screen with these schemes and classify the primitives. For horizontal decomposition, it uses inverse mapping function that inversely maps the scanlines to processors. This inverse mapping array is used for classification. Rectilinear and jagged decomposition exploits the same idea many times. This classification scheme is more efficient than both of other schemes as it is less dependent to the number of processors and mesh resolution.

The organization of this thesis is as follows: Chapter 2 presents previous work on parallel volume rendering of unstructured grids and on sort-first parallelism in computer graphics rendering, and motivations of this work. Chapter 3 presents the sequential and parallel algorithm for ray-casting-based DVR of unstructured grids. Workload model and taxonomy for image-space decomposition are also presented in Chapter 3. CCP, iterative rectilinear decomposition and optimal jagged decomposition algorithms are discussed in Chapter 4. Chapter 5 describes the creation of workload arrays and presents image-space decomposition algorithms. Primitive classification algorithms for redistribution are given in Chapter 6. Chapter 7 presents the experimental results. Chapter 8 evaluates the contribution of the thesis. Appendix A presents the experimentation with the communication performance of the Parsytec CC system.

2. PREVIOUS WORK AND MOTIVATIONS

This chapter summarizes previous work on parallel volume rendering of unstructured grids and on sort-first parallelism in computer graphics rendering and our motivations.

2.1 Previous Work

Mueller [36] presents a sort-first parallel polygon rendering algorithm for interactive applications. Static and adaptive division of the screen is examined for load balancing. In static decomposition scheme, the screen is decomposed into rectangular regions which are assigned to processors in a round-robin fashion using a scattered assignment for load balancing. In this assignment strategy, adjacent regions are assigned to different processors such that processor i is assigned regions i, $i+P$, $i+2P$, and so on. Here, P denotes the number of processor in the multicomputer. In adaptive decomposition scheme, the screen is decomposed adaptively using the distribution of triangles on the screen until the number of regions is equal to the number of processors. In order to find the distribution of triangles on the screen, a coarse mesh is superimposed on the screen. The number of primitives, which cover the mesh cell, is counted for each mesh cell. An amount inversely proportional to the number of cells a primitive covers is added to corresponding mesh cell count to avoid errors caused by counting large primitives multiple times. A single processor collects counts from each processor and forms a summed-area table [11], which has the same resolution as the fine mesh. This processor divides the screen recursively in alternate directions at each step using the summed-area table. The summed-area table allows binary search to determine the division line. The screen decomposition information is broadcast to each processor so that primitives are re-distributed according to new decomposition. Adaptive decomposition exploits frame-to-frame coherence existing in interactive applications. Current frames distribution is used to perform decomposition for the next frame. Static and adaptive decomposition schemes are evaluated experimentally using a simulator with respect to various factors such as number of regions, mesh resolution, effect of the number of processors.

Challinger [6, 7, 8] presents parallel algorithms for BBN TC2000[1] multicomputer, which is a distributed shared memory system. In the former work of Challinger [6], two algorithms are presented. In the single-phase algorithm, each scanline on the screen is considered as a task. Dynamic task allocation on a demand-driven basis is performed to assign scanlines to processors. In this scheme, each processor gets a scanline to render when it becomes idle. After receiving a scanline, each processor creates local x-buckets using the active cells at the current scanline. Each processor, then, creates an intersection list at the current pixel using the local x-bucket. The intersection list is then processed to perform composition. In the two-phase algorithm, the sampling and composition steps are separated as two phases. Scanlines on the screen are scattered to processors in a round-robin fashion statically. In the sampling phase, processors sweep through scanlines assigned to them and create intersection lists for each pixel on each scanline assigned to them. These intersection lists are stored in the local memories. In composition step, each of these intersection lists are processed to perform composition of sample values for the corresponding pixel. In two-phase algorithm, since intersection lists are saved, when a new transfer function is used to generate colors, only composition phase is executed. The main disadvantage of scanline based task generation is the low scalability. The scalability of these two algorithms [6] is limited by the number of scanlines on the screen. In the latter work of Challinger [7, 8], image-space is divided into square tiles which are considered as tasks assigned to processors dynamically. Image tiles are sorted according to the number of cells (primitives) associated to them, and they are assigned to processors in this sorted order to achieve better load balancing.

Williams [49] presents algorithms for parallel volume rendering on Silicon

[1]BBN TC2000 is a trademark of BBN Advanced Computers, Inc

Graphics Power Series (SGIPS)[2]. The target machine is a shared-memory multicomputer with computer graphics enhancement through the use of graphics processors. The processors in SGIPS do not contain local memories and access to shared memory is over a bus. The serial algorithms for direct volume rendering are based on object-space methods (such as projection and splatting). The cells are view sorted for proper composition by the view sort technique developed by Williams [49, 50]. The sorting technique. called meshed polyhedra visibility ordering (MPVO) algorithm, topologically sorts an acyclic directed graph generated from connectivity relation between cells. The topological sort is done by using either breadth-first search (BFS) or depth-first search (DFS) techniques on directed graph. Parallelization of the algorithms involves two stages: (1) parallelization of generating directed graph used by the MPVO algorithm and (2) parallelization of topological view sort of the graph and rendering of the view sorted cells. Stage (1) is parallelized by assigning a cell (volume primitive) to each processor to process. Each processor keeps local data structures (queues) to store the "source cell" used in the view sorting phase. These data structures are then merged and stored in the global memory. Different parallel algorithms are presented in stage (2) for convex and non-convex grids. Two schemes are presented for convex grids. In the first scheme, each processor takes a source cell from global queue and splats it onto the screen. Since BFS on the graph produces cells that are spatially not overlapping, splatting of the cells can be done in parallel. Then. each processor finds the children of the source cell it splats and puts them into a local queue. When all source cells in the global queue are processed, local queues are merged into global queue. In the second scheme, two global queues are used. A single processor selected as host processor performs BFS on the graph using source cells in the first global queue. This processor finds the children of all source cells in the first global queue and stores them in the second global queue, while other processors splat the cells in the first global queue. After all cells in the first global queue are processed and host processor finishes constructing the first queue, pointers to global queues are exchanged. If host processor finishes its work before others, it also helps splatting of the cells in the first queue. The MPVO algorithm for non-convex grids requires DFS of

[2]Silicon Graphics Power Series is a trade mark of Silicon Graphics Inc.

the graph. Host processor performs DFS on the graph and the other processors perform the splatting of the cells. Two queues are used for this purpose. While host processor updates first queue. cells in the second queue are processed. Since cells need to be processed in the order they are output from the DFS routine. only limited amount of work can be parallelized such as transformation of cells and partitioning of cells for projection.

Lucas [31] describes a volume rendering algorithm for shared-memory multicomputers. The algorithm consists of two steps. In the first step. viewing transformations and lighting calculations are done. These calculations are performed on partitions of the volume data set. The data set is partitioned into rectangular regions. Unstructured data sets are partitioned by dividing the data recursively. Details of how to perform the decomposition are not given in the paper. The second step of the algorithm is the rendering of the volume partitions. In this step. screen is divided into non-overlapping rectangular regions and processors render one or more screen regions. Each screen region is processed in three steps; checking each volume partition if it falls into corresponding screen region. then checking each primitive in the partition for quick rejection of totally clipped primitives, and clipping and scan-converting primitives that intersect the screen region. The effect of the number of screen regions and the number of volume partitions to the algorithm performance is examined to obtain an optimum division of the screen and volume data set. It is unclear from the paper how screen regions are assigned to processors for achieving even load distribution.

Ma [32] presents a sort-last parallel algorithm for distributed memory multicomputers. The multicomputer used in Ma's work is an Intel Paragon[3] with 128 processors. In Ma's algorithm, the volume data is divided into P subvolumes. where P is the number of processors. The volume is considered as a graph and partitioned into subvolumes of equal number of volume cells (e.g.. tetrahedrals) using Chaco graph partitioning tool [19]. The ray-casting volume rendering algorithm of Garrity [14] is used to render subvolumes in each processor. The subvolumes may have local exterior faces due to partitioning and it is possible that rays will exit from these faces and re-enter the volume from such faces, creating ray segments. Composition operations on color and opacity values are

[3]Intel Paragon is a trademark of Intel Corporation

associative, but *not* commutative. Thus, each processor inserts ray-segments (in sorted order) to linked lists. The partial images in each processor are composited to generate the final rendered image. In image-composition, screen is divided evenly into horizontal bands. Each processor is assigned a band to perform image-composition. The linked lists in each processor are packed and sent to respective processors for composition. Each processor unpacks the received lists and sorts them. Then, these sorted lists are merged for the final image. Ma overlaps sending of ray segments with rendering computations to reduce the overhead of communication.

2.2 Motivations

Most of the previous work on parallel rendering of unstructured grids were done on shared-memory multicomputers [6, 8, 31, 49]. The algorithms developed in [49] can be considered as fine-grain algorithms and exploit the use of shared memory in the system. Load balancing is done dynamically by assigning a cell to the idle processor for rendering. Such an assignment scheme will introduce substantial communication overhead due to fine granularity of the assignments. In addition, parallel algorithms developed for sorting the cells require a global knowledge of the database. Therefore, these algorithms are not very suitable for distributed-memory multicomputers.

In [6, 8, 31], screen is decomposed into equal size regions and load balancing is achieved by dynamic allocation of regions to processors on a demand-driven basis [6, 8] or by scattered assignment [6]. Scattered assignment has the advantage that assignment of screen regions to processors is known a priory and static irrespective of the data. However, since scattered assignment assigns adjacent regions to different processors, it loses the coherency in image-space and increases the duplication of polygons in the overall system. In addition, since decomposition is done irrespective of input data, it is still possible that some regions of the screen is heavily loaded and some processors may perform substantially more work than others. In demand-driven approaches, regions are assigned to processors when they become idle. Demand-driven assignment may incur substantial communication overhead in distributed-memory multicomputers. First of all, since region assignments are not known a priory, each assignment should be broadcast to all processors so that necessary polygon data is transmitted to the corresponding processor. In addition, since many processors will inject polygons to the network for different processors or for the same processor many times it is very likely that dynamic scheme will introduce high network congestion. Another disadvantage of the dynamic allocation is that adjacent regions may be assigned to different processors, which results in loss of coherency and increase in the number of primitives replicated. So, adaptive decomposition of the screen is a good alternative to these non-adaptive decomposition schemes.

Ma [32] uses sort-last parallelism. The volume is partitioned using a graph partitioning tool into subvolumes of equal number of elements. Unfortunately, the sequential rendering algorithm employed in the implementations is very slow. Thus, it hides many overheads of the parallel implementation. For example, image-composition operations take seconds even on large number of processors. In addition, composition time does not decrease linearly with increasing number of processors. This is basically due to sorting required on ray-segments for correct composition of colors and opacities. Moreover, even when viewing parameters are fixed (to visualize volume under different transfer functions), inter-processor communication is still needed for image-composition.

In this work, we take a different approach and investigate *sort-first* parallelism for volume rendering of unstructured grids. This type of parallelism was not previously utilized in volume rendering of unstructured grids on distributed-memory multicomputers.

3. RAY-CASTING BASED DVR OF UNSTRUCTURED GRIDS

Figure 3.1, based on the illustration by Yagel [51], illustrates types of grids that are commonly encountered in volume rendering. The common characteristic of the structured grids is that sample points are distributed regularly in 3-dimensional space. The distance between sample points may be constant or variable. Although this type of distribution is obvious in *cartesian*, *regular*, and *rectilinear* grids, this situation is not so obvious in *curvilinear* grids. In curvilinear grids, sample points are distributed in such a way that the grid fits onto a curvature in space. Hence, there exists a regularity in the distribution of sample points and this type of grids are also categorized as structured grids. The cell shapes in structured grids are hexahedral cells formed by eight sample points. These type of grids are also called *array oriented* grids since these grids are usually represented as a 3-dimensional array, for which there exists a one-to-one correspondence between array entries and sample points. Due to array oriented nature of structured grids, the connectivity relation between cells are provided implicitly. In unstructured grids, on the other hand, sample points in the volume data are distributed irregularly over three dimensional space and there may be voids in the volumetric grid. The spacing between sample points is variable. There exists no constraint on the cell shapes. Common cell shapes are tetrahedra and hexahedra shapes. Unstructured grids are common in engineering simulations such as computational fluid dynamics (CFD), finite volume analysis (FVA) simulations, and finite element methods (FEM). In addition, *curvilinear* grid types are also common in CFD. Unstructured grids are also called *cell oriented* grids. They are represented as a list of cells with pointers to sample points that form the respective cells. Due to cell oriented nature and irregular distribution of sample points,

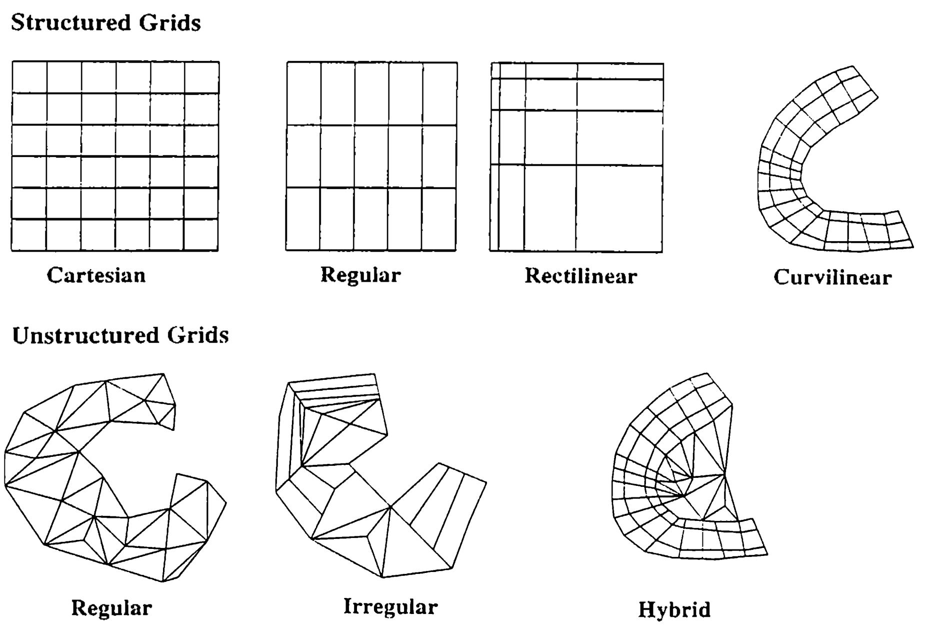


Figure 3.1. Types of grids encountered in volume rendering.

the connectivity information between cells are provided explicitly if it exists. In some applications, simulations do not require a connectivity information. In such cases, the connectivity between cells may not be provided at all. Unstructured grids can further be divided into three subtypes as *regular*, in which cell shapes are consistent and usually tetrahedral cells with at most two cells sharing a face, *irregular*, in which there is not consistency in cell shapes and a face may be shared by more than two cells, and *hybrid*, which is the combination of structured and unstructured grids.

In ray-casting DVR [29, 30, 47], a ray is cast from each pixel location and is traversed throughout the volume. In this work, the term *direct volume rendering* (DVR) refers to the process of visualizing the volume data without generating an intermediate geometrical representation such as isosurfaces [24]. The color value of the pixel is calculated by finding contributions of the cells intersected by the ray at the sample points on the ray and integrating these contributions along the ray. The scalar values, computed as contributions of the cells, at the sample points on the ray are converted into color and opacity values using a transfer function. The color and opacity values are then composited in a pre-determined sorted order (either back-to-front or front-to-back) [29, 30] to find the color of the associated pixel on the screen. The composition operation is associative but *not* commutative. The traversal of ray through the volume and calculating the color of the pixel introduces two problems referred to here as *point location* and *view sort* problems. Efficient solution of these problems is crucial to the performance of the underlying algorithm. Determining the volume element that contains the sample point on the ray in the re-sampling phase is called *point location* problem. For unstructured grids, it involves finding the intersection of the ray with the cell. Sorting sample points on the ray or finding the intersections in a sorted order is defined as *view sort* problem. Solving point location and view sort problems is difficult in *unstructured* grids because data points (original sample points), hence volume elements, are distributed irregularly over 3D space. A naive algorithm may need to search all cells to find an intersection, thus requiring very large execution times for large data sets. In addition, sorting sample points on a ray takes a lot of time, if not handled efficiently, because many cells may be intersected by the ray. Therefore, the performance of the underlying algorithm closely depends on how efficiently it resolves these problems. In the next section, a scanline z-buffer based algorithm, which utilizes image and volume coherency to resolve these problems efficiently, is presented.

3.1 Sequential Algorithm: A Scanline Z-buffer Based Algorithm

The sequential rendering algorithm chosen for DVR is based on the algorithm developed by Challinger [7, 8]. This algorithm adopts the basic ideas in standard polygon rendering algorithms. As a result, the algorithm requires that volumetric data set is composed of cells with planar faces. However, this algorithm does not require a connectivity information between cells, and provides a general algorithm to handle volume grids. In this work, it is assumed that volumetric data set is composed of tetrahedral cells. If a data set contains volume elements that are not tetrahedral, these elements can be converted into tetrahedral cells by subdividing them [14, 44]. A tetrahedral cell has four points and each face of the tetrahedral cell is a triangle, thus easily meeting the requirement of cells with planar faces. Since the algorithm operates on the polygons, the tetrahedral data set is further converted into a set of distinct triangles. Only triangle information are stored in the data files.

The algorithm processes consecutive scanlines of the screen from top to bottom, and processes the consecutive pixels of a scanline from left to right. Basic steps of the algorithm is given below:

1. Read volume data. In our case, the algorithm reads triangles representing faces of tetrahedrals from the data files.

2. Transform the triangles into screen coordinates by multiplying each vertex by a 4×4 transformation matrix. Perform *y-bucket sort* on the triangles. The *y-bucket* is a 1D array of pointers that point to triangles of the input database. Each entry of the y-bucket corresponds to a scanline on the screen and a linked list of pointers is stored at each entry. The pointer to the triangle is inserted at the entry which corresponds to the lowest numbered scanline that intersects the triangle.

3. Update *active polygon* and *active edge* lists for each new scanline, starting from the lowest scanline and continuing in increasing scanline number. The *active polygon* list stores the triangles that are starting and continuing at the current scanline. Before processing the current scanline, the corresponding entry of the y-bucket is inspected for new triangles. If there are new triangles, they are inserted into active polygon list. At the end of processing the scanline. triangles that end at the current scanline are deleted from the active polygon list. The *active edge* list stores the triangle edges that are intersected by the current scanline. Edges of triangle in the active polygon list are tested for the intersection. Note that if a triangle is already in the active polygon list, then a pair of its edges is in the active edge list. For such triangles, new edge intersections are calculated incrementally using the edge information in the active edge list.

4. For each active edge pair for the current scanline, generate a span, clip the span to the region boundaries, and insert it in *x-bucket*. The *x-bucket* is 1D array of pointers. Each entry corresponds to a pixel location on the current scanline and stores a linked list of spans starting at that pixel location.

5. Update *z-list* for each new pixel on the current scanline. The *z-list* is a linked

list and each entry of the z-list stores the z-intersection of the triangle with the ray shot from the pixel location, span information, a pointer to the triangle, and a flag to indicate whether the triangle is an exterior or an interior face. Note that two consecutive triangles, if at least one of them is an interior triangle, make up the corresponding tetrahedral cell in the volume. Hence, during the composition step, two consecutive triangles can be used for the determination of the sampling points on the ray. The z-intersections are calculated by processing the spans stored in the x-bucket. The z-intersections are updated incrementally by rasterizing spans. Each z-intersection is inserted into the z-list in such a way that the list remains sorted in increasing z-intersection values. The z-list can also be considered as an active span list because only the span information for the spans that are active at the current pixel location is inserted into the list. Note that as long as no new spans are inserted, there is no need to sort the list again for the next pixel.

6. Composite the sample values for the current pixel location using z-list ordering. Repeat steps 3–6 until all scanlines and pixels are processed.

The algorithm exploits image-space coherency for efficiency. The calculations of intersections of polygons with the scanline, insertion and deletion operations on the active polygon list are done incrementally. This type of coherency is referred to here as *inter-scanline* coherency. For each pixel on the current scanline, the intersection of the ray shot from the pixel and spans that cover that pixel are determined and put into the *z-list*, which is a sorted linked list, in the order of increasing z-intersection values. The z-intersection calculations, sorting of z-intersection values, insertion to and deletion from z-list are done incrementally. This type of coherency is referred to here as *intra-scanline* coherency.

3.2 Parallel Algorithm

Parallel algorithm is a sort-first parallel rendering algorithm. This algorithm consists of the following basic steps:

1. Read volume data. Initially, each processor receives V/P triangles. Here, V is the total number of triangles and P is the number of processors.

2. Divide the screen into regions. The screen is partitioned into P subregions using the distribution of workload on the screen. The decomposition is performed using one of the image-space decomposition algorithms presented in Section 5.2. After regions are created, each processor is assigned a screen region. The local triangles in each processor are re-distributed according to new screen regions and processor-region assignments. Each processor exchanges triangle information to receive triangles intersecting the region it is assigned. It sends the triangle information belonging to other regions to respective processors.

3. Perform steps 2–6 of the sequential algorithm on the local screen region.

3.3 Workload Model

The screen is divided into regions using one of the image-space decomposition algorithms described in the Section 5.2. Determining the actual computational workload in a region is crucial to achieve even distribution of computational load among processors. As stated in the earlier sections, number of primitives are used to approximate the workload in a region in previous work on polygon rendering [13, 36, 43, 48]. We use the same approximations in the experimental comparison of the image-space decomposition algorithms. However, in the sequential and parallel algorithms given in sections 3.1 and 3.2, there are three parameters that affect the computational load in a screen region. First one is the number of triangles (primitives), because the total workload due to transformation of triangles, insertion operations into y-bucket and insertions into and deletions from active polygon list are proportional to the number of triangles in a region. The second parameter is the number of scanlines each triangle extends. This parameter represents the computational workload associated with the construction of edge intersections (hence, corresponding spans), clipping of spans to region boundaries, and insertion of the spans into x-bucket list. The total number of pixels generated by rasterization of these spans is the third parameter affecting the computational load in a region. Each pixel generated adds computations required for sorting, insertions to and deletions from z-list, interpolation and composition operations. The operations on each parameter takes different amount of time. Therefore, the workload (WL) in a region can be approximated using Eq. (1).

$$WL = aN_T + bN_S + cN_P \tag{1}$$

here N_T, N_S, and N_P represent the number of triangles, spans (number of scanlines triangles extend), and pixels (generated by rasterizing the triangle), respectively, to be processed in a region. The values a, b, c represent the relative computational costs of operations associated with triangles, spans, and pixels, respectively. Finding exact number of pixels and spans generated in a region due to a triangle requires rasterization of the triangle. In order to avoid this overhead, the bounding box approximation is used for pixels and spans. That is, a triangle with a bounding box with corner points $(xmin, ymin)$ and $(xmax, ymax)$ is assumed to generate $ymax - ymin + 1$ spans and $(ymax - ymin + 1) \times (xmax - xmin + 1)$ pixels.

In the discussions of the image-space decomposition algorithms, we assume that the workload of a region is the number of primitives (based on the bounding box approximation) in that region. Incorporating the pixels and spans (Eq. 1) to these algorithms is accomplished by treating each span and pixel covered by the bounding box of the triangle as bounding boxes with computational loads of b and c, respectively. That is, for a triangle whose bounding box has corner points $(xmin, ymin)$ and $(xmax, ymax)$, there is one triangle with computational load of a, there are $ymax - ymin + 1$ triangles, whose height is one pixel and width is $xmax - xmin + 1$, each with computational load of b, and there are $(ymax - ymin + 1) \times (xmax - xmin + 1)$ triangles, whose height and width are one pixel, each with a computational load of c.

3.4 Image-Space Decomposition Algorithms

In this work, we propose algorithms that divide the screen adaptively using the workload distribution on the screen. The algorithms discussed in this thesis have the following basic steps:

1. Create screen space bounding boxes of the primitives (triangles). Initially, each processor receives V/P primitives. Here, V is the total number of

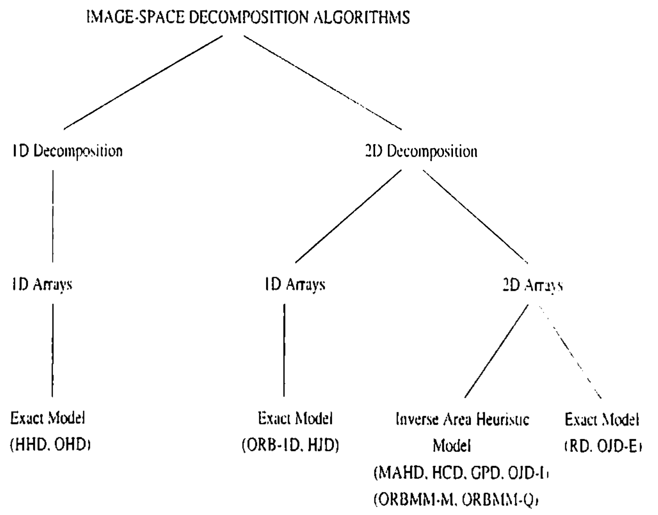


IMAGE-SPACE DECOMPOSITION ALGORITHMS
1D Decomposition
2D Decomposition
1D Arrays
1D Arrays
2D Arrays
Exact Model (HHD, OHD)
Exact Model (ORB-1D, HJD)
Inverse Area Heuristic Model (MAHD, HCD, GPD, OJD-I) (ORBMM-M, ORBMM-Q)
Exact Model (RD, OJD-E)

Figure 3.2. The taxonomy of image-space decomposition algorithms.

primitives and P is the number of processors. After receiving the primitives, each processor creates screen space bounding boxes of the local primitives.

2. Create the workload arrays using the distribution of primitives on the screen.

3. Decompose the screen into P regions using the workload arrays. Each processor is assigned a single region after decomposition.

4. Redistribute the local primitives according to screen regions and processor-region assignments. In order to carry out redistribution step, each processor should know about the region assignments to other processors. For this reason, each processor receives screen decomposition information from other processors if such information is distributed among processors during decomposition.

Each of the steps 2–4 are described in the following sections.

3.5 A Taxonomy of the Decomposition Algorithms

In this section, we propose a taxonomy for the decomposition algorithms proposed and presented in this thesis. This taxonomy is based on the decomposition strategy and workload arrays used in the decomposition.

We first classify algorithms based on the decomposition of the screen, which is a 2D space. There are basically two ways to decompose the screen. *1D decomposition* algorithms divide in only one dimension of the screen. These algorithms utilize the workload distribution with respect to only one dimension. *2D decomposition* algorithms, on the other hand, utilize the workload distribution with respect to both dimensions of the screen. They divide the screen in two dimensions.

We can further classify the algorithms based on the workload arrays used in the decomposition step. The term *arrays* will also be used to refer to *workload arrays*. Algorithms in the first group utilize *1D workload arrays*, while the algorithms in the second group use *2D workload arrays*. In the first group, *1D workload arrays* are used to find the distribution of workload in each of the dimensions of the screen. In the second group, a 2D coarse mesh is superimposed on the screen and distribution of workload over this mesh is used to divide the screen.

We introduce two models, referred to here as *inverse area heuristic* (IAH) model and *exact* model, to create workload arrays and query the workload in a region. In the IAH model, an estimate of the workload in a region can be found, whereas in the exact model, exact workload in a region can be found. IAH model allows rectangular or non-rectangular regions as well as regions consisting of non-adjacent cells when 2D workload arrays are used. However, exact model can only be used for rectangular regions consisting of adjacent mesh cells. Our formulation of IAH model needs only one 2D workload array, but exact model requires four 2D workload arrays.

The classification of the algorithms presented in this work is illustrated in Fig. 3.2. Abbreviations of the names of the algorithms are given in the parentheses, please refer to Section 5.2 for full names of the algorithms.

4. DECOMPOSITION USING CCP ALGORITHMS

In this chapter, chains-on-chains partitioning problem (CCP) is discussed. Optimal load balancing problem in the decomposition of 1D workload arrays can be modeled as CCP. Hence, CCP algorithms can be exploited for optimal decomposition of 1D domains. Beside, 2D decomposition schemes that utilize CCP algorithm are described. An iterative heuristic for rectilinear decomposition and an optimal jagged decomposition are discussed. This chapter presents decomposition algorithms for general workload arrays. Adaptation of these algorithms to image-space decomposition will be discussed.

4.1 Chains-On-Chains Partitioning Problem

Chains-on-chains partitioning problem is defined as follows: We are given a chain of work pieces called modules $A_1, A_2 \ldots A_N$ and wish to partition the chain into P subchains, each subchain consisting of consecutive modules. The cost function $W_{i,j}$ is defined as the cost of subchain $A_i, \ldots A_j$. Cost function must be non-negative and monotonically non-decreasing. The chain of modules can be partitioned optimally in polynomial time with an objective function that minimizes the cost of maximally loaded subchain. The subchain with maximum load is called the *bottleneck subchain* and the load of this subchain is called *bottleneck value* of the partition.

CCP problem arises in many parallel and pipelined computing applications. Such applications include image processing, signal processing, finite elements, linear algebra and sparse matrix computations. Their common characteristics are that, the divisible part of the domain is represented as a workload modules and dividing the domain to subdomains with contiguity constraint is necessary for the efficiency of the parallel program. In some applications, like image processing or finite elements applications, the modules need the value of its neighbor modules. Therefore, for efficient parallelization it is necessary to put contiguous modules to the same processor. Moreover, for the applications in linear algebra and computer graphics the non-contiguity results in inefficient computation power and more volume of communication. The load balance among the parts is necessary, as the parallel execution time is determined by the *bottleneck processor*.

Bokhari first studied the chain structured computations [3] and proposed polynomial time algorithm with complexity $O(N^3P)$ [4] for optimal partitioning. Then several algorithms have been proposed with better complexities. A dynamic programming (DP) based approach with a complexity of $O(N^2P)$ was proposed by Anily and Federgruen [1], and Hansen and Lih [18] independently. Later, Choi and Narahari [9], and Olstad and Manne [39], independently improved the DP-based approach to complexities of $O(NP)$ and $O((N-P)P)$ respectively. Iqbal and Bokhari [22], and Nicol and O'Hallaron [38] proposed $O(NP \log N)$ time algorithms. These algorithm are based on a function called *probe*. The probe function accepts a candidate value and determines if a partition exists with a bottleneck value less than the given candidate value. The partitioning strategy is based on repetitively calling the probe function for candidate values and finding an optimal solution. The complexity of this algorithm is better than the $O(NP)$ complexity of DP-based approaches if $P = N/(\log N)^2$. This strategy is more useful when there are many modules comparing to the number of partitions (processors). Iqbal [21] also give a probe-based approximation algorithm with a complexity of $O(NP \log(W_{tot}/\epsilon))$. where W_{tot} is the total workload and ϵ is the desired precision.This algorithm becomes an exact algorithm for integer-valued workload arrays with $\epsilon = 1$.

The cost function for a subchain may change according to the application. However, all must satisfy the non-negative and monotonically non-decreasing behavior. Moreover, for the algorithms given in next sections, it is assumed that the cost function $W_{i,j}$ is calculated in $O(1)$ time. If the cost of each module is independent, which is generally the case, then each module A_i is associated with a weight w_i and cost of subchain $A_i \dots A_j$ is defined as the sum of the weights

$w_i + w_{i+1} \ldots + w_j$. Then, the cost of $W_{i,j}$ can be calculated in $O(1)$ time, with $W_{1,j} - W_{1,i-1}$ if the cost of all prefix subchains are priori known. The $W_{1,k}$ for $1 \leq k \leq N$ can be calculated as a preprocessing by performing a prefix-sum operation over the workload array of modules. Olstad and Manne [39] also expand the requirement to include some more cases. In their definition, it is desired that, cost of single module is calculated in $O(1)$ time and if $W_{i,j}$ is given. then the cost of a subchain by incrementing or decrementing the i or j by one should be calculated in $O(1)$ time. Note that probe-based solutions are not work for this this definition.

4.1.1 Dynamic-Programming Approach

Dynamic programming based approaches exploit the *optimal substructure* of CCP problem. Let $S^k_{i,j}$ represent the bottleneck value of an optimal k-way partitioning of subchain $A_i, A_{i+1} \ldots A_j$, then. $S^p_{1,N}$ is the bottleneck value of desired partition. After this definition. the CCP problem can be formulated by the following recursion:

$$S^k_{1,i} = \min_{k-1 \leq j < i} (\max(S^{k-1}_{1,j}, W_{j+1,i})) \qquad (1)$$

We start with $k = 2$ where $S^1_{1,j}$ is equal to $W_{1,j}$, and the iterations finish at $k = P$. Similarly, for each k, i starts from k and goes up to N. The basic dynamic programming approach leads to an algorithm with complexity $O(N^2P)$.

Choi and Narahari [9] exploit the special structure of CCP and reduce the complexity of the DP algorithm to $O(NP)$. Olstad and Manne [39] also exploit the same structure and propose an $O((N-P)P)$ algorithm.

Here, we are going to explain the solution of Choi and Narahari [9]. The main contribution of Choi and Narahari [9] is to determine all values of $S^k_{1,i}$ for $k \leq i \leq N$ by searching only $O(N)$ values in $O(N)$ time. This is possible, because if the minimum value of $S^k_{1,i-1}$ occurs at index j_1, the minimum value of $S^k_{1,i}$ can not occur at an index j less than j_1. We start j from j_1 for the next iteration. Therefore, the complexity of the algorithm is $O(NP)$. Olstad and Manne [39] also show that at each iteration of k, we do not need the values $S^{k-1}_{1,m}$ for $1 < m < k-1$ and $N - p + k < m < N$. This comes from the fact that, in optimal solution each part contains at least one module ($N \geq P$). Therefore, they reduce the complexity to $O((N-P)P)$.

4.1.2 Probe-Based Approach

The probe-based approach is different from DP, it searches for candidate bottleneck values and reaches an optimal solution. It uses a function, called *probe*, which takes a bottleneck value W and determines whether a partition exists with bottleneck value W', where $W' \leq W$. The probe function loads consecutive processors with consecutive subchains in a greedy manner such that each processor is loaded as much as possible without exceeding W. That is, probe function finds the largest index i_1 such that $W_{1,i_1} \leq W$. Similarly, it finds the largest index i_2 such that $W_{1,i_2} \leq W_{1,i_1} + W$. This process continues until either all modules are assigned, or some modules remain after loading P parts. In the former case, we say that a partition with a bottleneck value no more than W exists. In the latter case, we know that no partition exists with a bottleneck value less than or equal to W. It is clear that this greedy approach will find a solution with bottleneck value no greater than W if there is any. The probe function takes $O(N)$ time with linear search and $O(P \log N)$ with binary search. The binary search version needs a prefix summed workload array. Each $i_s, 1 \leq s \leq P$ can be found in $O(\log N)$ time with binary search, resulting in $O(P \log N)$ total complexity.

The solution of Nicol and O'Hallaron [38] searches $4N$ candidate bottleneck values for an optimal solution with a total complexity $O(NP\log(P))$. Later, Nicol [37] improved the algorithm by searching for $O(P\log(N))$ candidates and reduced the complexity to $O(N + (P \log N)^2)$ where the $O(N)$ complexity comes from the cost of performing an initial prefix sum on workload array. The algorithm uses a binary search and finds the greatest index i_1 such that the call of probe with W_{1,i_1} returns false. Here, we can argue that, either W_{1,i_1+1} is the bottleneck value of an optimal solution, which means that processor 1 is bottleneck processor, or W_{1,i_1} is the cost of processor 1 at optimal solution. At this time we cannot know which is true. So, we save the W_{1,i_1+1} as C_1, and starting from $i_1 + 1$ perform the same operation to find i_2 and C_2. This operation is repeated P times and the minimum of the C_i's $1 \leq i \leq P$, is the bottleneck value. Once the bottleneck value is found the division points for subchains can be found using the greedy approach of probe function.

Despite the theoretical $O(P \log N)$ probe calls, we observe that the number of probe calls can be reduced in practice. We know that the probe function works in a greedy manner, and if it returns false for a candidate bottleneck value L it will return false for every value smaller than L. Similarly. if it returns true for a candidate bottleneck value U it will return true for every value greater than U. In this work, we dynamically reduce the interval $[L. U]$ during searching for candidate bottleneck values and calling the probe function only for a candidate bottlenecks value in reduced interval $[L, U]$. The initial bottleneck value range $[L. U]$ can be set to $[0, W_{tot}]$. This initial range can also be reduced such that L is set to $\max(W_{tot}/P, w_{max})$ and U is set to $\max(W_{tot}, W_{tot}/P + w_{max})$, where w_{max} is the maximum cost among modules. Experimental results show that dynamically reducing the interval with a good initial bound reduces the number of probe calls substantially.

4.2 Decomposition of 2D Domains

Decomposition algorithms presented in this section decompose a 2D domain into P rectangular region. The CCP algorithms are effectively exploited for finding optimal jagged decomposition. Rectilinear decomposition algorithm finds suboptimal solution with an iterative heuristic that utilizes the idea of CCP. The 2D domain is represented as a workload array of size $M \times N$. The workload of a cell C_{ij} is w_{ij} and total workload of the domain is W_{tot}. The processors are designed as a $p \times q$ processor mesh, where $p \times q = P$. The p and q values are chosen such that the resulting processor mesh is as close as to square. Decomposition of the domain is performed such that the cost of maximally loaded processor is minimized. Processor with maximum workload is called bottleneck processor and workload of that processor is called the bottleneck value.

4.2.1 Rectilinear Decomposition

Rectilinear decomposition, divides the rows of workload array into p interval of different sizes and columns into q interval. Grigni and Manne [17] proved that finding the optimal rectilinear decomposition is NP-complete. Nicol [37] proposed an iterative heuristic to find a well balanced rectilinear decomposition. The same

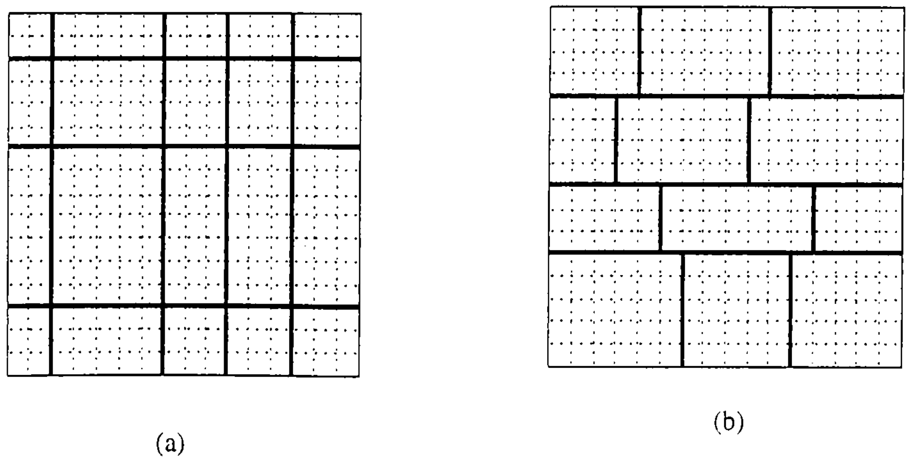


Figure 4.1. Decomposition schemes : (a) rectilinear decomposition, (b) jagged decomposition.

algorithm is also proposed by Manne and Sørevik [33]. The iterative algorithm is based on finding an optimal solution in one dimension given a fixed partition in alternate dimension. The next iteration uses the solution just found in one dimension as a fixed partition and finds optimal solution in the other dimension. This operation is repeated. each time fixing the partition in alternate dimensions. The decomposition problem in one dimension is the adaptation of the CCP. It is proven that the iterations converge very fast to a local optimum solution [37].

Nicol defines the optimal conditional partitioning (OCP) as an optimal partitioning in one dimension while fixing in the other. After defining OCP, the rectilinear decomposition problem becomes iteratively applying it to alternate dimensions. OCP algorithm utilizes CCP algorithms. In OCP $p(q)$ chains are partitioned concurrently at the same indices to minimize the cost of bottleneck subchain in all chains. Here, the chains are the strips found at previous OCP. The complexity of the rectilinear decomposition algorithm will be the number of iterations times the complexity of the OCP.

The DP-based OCP [17] is as follows: The first step of the algorithm is to collapse the p row strip into p chains. After this collapsing, we have p chains of length N for a workload array of size $M \times N$. The algorithm is to apply DP-based chain partitioning solution. This time, the cost $W_{i,j}$ is replaced with the cost

$$\max_{1 \le k \le p} (W_{i,j,k})$$

where $W_{i,j,k}$ is the total cost of subchain $A_i \ldots A_j$ at k^{th} chain/strip. The $W_{i,j,k}$ values can be calculated in $O(p)$ time. Therefore the complexity of OCP is $O(MN + pq(N-q))$ where $O(MN)$ cost comes from the collapsing operation.

The probe-based OCP [37] makes minor modifications to the probe function. The modified probe function, referred to here as probe-conditional, takes the candidate bottleneck value W and determines whether a partition exists for each chain with a bottleneck value $W' \leq W$, with a constraint that all chains are partitioned at the same indices. We find the largest index j_k such that $W_{1,j_k,k} \leq W$ for all k. Then, we take the smallest j_k as l_1. Next, starting from the $l_1 + 1$, we find the largest index j_k such that $W_{1,j_k,k} \leq W + W_{1,l_1}$ for all k. We take the smallest j_k as l_2. This process continues similarly and it returns true if it achieves to load all processors with a load no greater than W. Thus, the complexity of the probe-conditional is $O(pq \log N)$. The searching strategy is similar to the 1D version. We find the largest index i_j such that calling probe-conditional with $W_{i_{j-1}+1,i_j,k}$ returns false for all k, $1 \leq k \leq p$. Then, we find the minimum of $W_{i_{j_1}+1,i_j,k}$ (for all $k = 1 \ldots p$) such that probe-conditional returns true for $W_{i_{j-1}+1,i_j+1,k}$. The number of probe calls is $O(pq \log M)$. Therefore, the cost of optimal conditional partitioning is $O(MN + pq \log N \log M)$ [37]. The $O(MN)$ cost is the cost of collapsing the p strip into p chains.

Note that for both probe-based and DP-based OCP, the dominating cost is the cost of collapsing p strips into p chain. This collapsing requires $O(MN)$ time. Manne and Sørevik [33] propose a scheme to decrease the $O(MN)$ cost to $O(pM)$ by utilizing prefix sum over rows and columns. This requires 2 arrays each of size $M \times N$. One array store the prefix sum of each columns and the other array prefix sum of each row. This two arrays are used to collapse the strips to chains. For row strip [i,j], load of each module of the chain is calculated by $W_{1j,k} - W_{1(i-1),k}$, for $k \leq 1 \leq N$, where $W_{1j,k}$ represents the value at column-wise prefix summed array place at index [i,j]. The collapsing of each strip takes $O(N)$ time resulting $O(pN)$ time to collapse p strip.

In this work, we propose a much more efficient scheme which avoids the collapsing operation before each OCP operation. The proposed scheme requires a single 2D array of size $M \times N$. It exploits the idea of summed area table (SAT) [11]. SAT is used to query the workload of any rectangular region in $O(1)$

time. SAT is created by performing a 2D prefix sum over the workload array. 2D prefix sum is done by performing a 1D prefix sum on each individual row followed by a 1D prefix sum on each column of workload array. Once SAT is created as a preprocessing step, collapsing the strips before each OCP is not necessary any more. This improvement reduces the complexity of DP-based OCP to $O(pq(N-q))$ and probe-based OCP to $O(pq \log M \log N)$.

Nicol [37] proved that the iterative rectilinear algorithm converges. Suppose that we have already found a row partition R_k with a cost W_k at iteration k, Then, after fixing that, we perform OCP on columns and get a column partition C_{k+1} with a cost W_{k+1}. Then we fix the column partition C_{k+1} and perform OCP on rows. Surely, we will at least get the same column partition C_{k+1} with a cost W_k. This proves that the cost function is non-increasing. Nicol states that the iterative refinement algorithm converges in $O(U(N+M))$ iterations, where U is the number of unique bottleneck constraints. However, he also states that, in practice it converges in far fewer iterations, perhaps in $O(\max\{N, M\})$ iterations.

4.2.2 Jagged Decomposition

If we relax the decomposition in one dimension, we get jagged decomposition (also known as Semi generalized block partitioning). Here, rows (columns) are partitioned into $p(q)$ strips and each strip is independently partitioned in alternate dimension. Here, we explain the case, where rows are partitioned into p strip and each row strip is partitioned into q column strip independently.

Manne and Sørevik [33] proposed an algorithm to find optimal jagged decomposition of a workload array. Their algorithm is based on 1D CCP and it tries to minimize the cost of maximally loaded processor by finding the division lines for strips and the division of individual strips in alternate dimension. They perform a p-way chain partitioning on rows. The cost of any subchain in p-way partition, here a subchain is a row strip, is found by applying a q-way partitioning on columns of that strip and taking the cost of partition as the cost of the subchain.

Manne and Sørevik [33] used DP-based CCP. The first algorithm they propose performs the q way partition on columns in $O(q(N-q)(r_i - r_j))$ time where r_i, r_j are the row indices. The complexity of this algorithm is $O(p(M-p)(N+(N-q))) = O(pqM(M-p)(N-q))$ for optimal jagged decomposition.

In the second algorithm, they improve the complexity by a factor of M. They achieve this by collapsing the row strips. In DP-based CCP algorithm queried subchains differ by only one module from the previous queried subchain. Suppose that, we have already have collapsed array of strip $[r_i, r_j]$, during chain partitioning on rows, we will next need the collapsed array of either $[r_i, r_{j+1}]$ or $[r_{i+1}, r_j]$. Both strips can be calculated in $O(N)$ time using the previous collapsed array of $[r_i, r_j]$. Thus each column partition requires $O(N + q(N - q))$ time. Therefore, the complexity of jagged decomposition becomes $O(pq(M - p)(N - q))$. We note that, SAT scheme mentioned earlier can also be used for DP-based jagged decomposition algorithm instead of collapsing the rows at each step.

In this work, we have implemented the probe-based jagged decomposition algorithm. A straightforward implementation leads to an $O((p \log M)^2)(M^2 + (q \log N)^2))$ time algorithm. The improvement by collapsing the rows cannot be applied to probe-based approach, as probe-based approach does not have coherency between successive steps. We utilize the SAT scheme to achieve the complexity $O(MN + (p \log M)^2(q \log N)^2)$.

4.3 Conclusion

We have presented iterative heuristic algorithms for rectilinear decomposition. These algorithms was proposed by Nicol [37] and Manne and Sørevik [33]. We present that the use of SAT scheme decrease the complexity of both DP-based and probe-based iterative algorithms. In the SAT scheme, we perform the $O(MN)$ work only once as a precomputation and avoid the collapsing operation at each iteration.

We have presented the optimal jagged decomposition algorithm proposed by Manne and Sørevik [33]. They have implemented the DP-based solution and decrease the complexity of initial algorithm by collapsing the rows. We present a probe-based algorithm. We show that the complexity of probe-based solution is better than the DP-based solution, only if SAT idea is utilized.

Manne and Sørevik [33] showed optimal jagged decomposition achieves very good load balance. They compare the load balance performance of optimal jagged decomposition with orthogonal recursive bisection, rectilinear decomposition and jagged decomposition based on a heuristic. They state that, optimal jagged decomposition gives good load balance, but it is too time consuming. In our work, we see that probe-based solution with SAT works faster than DP-based algorithm. We believe that, good load-balance performance and decent run-time performance makes the jagged decomposition attractive for many applications including computer graphics and linear algebra.

5. IMAGE-SPACE DECOMPOSITION ALGORITHMS

5.1 Creating the Workload Arrays

This section describes how the workload arrays are created. Each processor creates local workload arrays using local primitives. Then, a global sum operation is performed on local arrays of each processor so that each processor receives the global workload arrays.

5.1.1 1D Arrays

1D arrays are used to find the distribution of primitives over one dimension of the screen region. In this section, we present 1D arrays for y-dimension of the screen region. 1D arrays used for x-dimension of the region are duals of these arrays.

There are two 1D arrays used to find the distribution of primitives in y-dimension. The first array is the *y-dimension local primitive start* (YPS_l) array of size N, where N is the resolution of the screen in y-dimension. The second array is the *y-dimension local primitive end* (YPE_l) array of size N. Each entry of these arrays corresponds a scanline in the region. Each processor updates these arrays using the local primitives via the algorithm given in Fig. 5.1.

After all local bounding boxes are processed, $\mathrm{YPS}_l[j]$ gives the number of local primitives that start at scanline j. Similarly, $\mathrm{YPS}_l[j]$ gives the number of local primitives that end at scanline j. A global sum operation is performed on these two arrays so that each processor receives the global arrays YPS_g and

```
for each local bounding box ($bbox$) $k$ do
  $ymin = bbox[k].ymin; ymax = bbox[k].ymax;$
  $\text{YPS}_l[ymin] = \text{YPS}_l[ymin] + 1;$
  $\text{YPE}_l[ymax] = \text{YPE}_l[ymax] + 1;$
endfor
```

Figure 5.1. Algorithm to update 1D arrays using bounding boxes.

YPE_g, containing the information for all primitives in the scene. Then, prefix-sum operation is performed on each global array to obtain prefix-summed arrays, YPS_p and YPE_p. The value $\text{YPS}_p[j]$ gives the number of primitives that start before scanline j, including the scanline j. $\text{YPE}_p[j]$, on the other hand, gives the number of primitives that end before scanline j, including that scanline. The number of primitives (workload, WL) in a region bounded by scanlines i and j ($> i$) is given by the following equation:

$$WL[i, j] = YPS_p[j] - YPE_p[i-1]. \quad (1)$$

This equation gives the exact number of primitives in a horizontal region bounded by $[i, j]$.

5.1.2 2D Arrays

In algorithms using 2D arrays, a 2D coarse mesh is superimposed on the screen. The mesh cell weights are updated using the distribution of primitives over this coarse mesh. Bounding boxes are projected onto this mesh.

5.1.2.1 Inverse Area Heuristic (IAH) Model

In this scheme, bounding boxes of the local primitives are tallied to mesh cells after the mesh is superimposed on the screen. Some primitives may intersect multiple cells. In order to decrease the errors due to counting such primitives many times, Mueller [36] uses a simple heuristic, referred to here as *inverse area heuristic*. Each primitive increments the weight of each cell it intersects by a value inversely proportional to the number of cells the primitive intersects. In this heuristic, if we assume that there are no shared primitives between screen regions, the sum of the weights of individual cells forming a region gives a value linearly proportional to the exact number of primitives in that region. However, shared primitives still cause errors when calculating the number of primitives in a region. The contribution of a shared primitive between regions is divided among those regions. Thus, the computed workload of a region is less than the actual value. However, it can be expected that such errors are less than counting such primitives multiple times while adding cell weights. Mueller points out that this heuristic gives better results.

Some of the decomposition algorithms presented in this thesis need to query workload of a rectangular region. In order to query the workload in $O(1)$ time workload array is converted into a SAT by performing a *2D prefix sum* over the 2D workload array. The 2D prefix sum is done by performing a 1D prefix sum on each individual row of the mesh followed by a 1D prefix sum on each individual column. After the 2D prefix sum operation, SAT$[x, y]$ gives the workload of the region bounded by corner points $(1, 1)$ and (x, y). The workload of a rectangular region, whose corner points are $(xmin, ymin)$ and $(xmax, ymax)$, can be computed using the following expression:

$$\begin{aligned} WL[(xmin, ymin), (xmax, ymax)] &= SAT[xmax, ymax] \\ &- SAT[xmax, ymin - 1] \\ &- SAT[xmin - 1, ymin] \\ &+ SAT[xmin - 1, ymin - 1]. \quad (2) \end{aligned}$$

5.1.2.2 Exact Model

The inverse area heuristic gives an estimated number of primitives in a region since shared primitives cause errors. In this work, we propose a new method to find the exact number of bounding boxes in a rectangular region efficiently. Our method uses four 2D arrays.

There are several ways of arranging the four 2D arrays to query the exact number of bounding boxes. We present a scheme that requires the minimum number of operations. The first array, which is called STARTXY, is filled by projecting the lower left corners of bounding boxes and incrementing the value of the cell. This lower left corner may be thought as a starting point of the bounding box as it is the closest corner to the $(1, 1)$ point of the screen. After projecting the starting points of bounding boxes, a 2D prefix sum is performed over this array. After this prefix sum operation, STARTXY$[x, y]$ gives the number of bounding boxes intersecting a region bounded by $(1, 1)$ and (x, y) (Fig. 5.2(a)). In other words, it finds the number of bounding boxes that start (or whose lower left corner reside) in that region. Similarly, the second array, ENDXY, is filled by projecting the upper right corner (also called end point) of bounding boxes. This operation is followed by a 2D prefix sum on the array. As seen in Fig. 5.2(b), ENDXY$[x, y]$ gives the number of bounding boxes whose upper left points are in the region bounded by $(1, 1)$ and (x, y). The third array, ENDX, is filled by rasterizing the left side of bounding boxes. The bounding box $[(sx, sy), (ex, ey)]$ contributes to the cells (ex, k), $sy \leq k \leq ey$. After we fill the array, we perform a row-wise prefix sum on each row. The value ENDX$[x, y]$ gives the number of bounding boxes, whose left side intersects the line $[(1, y), (x, y)]$ (Fig.5.2(c)). The fourth array, ENDY is similar to the ENDX array. It is filled by rasterizing the upper side of the bounding boxes. This side is equivalent to the indices (k, ey), $sx \leq k \leq ey$. The rasterization operation is followed by column-wise prefix sum on each individual column of the array. As seen in Fig. 5.2(d), ENDY$[x, y]$ gives the number of bounding boxes whose upper sides intersect the line $[(x, 1), (x, y)]$.

After defining the arrays, STARTXY, ENDXY, ENDX, and ENDY, we can calculate the exact number of primitives (WL) in a region bounded by $[(xmin, ymin); (xmax, ymax)]$ as follows:

$$\begin{aligned} WL = \; & STARTXY[xmax, ymax] \\ & -ENDXY[xmin-1, ymax-1] \\ & -ENDXY[xmax-1, ymin-1] \\ & -ENDX[xmin-1, ymax] \\ & -ENDY[xmax, ymin-1] \\ & +ENDXY[xmin-1, ymin-1] \end{aligned} \tag{3}$$

As seen in Eq.(3), this scheme requires only 4 subtractions and 1 addition operation to query the number of bounding boxes in a rectangular region.

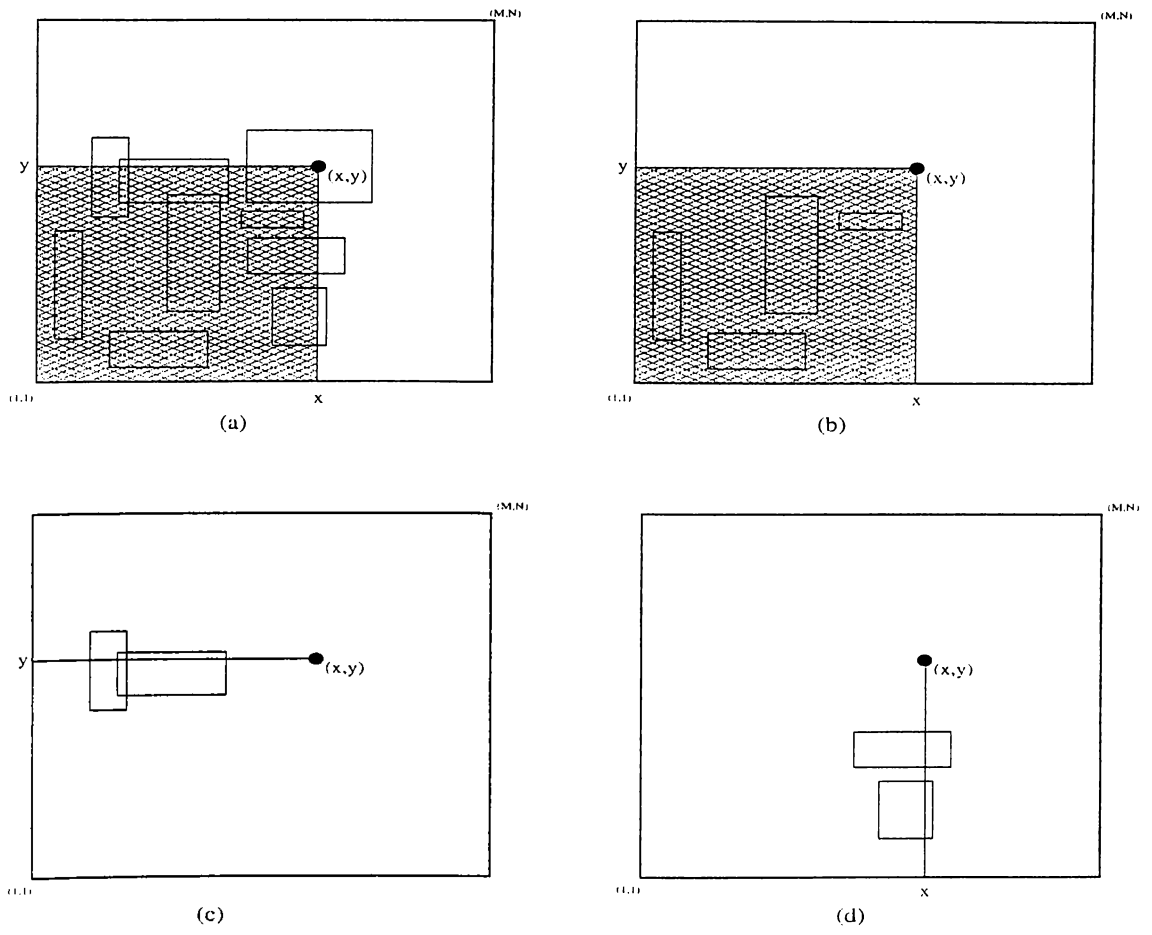


Figure 5.2. Arrays used for the exact model: a) STARTXY b) ENDXY c) ENDX d) ENDY

The correctness of the scheme can easily be verified. We wish to find the exact number of bounding boxes in a region bounded by $(xmin, ymin)$ and $(xmax, ymax)$ as given in Fig.5.3. Each letter $(A - P)$ represents the bounding boxes of the same type. We wish to calculate the number of bounding boxes A through I. The value of STARTXY$[xmax, ymax]$ gives the sum of the number of bounding boxes A through P. The ENDXY$[xmin - 1, ymax - 1]$ gives the number of bounding boxes J, K, L and ENDXY$[xmax - 1, ymin - 1]$ gives the number of L, M, N type bounding boxes. As we subtract L twice we add it with ENDXY$[xmin - 1, ymin - 1]$. Note that we can not omit the bounding boxes of type O and P using arrays STARTXY or ENDXY. Therefore, we use ENDX$[minx - 1, ymax]$ to subtract the P type bounding boxes and ENDY$[xmax, ymin - 1]$ for O type.

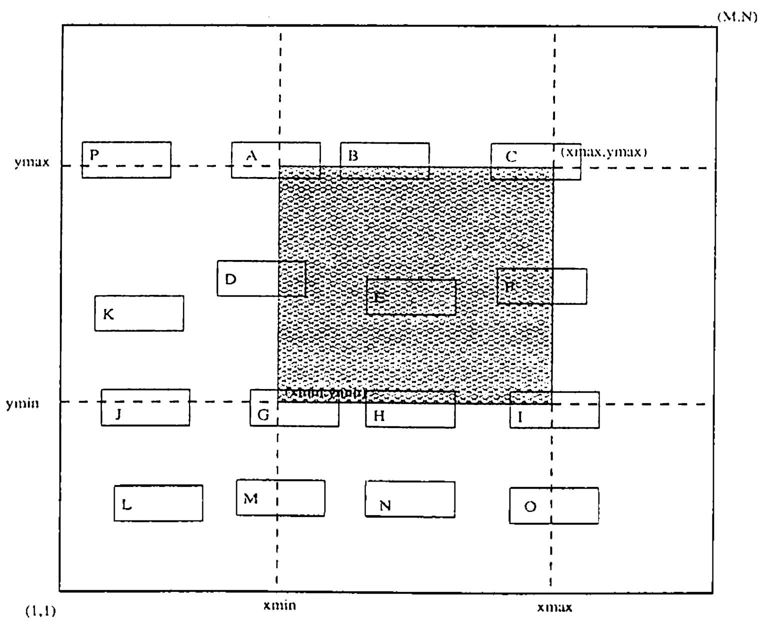


Figure 5.3. Exact Model for calculating number of primitives in a region

5.2 Decomposing the Screen

In this section, we describe decomposition algorithms to divide the screen using the primitive distribution.

5.2.1 1D Decomposition Algorithms

The schemes discussed in this section divide the screen into P horizontal strips. The screen is divided into horizontal strips to preserve the intra-scanline coherency of the rendering algorithm. Each strip consists of consecutive scanlines to preserve the inter-scanline coherency up to some extend and to decrease number of shared primitives by keeping the number of the region boundaries for each processor.

5.2.1.1 Heuristic Horizontal Decomposition (HHD)

HHD scheme is a 1D decomposition scheme using 1D arrays with exact model [27]. An example decomposition of the screen by HHD algorithm on 16 processors is given in Fig. 5.8(a).

The screen is decomposed into regions recursively. In this way, a full binary tree, whose root being the whole screen, is conceptually generated. At each decomposition level k ($k = 1, ..., log_2 P$), a region bounded by scanlines i and j is divided into two regions $[i, m]$ and $[m+1, j]$. The division line m that separates two regions is determined, by checking all possible lines, such that the following expression is minimized.

$$max(WL[i,m], WL[m+1,j]) - \frac{V}{2^k}. \qquad (4)$$

In this expression, function $max(a,b)$ returns the maximum of a and b. The value $WL[i,m]$ gives the workload in the region bounded by scanlines i and m. In our case, workload is equal to the number of primitives in that region. Similarly, $WL[m+1,j]$ gives the workload of the region bounded by scanlines $m+1$ and j. The minus term $V/2^k$ represents the average load at decomposition level k. Here, V is the original number of primitives in the scene. Note that ideal load balance is achieved when each processor is assigned V/P primitives. In this respect, the minus term also represents the perfect load balance condition at each decomposition level. The expression given above also tries to decrease the number of shared primitives since the term $max(WL[i,m], WL[m+1,j])$ will be equal to $V/2^k$ when there are no shared primitives. If there are multiple division lines that minimize Eq. (4), we can choose the division line such that

$$WL[i,m] + WL[m+1,j] \qquad (5)$$

is minimized. In this way, we choose the division line that results in minimum number of shared primitives.

5.2.1.2 Optimal Horizontal Decomposition (OHD)

In the previous section, a heuristic scheme is presented to divide the screen horizontally. In this section, we give an algorithm that divides the screen into P horizontal strips optimally utilizing CCP. This algorithm is a 1D decomposition algorithm using 1D arrays with exact model. An example decomposition of the screen by OHD algorithm on 16 processors is given in Fig. 5.8(a).

In this work, we have implemented the probe-based CCP explained in Section 4.1.2 for decomposition of image-space, since it runs faster. We use the arrays YPS_p and YPE_p to calculate the cost function $W_{i,j}$, which is the number of primitives in the region bounded by scanlines i and j, using Eq. (1).

5.2.2 2D Decomposition Algorithms

In the 1D decomposition algorithms, the atomic task is defined to be a scanline, i.e., scanlines are not divided. Due to this restriction, the scalability of 1D decomposition is limited by the number of scanlines. In addition, the workload at each region is determined by the workload at each scanline. Hence, if there are large differences in the workloads of scanlines, the load imbalance between regions may still be large. The limitations of the 1D decomposition can be eliminated to some extent by using workload distribution with respect to both dimensions of the screen. The schemes to implement this idea are given in the next sections.

5.2.2.1 Rectilinear Decomposition (RD)

This scheme is a 2D decomposition algorithm using 2D arrays with exact model. This scheme divides the x-dimension into p strips and y-dimension into q strips, where $pq = P$. We use an iterative algorithm for rectilinear decomposition, which utilizes the chain partitioning. An example decomposition of the screen by RD algorithm using a 64×64 workload array on 16 processors is given Fig. 5.8(b).

In this work, we implemented the probe-based iterative algorithms explained in Section 4.2.1. We use the exact model to query the number of primitives in a region in $O(1)$ time. The exact model performs the role of SAT and performs each iteration (OCP) in $O((pq \log M)^2)$ time.

5.2.2.2 Jagged Decomposition (JD)

In this section, we present different jagged decomposition algorithms. In jagged decomposition algorithms, processors are organized into a 2D $p \times q$ mesh, thus forming q clusters of p processors in each cluster. In these schemes, the x-dimension/y-dimension of the image space is partitioned into p strips in one dimension and each strip is independently partitioned into q strips in alternate dimension. Here, p represent the number of strips in the main axis of the division, which can be either x- or y-dimension. q represents the number of strips in the alternate dimension.

Heuristic Jagged Decomposition (HJD)

This algorithm is a 2D decomposition algorithm using 1D arrays with exact model [27]. In this scheme, image plane is divided into p horizontal strips as in the HHD scheme. After the decomposition of image plane into p horizontal strips, the workload distribution in x-dimension in each region is calculated using 1D arrays for each region. Then, each region is divided independently into q vertical strips of consecutive vertical lines in x-dimension. An example decomposition of the screen by HJD algorithm on 16 processors is given Fig. 5.8(c).

In this scheme, after p horizontal partitions are found, each processor treats each horizontal strips as a new image plane rotated 90 degrees. Hence, the number of scanlines in each new image plane is equal to the number of vertical scanlines in x-dimension of the global image plane. Each processor uses the bounding boxes of local primitives to find the workload distribution in each horizontal strip. If a bounding box spans two or more horizontal strips, it is divided into segments and workload distribution of each strip is updated according to the corresponding segment. After this step, a global sum operation is performed to obtain the global workload distribution in x-dimension in each strip. Afterwards, each processor finds vertical decomposition in the horizontal strip of the cluster that the processor belongs to.

In order to redistribute primitives, processors need the vertical division information in other clusters so that they can find the rectangular region the bounding box of a local primitive intersects. At the last step, a global expand operation, on the vertical divisions in each cluster, is performed so that each processor has the information about vertical divisions in other clusters.

Optimal Jagged Decomposition (OJD)

The algorithm presented in the previous section is a heuristic. In this section, we present algorithms to find optimal jagged decomposition of the screen for a workload array. The algorithms presented in this section are 2D decomposition algorithms using 2D arrays with inverse area heuristic or exact model. An example decomposition of the screen by OJD algorithms using a 64×64 workload array on 16 processors is given Fig. 5.8(c).

In this work, we use the probe-based optimal jagged partitioning algorithm described in Section 4.2.2. We propose algorithms to partition 2D arrays with inverse area heuristic and exact model. The algorithm, which is based on inverse area heuristic model, finds an optimal jagged decomposition of 2D workload array. However, since inverse area heuristic model is used, this algorithm creates a suboptimal solution for actual primitive distribution. We call this algorithm *optimal jagged decomposition with respect to inverse area heuristic model* (OJD-I). The optimal algorithm uses the exact model with 4 2D arrays described in Section 5.1.2.2. This algorithm generates an optimal partitions for given p and q values and main axis of the division in terms of actual distribution of primitives based on bounding box approximation. We call this algorithm *optimal jagged decomposition with respect to exact model* (OJD-E). Here, exact model performs the role of SAT.

5.2.2.3 Orthogonal Recursive Bisection (ORB)

In this section, we present algorithms based on the orthogonal recursive bisection paradigm. These algorithms divide a region into two subregions of equal workload. Then, they recursively iterate on each of the subregions until the number of regions is equal to the number of processors.

Mesh-based Adaptive Hierarchical Decomposition Scheme (MAHD)

This scheme is based on the work of Mueller [36]. MAHD is a 2D decomposition algorithm using 2D arrays with inverse area heuristic model. Mueller uses SAT to query the number of primitives in a region. An example decomposition of the screen by MAHD algorithm using a 64×64 workload array on 16 processors is given Fig. 5.8(d).

At each decomposition step, longer dimension of the intermediate region is divided. Dividing the longer dimension aims at reducing the perimeter of the final regions as an attempt to reduce the number of shared primitives crossing the region boundaries. In this scheme, resulting regions are rectangular and each region consists of adjacent cells.

After $log_2 P$ steps, each processor is assigned a unique rectangular region of

```
for each local bounding box ($bbox$) $k$ do
    $xmin = bbox[k].xmin; xmax = bbox[k].xmax;$
    $ymin = bbox[k].ymin; ymax = bbox[k].ymax;$
    $\mathrm{YPS}_l[ymin] = \mathrm{YPS}_l[ymin] + 1;$
    $\mathrm{YPE}_l[ymax] = \mathrm{YPE}_l[ymax] + 1;$
    $\mathrm{XPS}_l[xmin] = \mathrm{XPS}_l[xmin] + 1;$
    $\mathrm{XPE}_l[xmax] = \mathrm{XPE}_l[xmax] + 1;$
endfor
```

Figure 5.4. Algorithm to update horizontal and vertical workload arrays.

the screen. A global concatenate operation is performed on these rectangular region information so that each processor receives the region information to be used in the redistribution step.

Orthogonal Recursive Bisection with 1D Arrays (ORB-1D)

This scheme is a 2D decomposition algorithm using 1D arrays with exact model [27]. In this scheme [2, 20, 46], at each decomposition step i ($i = 1, \ldots, log_2 P$), the region assigned to a group of processors is divided into two new regions either vertically or horizontally. An example decomposition of the screen by ORB-1D algorithm on 16 processors is given Fig. 5.8(d).

In this scheme, primitive distribution over two dimensions of the screen is needed to divide the screen horizontally or vertically. This scheme uses 1D arrays for each dimension of the screen. That is, in addition to YPS_l and YPE_l arrays for y-dimension, each processor allocates XPS_l and XPE_l arrays for x-dimension of the screen. Initially, each processor is assigned the whole screen as its local image region. Each processor, then, updates its local copy of the YPS_l, XPS_l, YPE_l, and XPE_l arrays using the local bounding boxes by the algorithm in Fig. 5.4.

The workload distributions in two dimensions are obtained by performing global prefix-sum operations on these arrays to obtain $(\mathrm{X/Y})\mathrm{PS}_p$ and $(\mathrm{X/Y})\mathrm{PE}_p$ arrays for each dimension of the screen. Then, each processor divides its local image region into two regions either horizontally or vertically. The division that achieves better load balance is chosen. Note that for the group of processors that are assigned the same image region, the division will be the same. After the division, half of the processors are assigned one of the regions, and the other half of the processors are assigned the other region. Following the region assignment, bounding boxes crossing the boundary between two regions and intersecting the other region are exchanged between neighbor processors assigned to the other region. Neighborhood between processors can be defined according to various criteria such as interconnection topology of the architecture, labeling of the processors etc. In this work, we chose hypercube labeling for neighborhood definition since it is very simple. Processor k sends the local bounding boxes belonging to other region to the processor whose processor id is $k \oplus (2^{(i-1)})$ at decomposition step i. After this exchange operation, each processor has bounding boxes that project onto its new local image region and the decomposition operation is repeated for new image region.

In order to decompose the new region, we need to update (X/Y)PS$_l$ and (X/Y)PE$_l$ arrays for each dimension of the new region. We update these arrays incrementally using bounding boxes exchanged between processors. Each processor decrements the appropriate positions in (X/Y)PS$_l$ and (X/Y)PE$_l$ arrays for bounding boxes sent to the other processor and increments the appropriate locations in (X/Y)PS$_l$ and (X/Y)PE$_l$ arrays using received bounding boxes.

After $log_2 P$ steps, each processor is assigned a unique rectangular region of the screen. A global expand operation is performed on these rectangular region information so that each processor receives the region information to be used in redistribution step.

Orthogonal Recursive Bisection with Medians-of-Medians (ORBMM)

In this scheme, the screen is divided using orthogonal recursive bisection with medians-of-medians (ORBMM) [41, 45]. Medians-of-medians scheme is used to decrease the load imbalance at each recursive decomposition step by relaxing the division line.

We apply ORBMM on a Cartesian mesh (ORBMM-M) and a quadtree (ORBMM-Q). Example decompositions of the screen by ORBMM-M and ORBMM-Q algorithms using a 64×64 workload array on 16 processors are given in Fig. 5.9(a) and Fig. 5.9(b), respectively. Both algorithms are 2D decomposition algorithms using 2D arrays with inverse area heuristic model. In the quadtree based algorithm, we generate a quadtree from the mesh superimposed on the screen. Each leaf node of the tree is referred to here as *quadnode*. The quadtree is generated in such a way that each quadnode has approximately the same workload. The screen is decomposed at the quadnode boundaries.

In this work, we use two different approaches to generate the quadnodes. The first one work in top-down while second work in bottom-up. The first scheme uses a 2D segment tree [42] to generate the quadtree. The 2D segment tree data structure has hierarchical and recursive structure. The root of the tree covers all screen and at each level a node is divided into four quadrants forming its child nodes. This division is repeated until the size of a leaf node is equal to one mesh cell. In our work, segment tree is implemented as an array in which children of any node are reached with $4 \times node\_index + i$ for $i = 1, ..., 4$, and $node\_index$ of the root is equal to 0. Each processor creates its own segment tree using local primitives. A bounding box contributes to a tree node if and only if the box overlaps with the node and no ancestor of the node is contained by the bounding box. Our structure is an *augmented* data structure such that each non-leaf node stores the contributions to itself and sum of its children. One bounding box can be partitioned into at most $O(N)$ squares for an $N \times N$ mesh. In order to reduce the errors due to counting primitives contributing to multiple nodes, we use a similar method to IAH scheme. We add a value proportional to the ratio of the overlapping area of bounding box on the node to the total area of the bounding box. After all bounding boxes in each processor are inserted into the segment tree, local segment trees are merged to obtain the global segment tree in each processor to generate the quadtree.

Quadtree has a similar structure to the segment tree. Each node is also divided into four quadrants, but this time decomposition ends when workload of a node drops under a specified threshold value. The aim of the quadtree is to divide the space to rectangular subregions such that each subregion has approximately the same amount of workload. Quadtree is generated by traversing the segment tree recursively as follows. When a node in the segment tree has a value under the specified threshold value or it is a leaf node in the segment tree, then it is added to quadnode list, otherwise it is decomposed into four by traversing the child nodes. When segment tree is traversed completely leaf quadnodes of the quadtree are inserted into a linked list structure to be used in the decomposition.

The second scheme is a bottom-up approach since it uses a mesh and joins the mesh cells to form the quadnodes. Mesh cells may be considered as leaf nodes of the segment tree. In this scheme, we do not explicitly create a segment tree. Instead, we think of a virtual segment tree superimposed on the mesh. Each processor tallies its local primitives and updates the corresponding mesh cells. Mesh cells are updated using IAH scheme. After tallying operation. local meshes are merged to obtain the global mesh. Then, the virtual segment tree is traversed in a bottom-up fashion and the quadnodes are inserted into a linked list structure. In this traversal, if the cost of a node is under a specified threshold value, but one of the siblings exceeds the threshold value, we add this node to the quadnode list. In addition, if four sibling nodes do not individually exceed the threshold value, but their sum exceeds. we add four of them to the quadnode list.

The threshold value is selected empirically as a certain percentage of number of primitives. Larger threshold values cause poor load balance. whereas smaller values result in more irregular partitions and more expensive partitioning and redistribution times.

In ORBMM-M, the mesh cells are inserted into linked list structure in a similar way as for quadnodes. This scheme may also be thought as a special case of quadtree scheme, where the threshold value is 0 and each leaf node is inserted to linked list.

After quadnodes have been created, decomposition is performed using ORBMM. ORBMM splits the space into two regions each of which has approximately the same amount of workload. In ORBMM-Q algorithm. each region is recursively divided in alternate dimensions. In ORBMM-M scheme, algorithm recurses on each region by dividing the longer dimension of the region. The midpoints of quadnodes are used to find the median-line. One problem with this scheme is how to assign nodes which are intersected by the median-line. Taking centers of nodes and assigning them according to their centers may cause load imbalance. The medians-of-medians (MM) scheme [41, 45] is used to alleviate this problem. The idea in MM is once bisection has been determined, the border nodes that straddle the median-line are identified and repartitioned. In this phase, we sort the border nodes along the bisection direction and assign nodes to the one side until it has half of the total cost. Then, we assign remaining nodes to the other side.

ORBMM algorithms may generate screen regions that are not rectangular. However, generated regions are restricted to be composed of adjacent mesh cells.

5.2.2.4 Hilbert Curve Based Decomposition (HCD)

HCD scheme is a 2D decomposition algorithm using 2D arrays with inverse area heuristic model. In this scheme, the 2D coarse mesh is traversed in a predetermined way. Then, the mesh cells are assigned to processors such that each processor gets the cells that are consecutive in this traversal. An example decomposition of the screen by HCD algorithm using a 64×64 workload array on 16 processors is given in Fig. 5.9(c).

The curves, which are used to traverse the 2D mesh, belong to the set of *space filling curves* [35, 41]. Among various space filling curves [35], Hilbert curve is widely used in many applications. An example of traversing the 2D mesh with Hilbert curve is illustrated in Fig. 5.5. The numbers on each cell represents the order the mesh cells are traversed. The advantage of Hilbert curve over other space filling curves is that large jumps in the 2D mesh do not occur. Therefore, we may expect that the perimeter of the resulting regions will be less compared to the regions obtained by using other curves.

Our approach to traverse the mesh is based on the work of Singh et al. [45] (referred to as costzones scheme). This approach traverses the 2D segment tree already superimposed on the space in their work. Here, we superimpose a virtual 2D segment tree over the screen. The key idea of this approach is to traverse the child nodes in a predetermined order such that the traversal of leaf nodes forms space filling curve. As the child nodes of a node are the quadrants of a region represented by that node, we have four possible starting point and two possible directions (clockwise or counter-clockwise) for each starting point. An appropriate choice of four out of the eight ordering is needed. The ordering of the children of a cell C depends on the ordering of C's parent's children and the

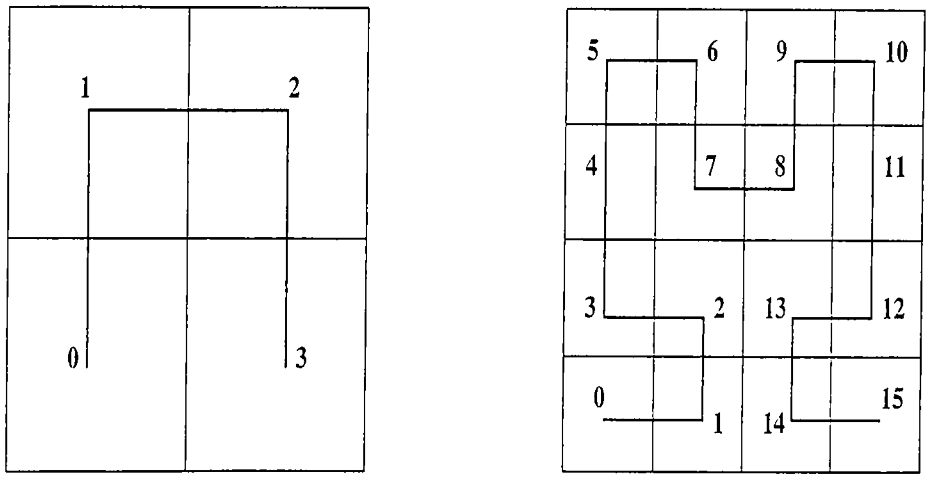


Figure 5.5. Traversing of the 2D mesh with Hilbert curve and mapping of the mesh cells locations into 1D array indices.

position of C in this ordering (Fig. 5.6). In this approach the cells are assigned to the processors during this virtual 2D segment tree traversal. In HCD, resulting regions may be non-rectangular. However, they still consist of adjacent cells on the mesh.

5.2.2.5 Graph Partitioning Based Decomposition (GPD)

This scheme is a 2D decomposition algorithm using 2D arrays with inverse area heuristic model. An example decomposition of the screen by GPD algorithm using a 64×64 workload array on 16 processors is given in Fig. 5.9(d).

This algorithm models the image-space decomposition as a graph partitioning problem [27]. Each cell in the mesh is assumed to be connected to its north, south, west and east neighbors. The vertices of the graph are the mesh cells and conceptual connections between mesh cells form the edges of the graph. The weight of a cell represents the number of primitives intersecting this cell. The weight of the edge between two cells represents the number of primitives crossing the boundary between these two cells. The objective in graph partitioning is to minimize the cutsize among the parts while maintaining the balance among the part sizes. Here, cutsize refers to the weighted summation of cut edges which connect more than one part. The size of a part refers to the weighted summation of the vertices in that part. In our case, balanced partitioning corresponds to maintaining computational load balance during rendering. Minimizing cutsize

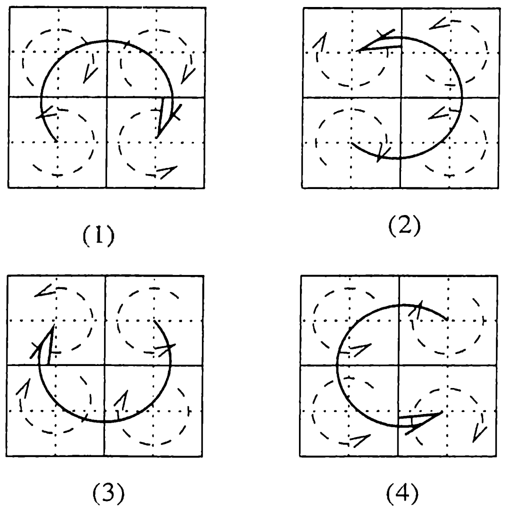


Figure 5.6. Child Ordering of Costzones Scheme.

corresponds to minimizing the number of shared primitives. A state-of-the-art graph partitioning tool, MeTiS [23], is used in this work.

In GPD scheme, each processor tallies local primitives and updates corresponding cell and edge weights. Cell weights are updated using IAH scheme. Edge weight update scheme will be described in the next paragraphs. Each local graph is globally merged to obtain the global graph representing the distribution of all primitives in the scene. The mesh representation of the graph is converted into the representation used by MeTiS. In the original mesh representation, cell and edge weights are real numbers. These values are converted into integers since MeTiS operates on integer vertex and edge weights.

MeTiS uses multilevel partitioning approach consisting of three phases: coarsening, initial partitioning, and refinement. In the coarsening phase, the graph is coarsened down level-by-level to decrease the number of vertices by combining vertices to form new vertices. The coarsest graph is partitioned in the initial partitioning phase and this partitioning is refined in the refinement phase.

In the coarsening phase, various matching schemes can be used in MeTiS to combine appropriate vertices. *Heavy edge matching* scheme is used for coarsening in this work. In the heavy edge matching scheme, at each level of coarsening, an unmatched vertex is combined with another unmatched neighbor vertex such that the weight on the edge between two vertices is maximum. When two vertices are matched and combined to form a new vertex, which is used in the next level of coarsening, the weight of the new vertex is the sum of the weights of the cells forming this new vertex. The weight of the edge between two vertices on the same level is equal to the sum of weights of the edges between vertices forming these two new vertices. Since edge weights are directly added, the weight of the edge between two mesh cells should be updated appropriately during the tallying phase to reflect the number of shared primitives between vertices in the coarse graph. In order to decrease errors caused by primitives shared between more than two cells, we adopt the following scheme to update edge weights. First, we classify shared primitives into three categories: vertical primitives, horizontal primitives, and general primitives. Vertical primitives are the ones that intersect only the cells in a single column. Similarly, horizontal primitives intersect only the cells in a single row. General primitives intersect cells in different rows and columns. The weight of the edge between two cells is incremented by a value proportional to the number of vertical or horizontal primitives intersecting those two cells. On the other hand, the weight of the edge between two cells is incremented by a value inversely proportional to the number cells a primitive intersects for general primitives. In this way, we try to minimize the errors incurred on the edge weight between two vertices formed by cells in neighboring rows or columns.

The graph partitioning approach decomposes the screen in the most general way. Unlike previous partitioning algorithms, noncontiguous sets of cells may be assigned to a processor. In addition, generated regions may be non-rectangular regions.

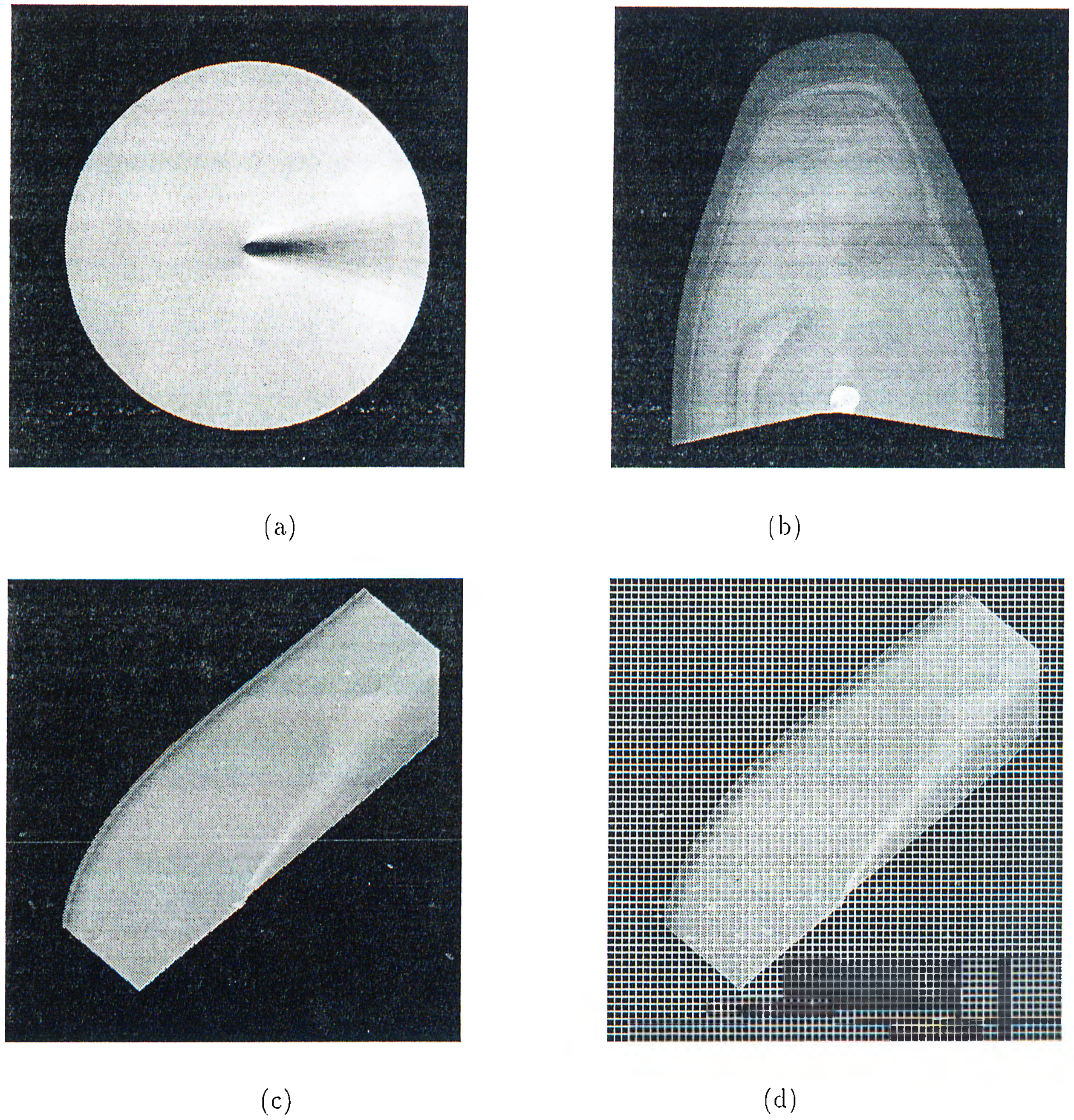


Figure 5.7. (a) Liquid oxygen post image, (b) delta wing image, (c) blunt fin image, and (d) 64×64 coarse mesh superimposed on the screen.

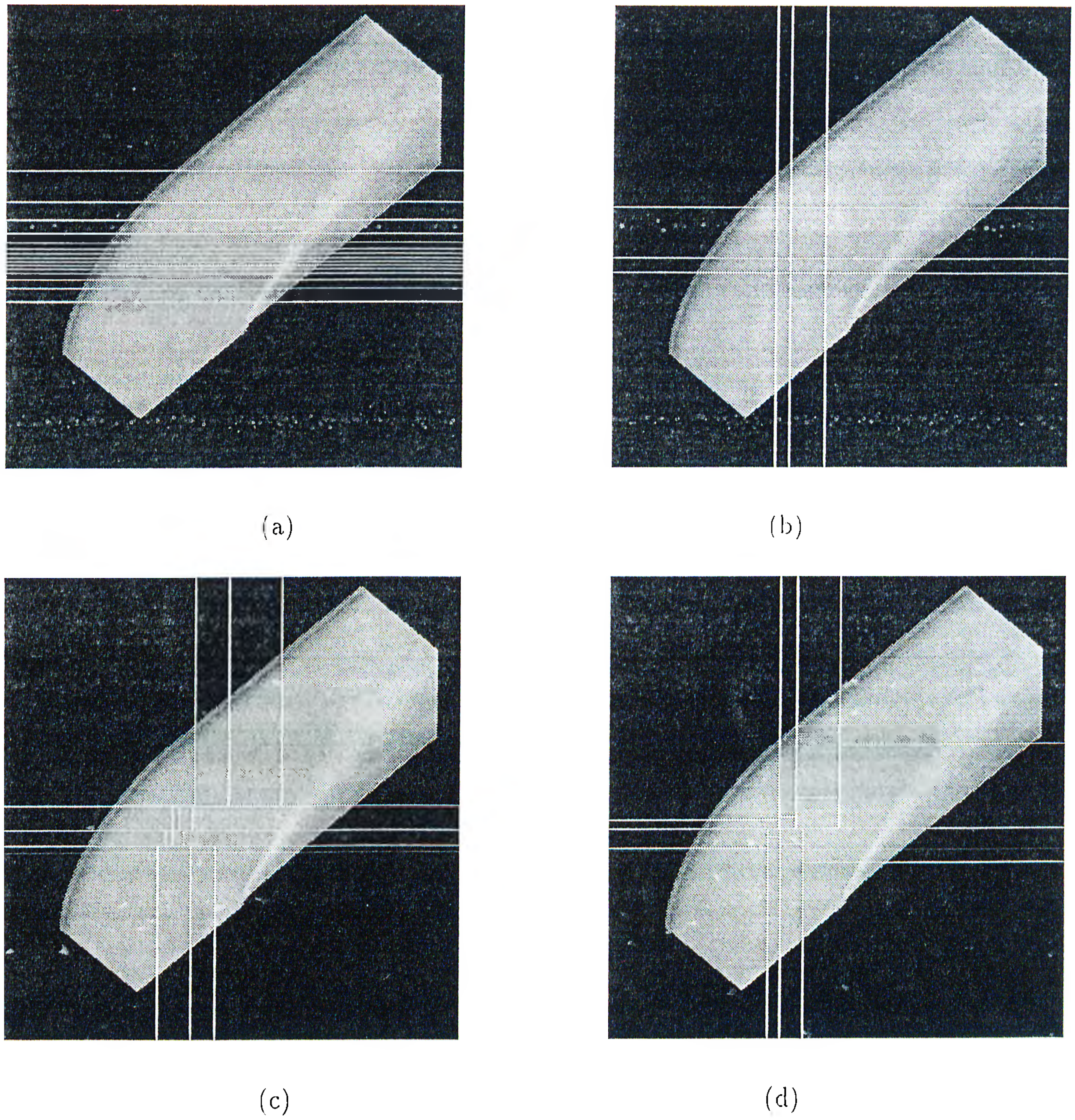
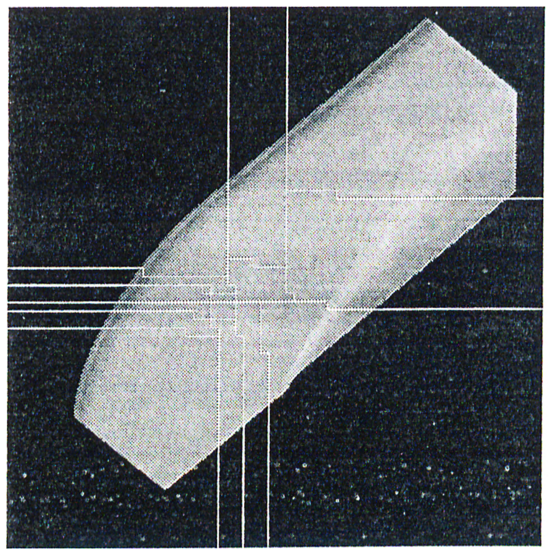
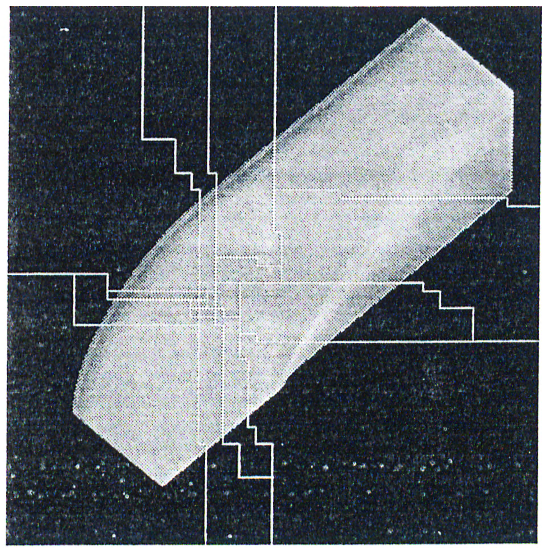


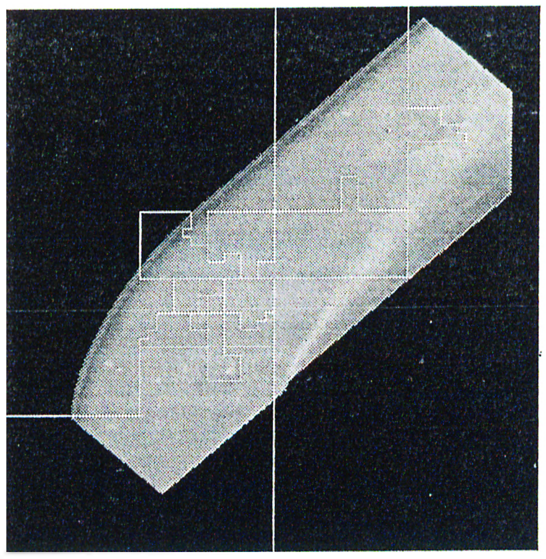
Figure 5.8. Decomposition Algorithms: (a) HHD and OHD algorithms (b) RD algorithm (c) HJD, OJD-I and OJD-E algorithms (d) MAHD and ORB-1D algorithms.



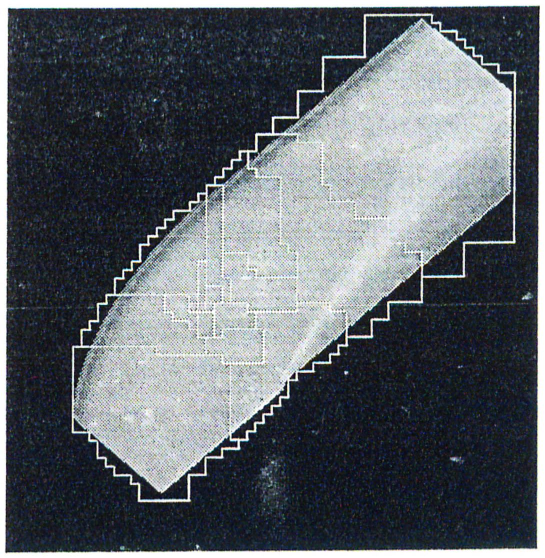
(a)



(b)



(c)



(d)

Figure 5.9. Decomposition Algorithms: (a) ORBMM-M algorithm (b) ORBMM-Q algorithm (c) HCD algorithm (d) GPD algorithm.

6. PRIMITIVE REDISTRIBUTION ALGORITHMS

After decomposition of the screen, each processor needs the primitives overlapping the region it is assigned in order to perform the local rendering calculations. Thus, local primitives in each processors should be redistributed according to the region-to-processor assignment. Assignment of regions to processors constitute the one-to-one mapping problem. In this work, assignment of regions to processors is done using simple schemes. In orthogonal recursive bisection algorithms (MAHD, ORB-1D, ORBMM-M, ORBMM-Q) and HHD algorithm, when a region is decomposed into two regions, lower half of the processors in processor numbering are assigned bottom/left region, whereas processors in the upper half are assigned top/right region. In OHD, OJD-E, OJD-I, and RD algorithms, regions are assigned to processors in row-major or column-major ordering (Fig. 6.2). In HCD algorithm, region-to-processor assignment is done using the traversal of the mesh. The ith region in the traversal is assigned to ith processor of the machine. In GPD algorithm, we use the partitioning vector returned from MeTiS. A region in part i is assigned to processor i. More complicated and better one-to-one mapping algorithms can be found in [5].

Each processor classifies the local primitives according to the regions they overlap. According to the classification, each primitive is stored in the respective send buffer of that region. If a primitive overlaps multiple regions, the primitive is stored in the send buffers of those regions. These buffers are exchanged to complete redistribution of the primitives. In this work, we propose several algorithms for classifying primitives in the redistribution step.

```
for each bounding box ($bbox$) $k$ do
   for each processor $p$ do
      if ($bbox[k]$ intersects with $region[l]$)
         Store the primitive $k$ into the send buffer of processor $l$
   endfor
endfor
```

Figure 6.1. The algorithm to classify the primitives at redistribution step of HHD, OHD, HJD, OJD, RD, ORB-1D, and MAHD algorithms.

6.1 Rectangle Intersection Based Algorithm

The decomposition schemes HHD, OHD, HJD, OJD, RD. ORB-1D, and MAHD divide the screen into rectangular regions. The algorithm to classify the local primitives in these schemes is given in Fig. 6.1. Since the regions are rectangular, the bounding box structure is used to represent regions for each processor. The variable $region[l]$ denotes the region assigned to processor l.

In this scheme, each local bounding box is tested for intersection with each of the regions. If a bounding box intersects a region, corresponding primitive is inserted into the send buffer of processor l.

6.2 Inverse Mapping Based Algorithms

We propose more efficient algorithms for horizontal, rectilinear, and jagged decompositions. The proposed algorithms exploit the regularity of decomposition in horizontal, rectilinear and jagged decompositions. All of these decomposition schemes have a common characteristic: the screen is divided in one of the main axes either in y-dimension or in x-dimension. Division on the main axis creates horizontal or vertical strips. We can consider the horizontal strips as *rows* and vertical strips as *columns*. These rows or columns are then divided in the alternate axis. Thus, it is possible to give numbers to the resulting regions in row-major (if the screen is first divided in y-dimension) or column-major (if the screen is first divided in x-dimension) order. The rectangular region, whose one of the corner points is $(1,1)$ (i.e., lower left corner of the screen), is numbered

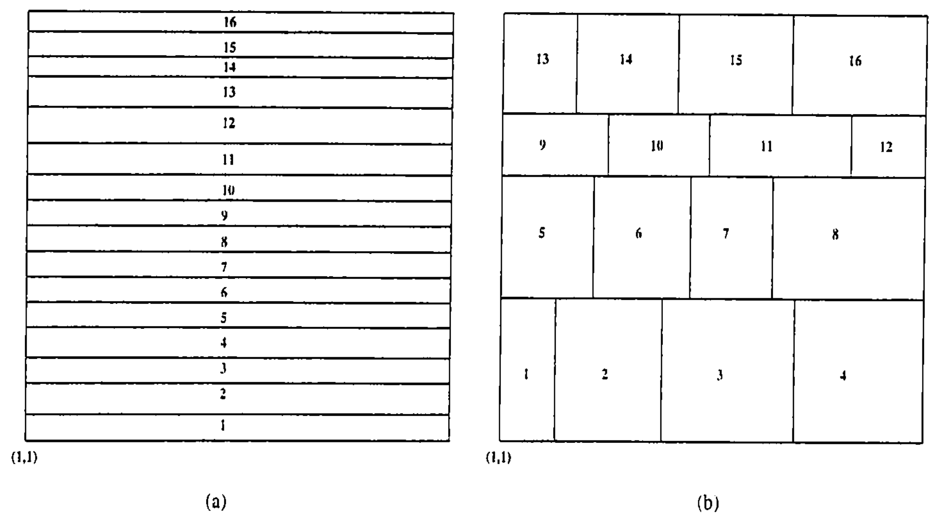


Figure 6.2. Row-major order numbering of regions in (a) horizontal decomposition (b) jagged decomposition for 16 processors.

as 1. Then, numbering of the regions continues in row-major or column-major order. An example numbering of regions is given for a horizontal and a jagged decomposition in Fig. 6.2.

After the screen is divided, we create an array, called region-processor assignment (RPA) arrays, of size P. This array stores the assignment of regions to processors. If a rectangular region i is assigned to processor l, number l is stored into the index i of RPA.

In 1D decomposition schemes, we use an array, called inverse mapping (IM) array, of size N, where N is the resolution of screen in y-dimension. This array represents the assignment of scanlines to processors. That is, if a horizontal strip is assigned a number k in row-major ordering, then indices of IM corresponding to the scanlines in that region are filled with the number k. After IM is filled in this way, primitives are classified with the algorithm given in Fig. 6.3. This algorithm first finds the regions n and m, which include the scanlines corresponding to the end points $ymin$ and $ymax$ of the bounding box, respectively. Then, because of the regularity of horizontal decomposition and row-major ordering, it is enough to insert the primitive associated with the bounding box into the send buffer of processors, which are assigned regions from n to m, including n and m. The classification algorithm given in Fig. 6.3 is expected to be faster than the scheme in Fig. 6.1, because only simple array lookups are necessary to classify

```
for each bounding box ($bbox$) $k$ do
    $ymin = bbox[k].ymin$
    $ymax = bbox[k].ymax$
    for $j = IM[ymin]$ to $IM[ymax]$ do
        Store the primitive $k$ into the send buffer of processor $RPA[j]$
    endfor
endfor
```

Figure 6.3. Classification of primitives in horizontal decomposition scheme using inverse mapping array.

the primitives. If a primitive intersects only one region, only two array lookups are enough to classify the primitive independent of the number of processors.

For rectilinear decomposition scheme, two IM arrays are necessary, one for horizontal strips and one for the vertical strips. The jagged decomposition requires $(p + 1)$ IM arrays if there are p horizontal strips (assuming that screen is first divided in y-dimension, similar is true if screen is first divided in x-dimension). One array (IM^0) is needed for p horizontal strips and p arrays (IM^j, $j = 1, ..., p$) are needed for the partitions in each strip. First, each horizontal strip is given a number in row-major ordering and IM^0 is filled with numbers to reflect this row-major ordering of strips. The strip, whose lower left corner is $(1, 1)$ is given the number 1. Then, the array IM^j for the horizontal strip j is filled with the numbers of the rectangular regions in that strip using the numbering of the regions in row-major ordering in the jagged partition (Fig. 6.2(b)). The algorithm to classify the primitives in jagged partition is given in Fig. 6.4.

6.3 2D Mesh Based Algorithm

Resulting regions in HCD, ORBMM-Q, ORBMM-M, and GPD algorithms may be non-rectangular regions. Furthermore, regions may consist of disconnected mesh cells in GPD algorithm. Therefore, the intersection test of the bounding box with the screen regions to classify the primitives will be more complicated for these algorithms. Instead, a different classification scheme is used in these decomposition algorithms. After the decomposition of the screen, each mesh cell

```
for each bounding box ($bbox$) $k$ do
   $ymin = bbox[k].ymin$
   $ymax = bbox[k].ymax$
   $xmin = bbox[k].xmin$
   $xmax = bbox[k].xmax$
   for $i = IM^0[ymin]$ to $IM^0[ymax]$ do
      for $j = IM^i[xmin]$ to $IM^i[xmax]$ do
         Store the primitive $k$ into the send buffer of processor $RPA[j]$
```

Figure 6.4. Classification of primitives in jagged decomposition using inverse mapping arrays.

```
for each bounding box ($bbox$) $k$ do
   for each mesh cell $c$ the $bbox[k]$ covers do
      $l$ = mark of the cell $c$
      if ($stored[l] < k$) then
         $stored[l] = k$
         Store the primitive $k$ into the send buffer of processor $l$
   endfor
endfor
```

Figure 6.5. The algorithm to classify primitives in HCD, ORBMM-Q, ORBMM-M, and GPD algorithms.

is marked with the processor number whose screen region covers this particular cell. Note that each cell will be marked with a unique processor number. At the redistribution step, primitives are tallied to mesh cells as in the decomposition step. During tallying of a primitive, the primitive is stored into the respective send buffers according to the marks of the cells the primitive covers. The algorithm to classify the primitives is given in Fig. 6.5. The *stored* array (of size P) is used in the algorithm to prevent storing a primitive into the same send buffer multiple times. Initially, each entry of the array is set to -1.

7. EXPERIMENTAL RESULTS

The algorithms presented in this thesis are implemented on a Parsytec's CC-24 system, installed in our department. Embedded Parix (EPX) [40] and PVM 3.3 [15, 16] libraries were used for message passing. Embedded Parix is the native message passing library of Parsytec. For the sake of portability, we implement the algorithms using PVM, but experimental results show that the programs with PVM run 10 to 25 percent slower than the EPX version. The experimental results presented in this paper are the results of the EPX version. The algorithms are implemented using the C language.

Experiments are done using three data sets called *blunt fin*, *delta wing* and *liquid oxygen post* data set. These data sets are used by many researchers in the volume rendering field. They are structured curvilinear data sets. These data sets are converted first into tetrahedrals [14, 44], by dividing each cell into five tetrahedrals, then into a set of distinct triangles. Each triangle in the data set represents a face of a tetrahedral. The blunt fin contains 381548 triangles, delta wing contains 2032084 triangles and liquid oxygen post contains 1040588 triangles after conversion. All results presented in this section are the averages of results for three data sets obtained for six different viewing locations for each data set for the screen resolution of 512×512.

In the first part, we use the number of primitives in each processor to measure percent load imbalance and to measure percent increase in the total number of primitives after decomposition. The percent load imbalance values are calculated as $100 * (Max - Average)/Average$. Here, Max is the maximum of the number of primitives in each processor after decomposition. $Average$ is the average number of primitives and is calculated by dividing the number of primitives in the scene before redistribution by the number of processors. The execution times of the

GPD	: Graph Partitioning Based Decomposition
HCD	: Hilbert Curve based Decomposition
HHD	: Heuristic Horizontal Decomposition
OHD	: Optimal Horizontal Decomposition
RD	: Rectilinear Decomposition
HJD	: Heuristic Jagged Decomposition
OJD-I	: Optimal Jagged Decomposition using Inverse area heuristic for workload array,
OJD-E	: Optimal Jagged Decomposition using Exact model for workload array,
MAHD	: Mesh based Adaptive Hierarchical Decomposition,
ORB-1D	: Orthogonal Recursive Bisection with 1D arrays
ORBMM-Q	: Orthogonal Recursive Bisection with Medians of Medians on Quadtree
ORBMM-M	: Orthogonal Recursive Bisection with Medians of Medians on Cartesian Mesh

Figure 7.1. The abbreviations used for the decomposition algorithms.

algorithms are also a comparison metric. The execution time of each algorithm is the sum of the time for pre-transformation step, execution time of the division and redistribution time. The execution times are given for 2, 4, 8 and 16 processors. We also take 32, 64 and 128 processor results for percent load imbalance and percent increase metrics. The abbreviations used for the decomposition algorithms are displayed in Figure 7.1.

In ORBMM-Q, threshold value is taken as 1/20000 of total number of primitives. In the GPD scheme, heavy edge matching is used as the matching algorithm, the number of vertices the graph should be coarsened down in the coarsening phase is taken as 100; Boundary Kernighan-Lin scheme is selected as the refinement algorithm. These algorithms and values were chosen based on the observations in [23] and our tests.

As explained, jagged decomposition algorithms, HJD. OJD-I, OJD-E, and RD are found to yield better results if p and q are chosen such that the resulting processor mesh is as close to square as possible. In the OJD-E scheme, we perform the decomposition twice for each main axis of division. when $p = q = \sqrt{P}$. If $p \neq q$, we perform two more decompositions by interchanging the p and q values for each main axis of division. Hence, our OJD-E scheme performs 2 or 4 decompositions and chooses the one giving the best load balance. In a similar way, for the RD scheme, we perform two decompositions to choose the values for p and q when $p \neq q$.

The algorithms HHD, OHD, HJD and ORB-1D use 1D arrays for decomposition. In these algorithms, we use the highest screen resolution. The decomposition algorithms MAHD, HCD, ORBMM, OJD, RD and GPD use 2D arrays as a 2D mesh superimposed on the screen for decomposition. The mesh resolution affects the performance of these algorithms. The experimental results for these algorithms are taken for coarse mesh resolutions of 32×32, 64×64, 128×128, 256×256, and 512×512.

Figure 7.2 represents the load balancing performance of the algorithms with varying the mesh resolution on 16 processors. As the schemes HJD, HHD, OHD and ORB-1D perform decomposition at only the highest resolution, we give their results only for mesh resolution of 512×512. As seen from Figure 7.2, OJD-E, OJD-I, RD and MAHD achieve the best load balancing performance at the maximum coarse mesh resolution (of 512×512). This is an expected result for OJD-E, since it finds an global optimum for jagged decomposition and increasing the mesh resolution increases the search space. For the algorithms that use inverse area heuristic, increasing the mesh resolution also increases the search space. However, in these algorithms, primitives that overlap multiple cells incur errors. Increasing the mesh resolution is likely to increase such errors due to inverse area heuristic. Thus, beyond some mesh resolution, these errors may consume the gain due to increasing the search space. We observe that ORBMM-Q, ORBMM-M, GPD and HCD sometime achieve their best performances at mesh resolutions less than 512×512, whereas OJD-I and MAHD achieve their best results always at 512×512. Based on these observations, we hypothesize that errors due to inverse area heuristic have more affect on the algorithms that generate non-rectangular regions. The ORBMM-Q scheme always gives the best result at a mesh resolution of 256×256. For the GPD scheme the best mesh resolution increases as the processor number increases. For P = 2, 4, 8 it gives the best result at mesh resolution of 128×128, for P=16 and 32 it gives the best result at mesh resolution of 256×256 and finally when P is equal to 64 or 128 it gives best load balance at the highest resolution of 512×512. For HCD, it achieves its best at 128×128 when P = 16, 32, at 256×256 when P = 4 and 8, and at 512×512 when P = 2, 64, 128. The ORBMM-M scheme achieves its best at mesh resolution of 256×256 when P is 32 and 64, and at 512×512 for the others.

Figure 7.3 and Table 7.1 illustrate the percent load imbalance behavior of algorithms as the number of processors varies. The load imbalance increases with increasing number of processors. This is expected since we divide the same area among more processors (or smaller area for each new division) and the division is performed at discrete space. Among the algorithms, OJD-E gives the best result. The performances of ORB-1D and HJD are comparable with that of OJD-E. The common characteristics of these three algorithms are that they perform the division in two dimension with a straight line and use the exact number of primitives in a region for decomposition. The ORB-1D divides the screen recursively. It is likely that the load imbalance at a division step propagates and increases at further steps. On the other hand, OJD-E finds global optimum for jagged partition for a given processor number. Therefore, the difference in the load balance performance between these two algorithms increases with increasing number of processors. The performances of HHD and OHD are very bad. This is due to the fact that they try to divide only one dimension and the search space of the algorithms is very restricted. As the number of processors increases, they become worse, due to the poor scalability of 1D decomposition. Among these two, OHD overperforms the HHD as expected, since it finds the global optimum solution. Although RD uses exact number of primitives for decomposition, it does not give good load imbalance performance. This is due to the fact that the iterative algorithm generally converges to a local optimum before reaching the global optimum solution. The resulting partition generally is not good. Nicol [37] states that starting with many randomly chosen initial partitions and then taking the one giving the best result increases the performance of RD. Performing many initial decompositions increases the possibility to reach a global optimal solution. The algorithms which utilize the inverse area heuristic are in the middle for load imbalance performance. For smaller P values, both ORBMM schemes are better. Among two ORBMM schemes, ORBMM-M is better than ORBMM-Q when $P = 4$. But as the number of processors increases, the quadtree based approach becomes better. This is because of the fact that the errors incurred due to inverse area heuristic is less with larger quadnodes. As the number of processors increases, the performance of GPD becomes the best among the ones utilizing the inverse area heuristic. The GPD scheme divides the screen in a more general way and the error propagation at successive steps does not exist for this scheme. The experimental results show that HCD scheme is not suitable for screen decomposition, as the resulting load imbalance is higher than other algorithms. Except for the HCD, our schemes using inverse area heuristic achieve better load balance than MAHD. Although the ORBMM schemes divides the screen in a similar way to MAHD, the medians-of-medians heuristic decreases the load imbalance at each decomposition step for ORBMM schemes. Hence, the error propagation of ORBMM schemes is less than that of MAHD.

Figure 7.4 illustrates the percent increase in the total number of primitives after redistribution with varying the mesh resolution at P=16. For HCD, as the mesh resolution increases, the number of shared primitives increases. This is expected, as the mesh resolution increases, the total perimeter of the resulting regions generally increases. For the other schemes, we do not observe any regularity between load balance and mesh resolution change. We hypothesize that this can be attributed to the fact that our main concern was the load balance. We try to reduce the cost of the maximally loaded processor while trying to reduce the shared primitives implicitly.

Figure 7.5 and Table 7.1 show the percent increase in the number of primitives after redistribution as the number of processors varies. The percent increase in number of primitives also represents the number of shared primitives. The number of shared primitives increase with increasing number of processors as expected. The values presented here are for the mesh resolution, at which the respective algorithm achieves its best load balance. The perimeter of the resulting partitions affects the percent increase in the number of primitives. The resulting perimeter of 1D partitions, HHD and OHD, are large by their nature. So, the percent increase is higher. Similarly, the resulting perimeter of HCD partitions are also high, resulting in more shared primitives. Among the algorithms, only the GPD scheme explicitly tries to decrease the number of shared primitives. Note that graph decomposition aims to reduce the edge cut (which represents the shared primitives) while maintaining the balance among the parts. The algorithms OJD-I, OJD-E, ORBMM-M, ORBMM-Q, MAHD, HJD and RD implicitly try to reduce the number of shared primitives by reducing the total perimeter of the resulting partition. ORBMM-Q reduces the perimeter by alternating the division axis for decomposition. MAHD and ORBMM-M perform the bisection each time along a longer dimension. OJD-I, OJD-E, HJD and RD select the processor mesh as close as to square, so that the total perimeter of the partition is smaller. Among these algorithms ORB-1D gives the best performance. The performance of RD is comparable to that of ORB-1D. Although the load imbalance is high with RD, the number of shared primitives is low. We hypothesize that the regular division with RD has effect on this result. The regions generated with RD has at most four neighbors. However, in the other schemes the number of neighbors may be higher. More neighbors increase the possibility that a primitive crosses multiple region. The performance of GPD is best among the inverse area heuristic based schemes as it reduces the number of shared primitives explicitly.

The execution times of algorithms with varying mesh resolution are given in Fig. 7.6. The execution times increase with increasing mesh resolution as expected. We observe that the biggest time change in execution time is seen between mesh resolutions of 256×256 and 512×512. The screen space bounding boxes of the primitives in our data sets are small. Most of them overlap few mesh cells, generally one or two cells, for small mesh resolutions. However, at the highest mesh resolution of 512×512, most of the primitives overlap multiple cells. We observe that the algorithms that use mesh structure for redistribution, are more sensitive to mesh resolution change. This is because of the fact that classifying primitives in that way is similar to tallying primitives for decomposition.

Figure 7.7 presents the execution times of the algorithms with varying the number of processors. In this figure, the execution times of GPD, ORBMM-Q and HCD are given at mesh resolution of 256×256 and others at 512×512. We choose the mesh resolution where an algorithm generally gives the best load balance. The execution of each algorithm decreases as the number of processors increases. However, we observe a saturation in the speedup after 8 processors.

Table 7.2 illustrates the dissection of the execution times of the algorithms for the processor numbers 2, 4, 8 and 16. The bounding box creation time is the same for all decomposition algorithms on the same number of processors. It is inversely proportional to the number of processors as the primitives are distributed evenly among the processors before the division step. The decomposition and redistribution times of the algorithms are for the mesh resolution of 512×512. The decomposition time of the algorithms can also be further divided into four parts as memory allocation, filling the workload arrays, performing a global sum over this workload array, and decomposition operation. Table 7.3 illustrates the dissection of the execution times of the algorithms ORBMM-M and OJD-E with varying the mesh resolution and the number of processors for single viewing location of delta wing data. The memory allocation part is dependent on the size of the workload array used in decomposition and it is independent of the number of processors. HHD and OHD fill two 1D arrays and perform global sum on these arrays. ORB-1D fills four 1D arrays and performs global sum. HJD fills $p+1$ such arrays and performs global sum. The algorithms GPD, ORBMM-M, ORBMM-Q, HCD, OJD-I and MAHD (which use inverse area heuristic) tally the primitives into mesh of given resolution and then perform the global sum operation over this 2D array. The algorithms OJD-E and RD (which utilize the exact model) need to fill the four 2D array and perform global sum operation over these arrays. Filling the workload array(s) depends on the number of primitives that remains after pre-transformation. Although the primitives are distributed evenly among the processors, there may be load imbalance after pre-transformation since some of the primitives are eliminated. These primitives are the ones that no rays shot for the intersection. The eliminated primitives at pre-transformation depends on the viewing parameters. However, for the algorithms that use 2D mesh structure, increasing the mesh resolution increases the tallying time as expected. Among these algorithms, algorithms utilizing inverse area heuristic spend more time to fill workload array(s). This is because of the fact that time to tally a primitive is proportional to the number of cells the primitive overlaps. We observe that the tallying time (filling workload arrays) is dominant in the overall execution time for GPD, ORBMM-M, ORBMM-Q, OJD-I, MAHD and HCD. Tallying time takes less time for the algorithms using exact model. Although these algorithms use four 2D arrays, the time to tally the primitives is not proportional to the area of the primitive. In these algorithms, filling *STARTXY* and *ENDXY* only requires updates to the single mesh cells. For *ENDX* and *ENDY* the mesh tallying time depends on the height and the width of the bounding box. Concurrent communication volume overhead due to the global sum operation is directly proportional to the mesh resolution used. It increases with increasing number of processors with a factor of $(P-1)/P$ [28]. The communication time becomes dominant in division time with increasing number of processors. The global sum operation takes more time in the algorithms that use exact model, as they have to perform global sum operation on four workload arrays. We observe that the global sum operation is dominant for OJD-E and RD. The time for the decomposition operation increases with increasing number of processors as expected, since more divisions are performed with increasing number of processors. It also increases with increasing mesh resolution since it increases the search space. Among the decomposition algorithms, GPD scheme also has an extra overhead of converting the mesh representation into graph representation of the graph partitioning tool MeTiS.

The execution time of the redistribution step decreases with increasing number of processors (Table 7.4(a)). This is expected since less number of primitives are classified at each processors. This decrease is less for the algorithms MAHD, ORB-1D, HHD and HJD, which use intersection test between rectangular screen region and bounding boxes. This is due to the fact that as the number of processors increases, the number of intersection tests for a primitive increases proportionally. The redistribution times for algorithms, ORBMM-M, ORBMM-Q, GPD and HCD, which use mesh based classification, increases with increasing mesh resolution. The other algorithms are not affected from mesh resolution change (Table 7.4(b)).

Figures 7.8–7.10 illustrate average speedup values of the parallel rendering algorithm on Parsytec CC system. Figure 7.8 illustrates speedup for parallel rendering phase when only the number of triangles is used to approximate workload in a region. In this case, maximum speedup obtained is 5.93 on 16 processors. Figure 7.9 illustrates speedup for the rendering phase when spans and pixels are incorporated into the workload metric. In this case, speedup increases to 11.87 on 16 processors, which is more than double the speedup when only the number of triangles is considered. Figure 7.10 illustrates the speedup values when execution times of decomposition and redistribution algorithms are included in the running times. Comparison of Figures 7.9 and 7.10 shows that the proposed decomposition and redistribution algorithms do not introduce substantial performance degradation in the overall parallel algorithm. For example, the maximum speedup on 16 processors slightly reduces from 11.87 to 10.69.

As seen in Figures 7.10, best speedup values are achieved by the 1D horizontal decomposition (HD) algorithms. This is an unexpected result since HD algorithm is the most restricted algorithm in terms of search space among other algorithms. However, this algorithm has an advantage over the other algorithms for the volume rendering algorithm chosen in this work. It only disturbs *inter-scanline* coherency. It does not disturb *intra-scanline* coherency since screen is not divided vertically. Hence, HD incurs overheads due to inter-scanline coherency in step 4 of the parallel algorithm. However, 2D decomposition algorithms disturb both type of coherence, thus incurring more overhead in step 4 of the parallel algorithm. In addition, bounding box approximation used for spans and pixels is likely to introduce more errors when screen is divided horizontally and vertically than it is divided only horizontally. For example, the number of spans in a region can be calculated more precisely when only horizontal division lines are allowed. However, when vertical divisions are also allowed, bounding box approximation for the number of spans in a region introduces errors. In spite of these findings, 2D decomposition algorithms are expected to yield better parallel performance for larger number of processors due to their better scalability.

The speedup values are not very close to linear. One of the reasons for this deviation from linear speedup is the bounding box approximation for triangles. The number of spans and pixels generated due to a triangle are calculated erroneously. Thus, the workload in a region calculated using bounding boxes does not truly reflect the actual workload. The second reason is that determining relative workloads of a triangle, a span and a pixel (i.e., constants a, b, and c in Eq. (1)) is not easy. These values should be determined experimentally and the operations involving triangles, spans and pixels are not separated by solid boundaries. It is difficult to separate operations exactly related to a triangle, a span, and a pixel in the implementation.

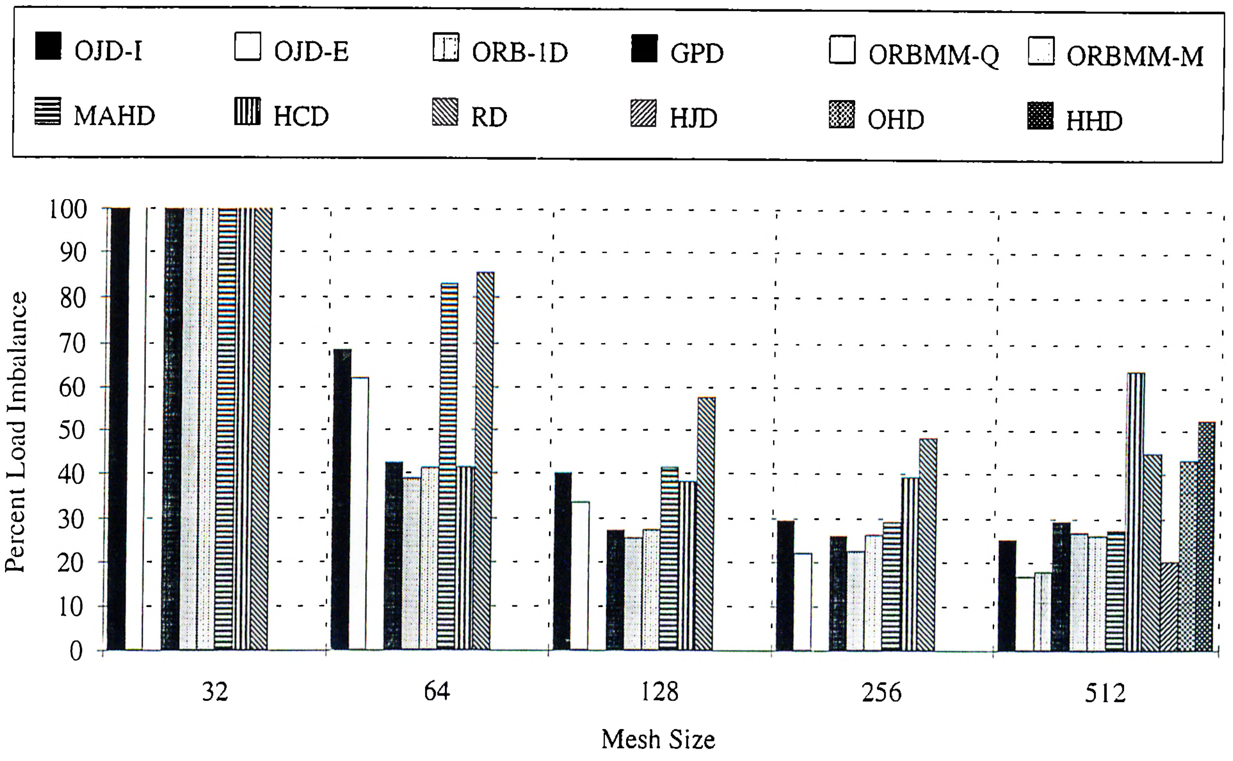


Figure 7.2. Effect of mesh resolution on the load balancing performance.

Table 7.1. Percent load imbalance (L) and percent increase (I) in the number of primitives for different number of processors.

P	2		4		8		16		32		64		128	
Algorithm	L %	I %	L %	I %	L %	I %	L %	I %	L %	I %	L %	I %	L %	I %
OJD-I	2.8	1.8	6.3	3.9	14.0	7.4	25	13	47	20	78	31	153	48
OJD-E	2.0	1.2	5.2	3.6	9.5	6.7	17	12	26	18	42	29	66	456
ORB-1D	2.0	1.2	5.0	3.4	10.1	6.4	18	11	31	17	53	26	102	40
GPD	2.5	1.5	6.8	3.9	14.0	7.1	26	12	40	19	71	29	119	45
ORBMM-Q	1.9	1.8	5.3	4.2	13.2	8.5	23	13	44	22	74	33	138	53
ORBMM-M	2.2	2.1	4.8	3.8	14.0	8.3	26	14	48	23	88	35	149	54
MAHD	3.4	2.1	6.4	4.0	14.7	8.4	27	14	52	22	92	34	155	52
HCD	5.1	4.8	11.3	8.8	21.7	13.9	38	21	64	31	114	48	207	70
RD	2.4	1.5	10.3	3.6	21.8	6.8	45	11	66	18	108	26	149	41
HJD	3.2	2.0	5.8	3.6	12.9	8.0	20	12	36	19	61	28	125	44
OHD	3.2	2.0	9.1	6.2	20.1	13.9	43	30	94	59	216	104	516	120
HHD	3.2	2.0	9.6	6.1	22.5	13.8	52	30	108	60	288	122	675	245

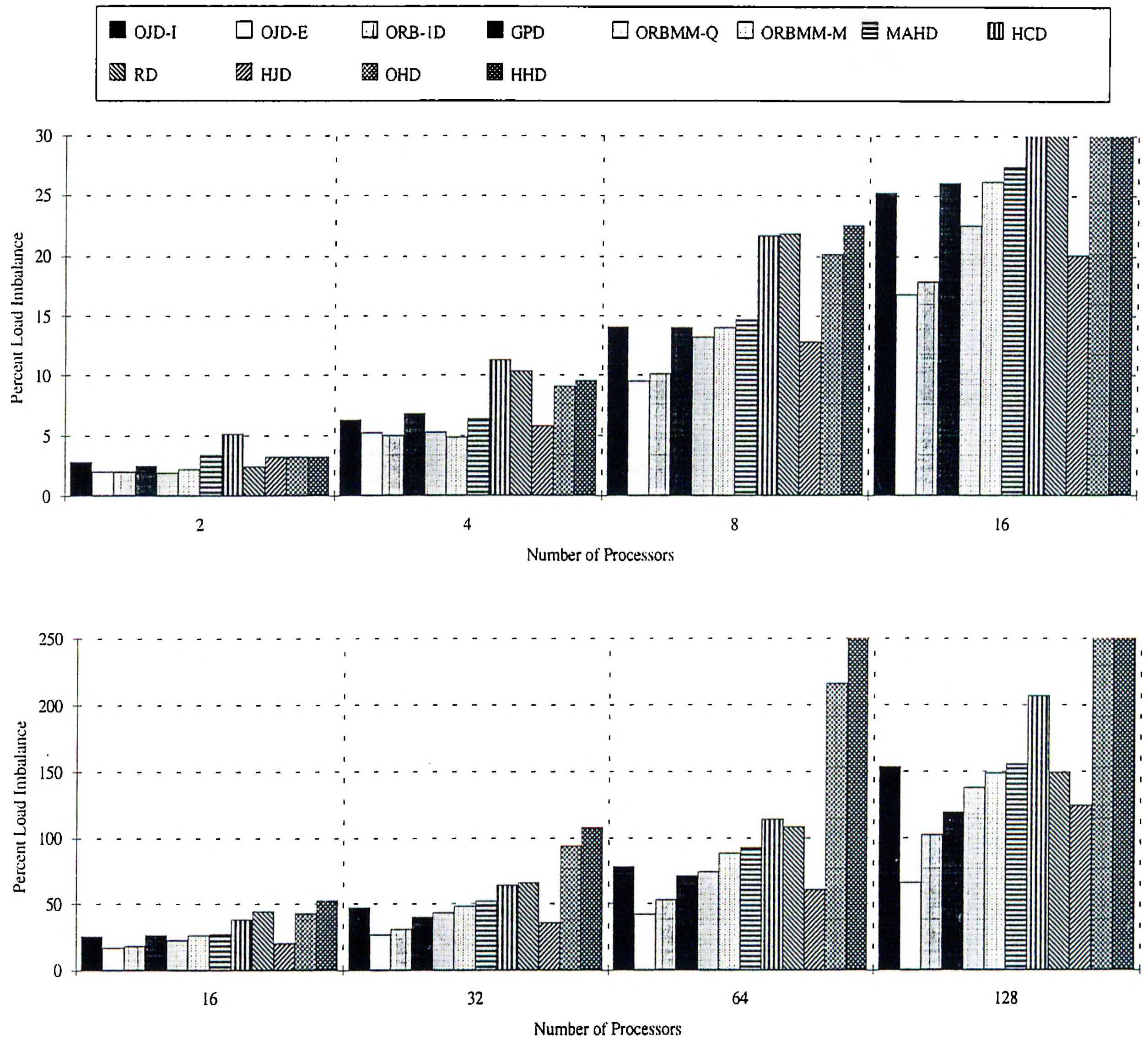


Figure 7.3. Load balance performance of algorithms on different number of processors.

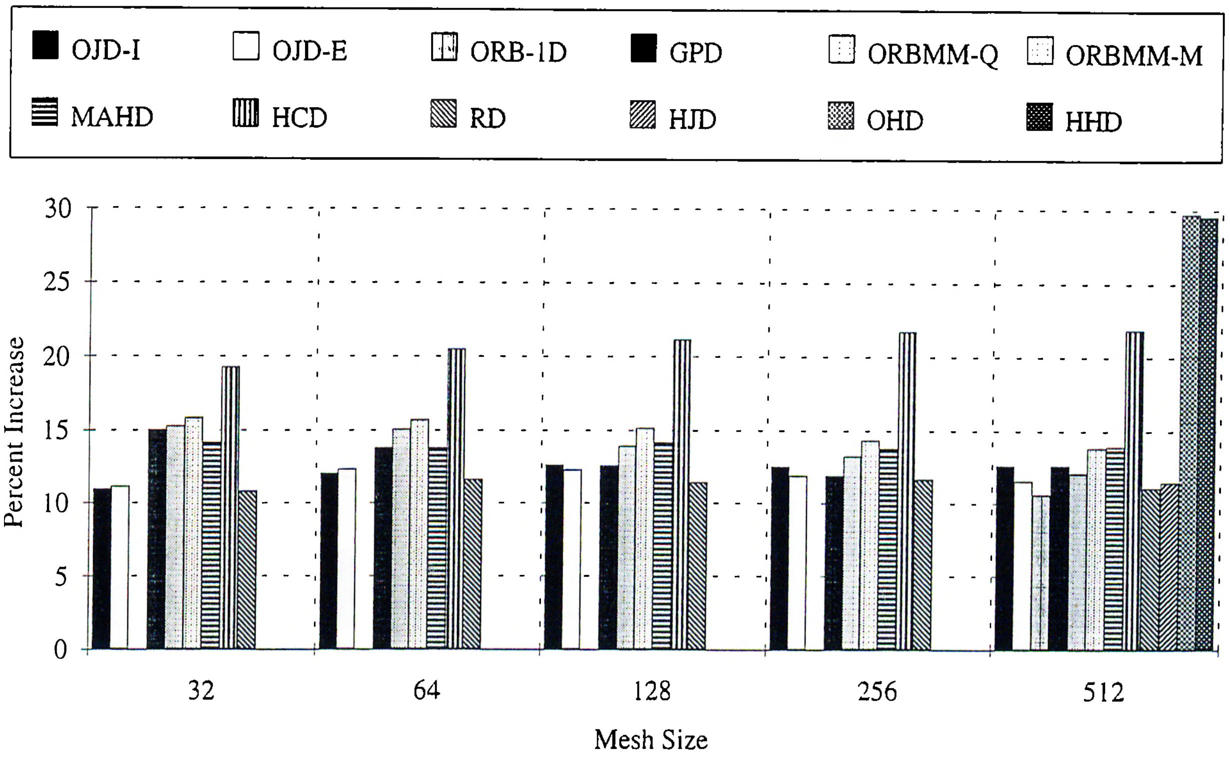


Figure 7.4. Percent increase in the number of primitives after redistribution for different mesh resolutions on 16 processors.

Table 7.2. Dissection of execution times (in seconds) of the decomposition algorithms for different number of processors. Here, BB denotes bounding box creation time, D denotes decomposition time, and R denotes redistribution time.

P	2			4			8			16		
Algorithm	BB	D	R	BB	D	R	BB	D	R	BB	D	R
OJD-I		1.79	0.88		1.34	0.6		1.11	0.48		0.97	0.40
OJD-E		1.81	0.88		1.77	0.63		1.8	0.47		1.84	0.39
ORB-1D		0.47	0.88		0.49	0.71		0.42	0.58		0.39	0.51
GPD		7.50	4.73		5.53	2.09		5.06	1.57		4.75	1.20
ORBMM-Q		2.06	3.04		1.63	2.15		1.43	1.63		1.28	1.25
ORBMM-M	2.18	1.74	2.93	1.19	1.41	2.03	0.65	1.23	1.50	0.34	1.13	1.14
MAHD		1.52	0.88		1.15	0.71		0.95	0.60		0.80	0.52
HCD		2.06	2.96		1.60	2.05		1.39	1.53		1.22	1.18
RD		1.56	0.88		1.59	0.64		1.59	0.48		1.56	0.40
HJD		0.18	0.93		0.14	0.71		0.15	0.60		0.17	0.53
OHD		0.05	0.88		0.05	0.65		0.05	0.49		0.08	0.43
HHD		0.05	0.93		0.05	0.74		0.06	0.63		0.08	0.59

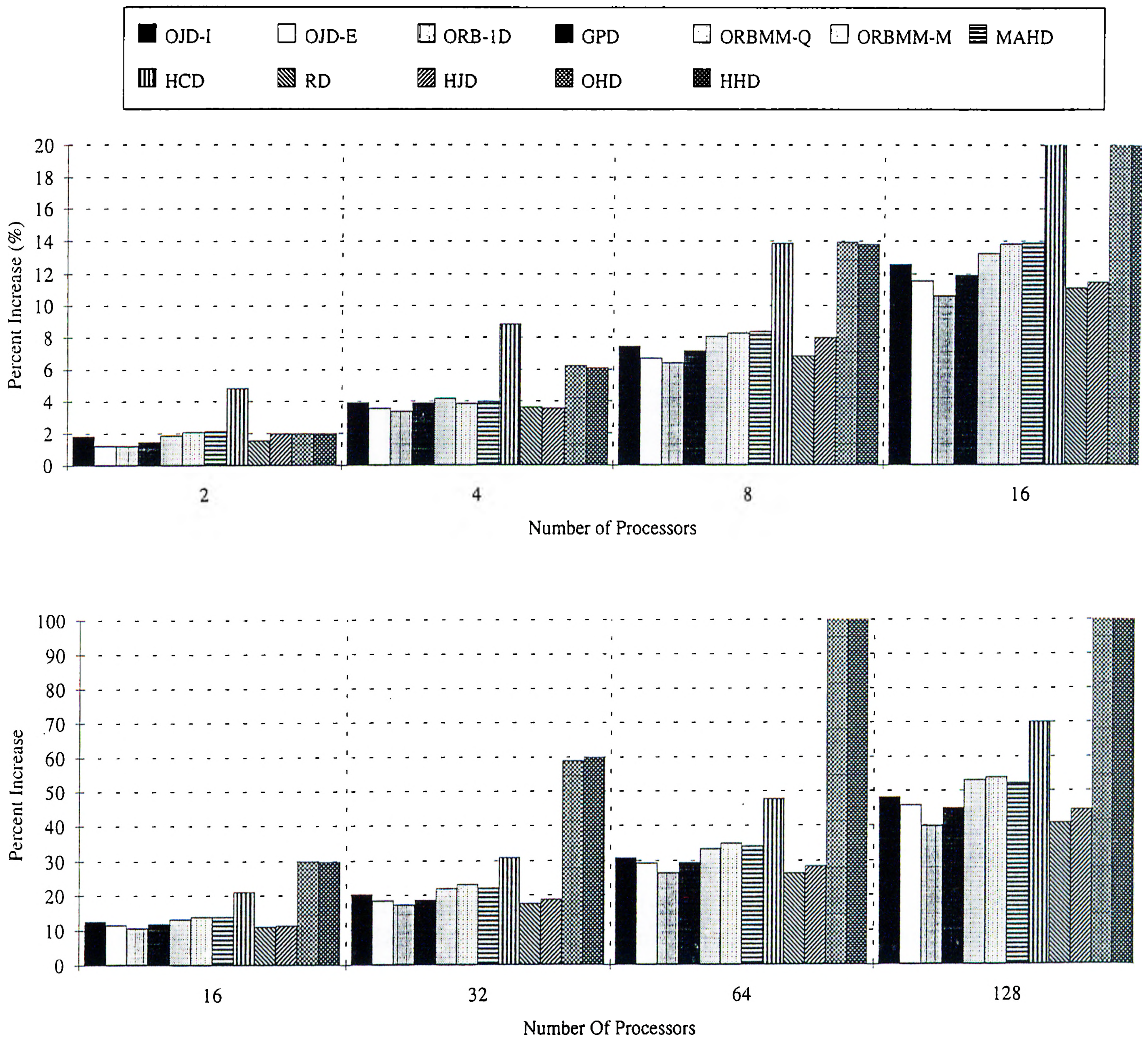


Figure 7.5. Percent increase in the number of primitives after redistribution for different number of processors. Each value in the graph represents the percent increase for the mesh resolution the algorithm achieves its best load balance.

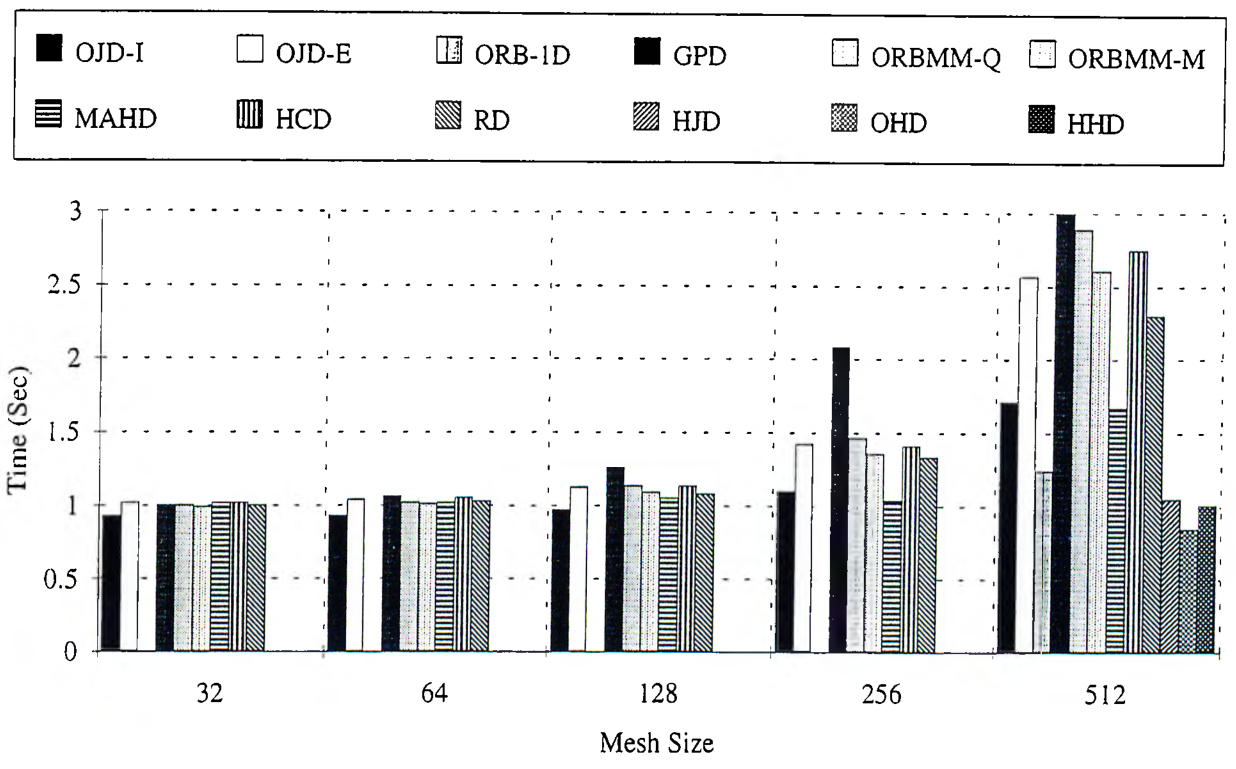


Figure 7.6. Execution times of the decomposition algorithms varying the mesh resolution on 16 processors.

Table 7.3. Dissection of decomposition times (in milliseconds) of the decomposition algorithms ORBMM-Q and OJD-E varying the mesh resolution and number of processors. Here A denotes allocation time, F denotes workload array fill time, G denotes global sum time, and D denotes decomposition operation time.

P		4				8				16			
Algorithm	Mesh	A	F	G	D	A	F	G	D	A	F	G	D
ORBMM-M	128	3	163	26	6	5	92	48	10	7	51	90	10
	256	12	320	57	27	12	183	82	37	20	107	127	52
	512	53	1113	210	147	52	675	245	194	57	425	278	340
OJP-E	128	15	85	85	10	20	45	142	42	22	30	242	52
	256	52	173	242	18	57	85	327	92	52	52	412	107
	512	198	445	961	35	202	270	1062	160	207	170	1125	210

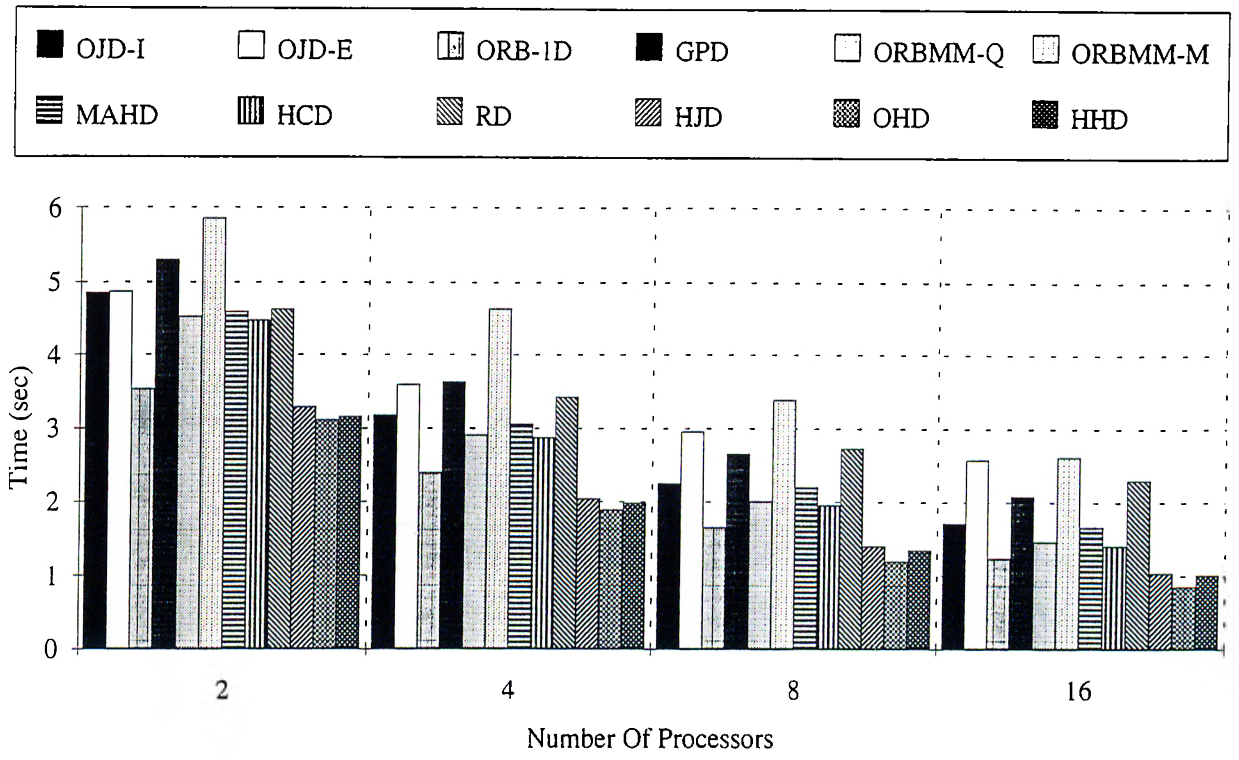


Figure 7.7. Execution times of the decomposition algorithms on different number of processors. The execution times of GPD, ORBMM-Q and HCD are measured at mesh resolution of 256×256, others at 512×512.

Table 7.4. Redistribution times of different approaches varying the mesh resolution when P = 16 and varying the number of processors when mesh resolution is 512×512.

	Mesh Resolution			
Algorithm	64	128	256	512
Inverse Mapping	0.40	0.40	0.40	0.39
Rectangle Intersect.	0.53	0.53	0.53	0.52
Mesh Based	0.47	0.50	0.61	1.14
	Number of Processors			
	2	4	8	16
Inverse Mapping	0.88	0.63	0.47	0.39
Rectangle Intersect.	0.88	0.71	0.60	0.52
Mesh Based	2.93	2.03	1.50	1.14

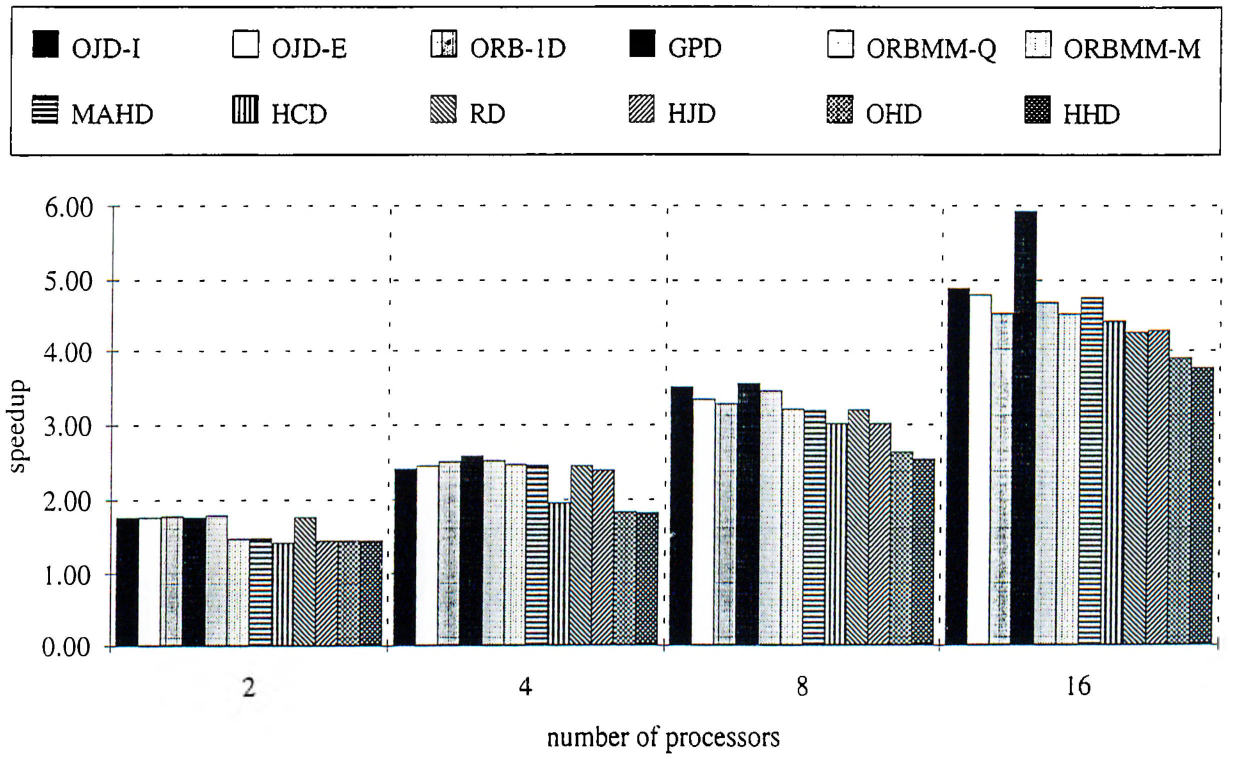


Figure 7.8. Speedup for parallel rendering phase when only the number of triangles is used to approximate the workload in a region

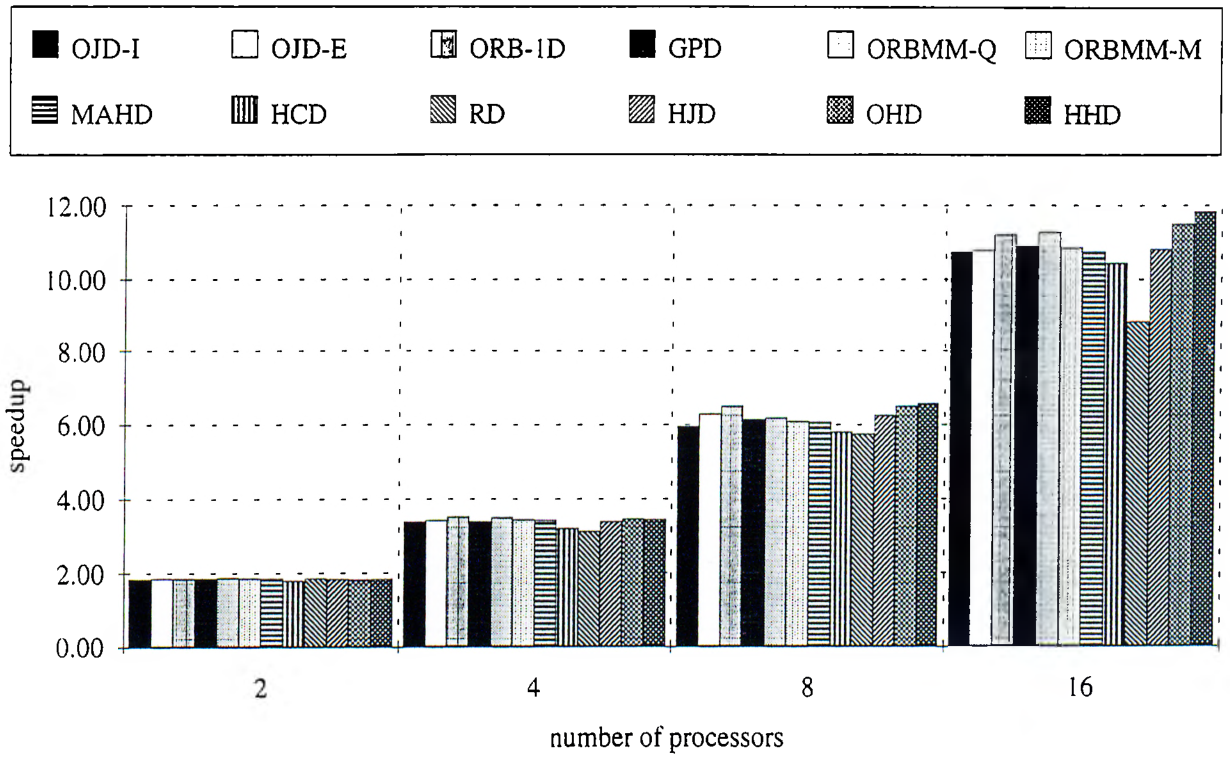


Figure 7.9. Speedup for parallel rendering phase when spans and pixels are incorporated into the workload metric

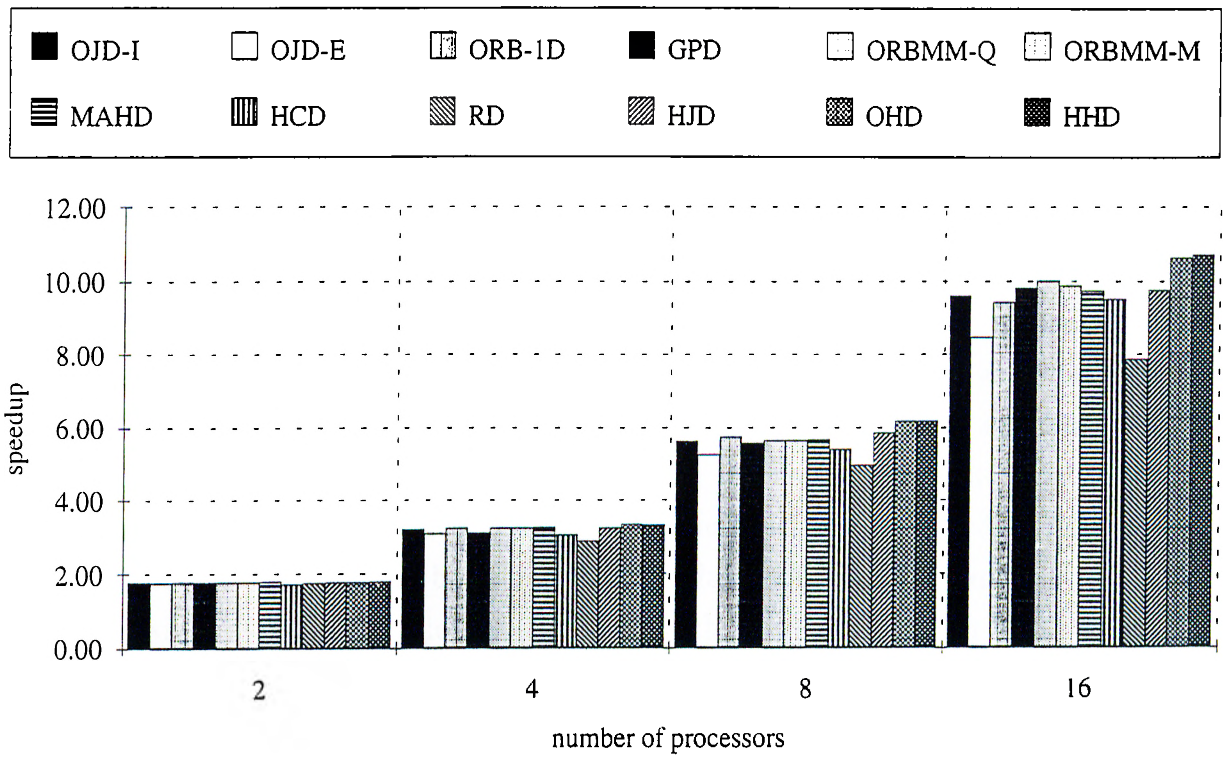


Figure 7.10. Speedup for overall parallel algorithm (including decomposition and redistribution times) when spans and pixels are incorporated into the workload metric

8. CONCLUSIONS

In this thesis, image-space decomposition algorithms are proposed and utilized for parallel implementation of a direct volume rendering (DVR) algorithm. Screen-space bounding box of a primitive is used to approximate the coverage of the primitive on the screen. Number of bounding boxes in a region is used as a workload of the region.

A taxonomy is proposed for image-space decomposition algorithms described in the thesis. This taxonomy is based on the decomposition strategy and workload arrays used in the decomposition. There were three workload-array schemes used by the decomposition algorithms : 1D arrays, inverse area heuristic and exact model. Among these three exact model is a new scheme proposed to find the exact number of bounding boxes in a rectangular region in $O(1)$ time.

Chains-on-chains partitioning (CCP) algorithms are exploited for load balancing in some of the proposed decomposition schemes. Probe-based CCP algorithms are used for optimal 1D horizontal decomposition and iterative 2D rectilinear decomposition. A new probe-based optimal 2D jagged decomposition algorithm is proposed for general workload arrays. Summed-area table(SAT) scheme is exploited to increase the efficiency of 2D optimal jagged and iterative 2D rectilinear decompositions of general 2D workload arrays. SAT scheme allows to find the workload of any rectangular region in $O(1)$ time, thus avoiding many collapsing operations in these algorithms. These two decomposition algorithms are successfully adopted for image-space decomposition using the exact model (OJD-E and RD algorithms).

Also, new algorithms that use IAH are implemented for image-space decomposition. Orthogonal recursive bisection algorithm with medians of medians scheme is applied on regular mesh and quadtree superimposed on the screen (ORBMM-M and ORBMM-Q algorithms). Hilbert space filling curve is also exploited for image-space decomposition (HCD algorithm).

An efficient inverse-mapping based redistribution algorithms are proposed for horizontal, rectilinear and jagged decompositions.

We experimentally evaluated 12 image-space decomposition algorithms on a common framework with respect to the load balance performance, the number of shared primitives, and execution time of the decomposition algorithms. The experimental results show that

- 2D partitioning gives better load balance performance than 1D partitioning algorithms since search space of 2D partitioning algorithms is larger.
- OJP-E achieves the best load balance, since it uses exact number of bounding boxes for load balance and finds global optimum for jagged decomposition.
- The perimeter of the resulting partitions and number of neighbor regions effect the number of shared primitives.
- ORB-1D and RD gives the least number of shared primitives.
- Algorithms that uses 1D arrays run faster. Global sum operation becomes bottleneck for the exact model.

The sequential DVR algorithm is a polygon rendering based algorithm. It requires volume elements composed of polygons and utilizes a scanline z-buffer approach for rendering. The volume grids used in the experiments are converted into a set of distinct triangles for rendering these data sets. First, the number of primitives in a region is used to represent the work load associated with that region. We observe that only the number of primitives in a region does not provide a good approximation to actual computational load. The number of spans and pixels generated during the rendering of primitives are incorporated into the algorithms to approximate work load better. Spans and pixels are also represented as a bounding box for decomposition algorithms. Each pixel and span has a workload weight relative to the workload of the triangle. The speedup values have almost doubled using these additional factors. Experiments are carried out using three data sets *blunt fin* (composed of 381K triangles), *delta wing* (composed of 2032k triangles) and *liquid oxygen post* (composed of 1040K triangles) using six different viewing positions for each data set. On the average, we can render the data sets used in the experiments in about 7.63 seconds on 16 processors of Parsytec CC system with a speedup of 10.69.

A. EXPERIMENTING WITH THE COMMUNICATION PERFORMANCE OF PARSYTEC CC SYSTEM

In this work, we have performed series of experiments on the communication operations of the Parsytec CC parallel system installed in our department. The results are evaluated and some important properties of Parsytec CC system are discussed. The algorithms were implemented in the C language and the native message passing library of CC system called Embedded Parix (EPX) were used.

The notations and terminology used throughout this report are based on the terminology used in [25] and [40]. The reader is assumed to have some experience with parallel processing.

We present and experimentally evaluate three basic communication operations: ping-pong, collect and distributed global-sum operations. The collect operation is implemented for star, ring and hypercube, and distributed global sum operation for ring and hypercube topologies. In ping-pong operation, the constant PARTNER is found by XORing node id of related process. For the ring topology the node is connected to its two neighbors with FORWARD and BACKWARD links. The node is connected to the node with an id $((my\_node\_id + 1)\ mod\ P)$ by FORWARD link and to the node with the id $((my\_node\_id - 1)\ mod\ P)$ by BACKWARD link. For both collect and distributed global sum operations the data is stored in a buffer called X and the partitions of X is specified by an index array IX. The buffer segment for node i stored in X is bounded by $IX[i]$ to $IX[i+1]-1$.

The algorithm complexities are given as the number of communication steps (startup time) and concurrent volume of communication. The startup time is taken to be the time spent to send zero-byte message to a destination node. It is denoted as T_s in this work. The concurrent volume of communication is given in bytes. The next section will give brief information about the architecture of the Parsytec CC system and its environment for parallel programming. The third section presents the algorithms and discusses experimental results. In section 4, we conclude the report.

A.1 Parsytec CC System

A.1.1 Hardware

The Parsytec CC system is a parallel computer manufactured by Parsytec GmbH in Aachen, Germany. It is based on distributed-memory MIMD architecture. The nodes are connected with each other via the high speed serial communication link. The communication network uses router modules that use deadlock free worm-hole packet routing mechanism.

The Parsytec CC system, installed in our department consists of 24 nodes each with a 133 MHz PowerPC 604 chip, an L1 cache of 2 * 16KB (instruction + data) and a 512 KB L2 cache. Each node contains 1 Gbits/s HS(high speed) link card connected to PCI bus. The memory controller MPC105 allows PCI bus and processor to access either to memory or cache. The DMA access also exploits the existence of data in cache.

Our system has two entry nodes each with 128 MB memory and 2 GB local disk space and 2 I/O nodes with 128 MB memory and 2 GB local disk space. Each of the 20 compute nodes has 64 MB memory and 340 MB local disk space used for page swapping. Communication between nodes is handled by a high speed network with a peak performance of 40 MB/s. The high speed network uses 6 routers for establishing communication paths between all nodes. The entry nodes have an Ethernet adapter for communication with the outside world.

Each router in the system has 8 ports, four of which are used for processor connection and the other four are used for inter-router connections. Two nodes connected to the same router communicate directly with each other via the router. Two processors connected to different routers communicate with each other via the inter-router connection network. That is, the message from the source processor traverses a path from the router of the source processor to the router of the destination processor through the inter-router connection network. A node communicates with a node, which is at the same column with the source node but at a separate router, in the following way: the message exits from source node to its router, then it goes to a router at the opposite side and then goes to the router of the destination node. In this type of communication, a message traverses three routers. The logical topology of our CC system is shown in Figure A.1.

A.1.2 Software

All nodes of Parsytec CC system run the AIX operating system with EPX, Embedded Parix, on top. ANSI C, F77 and C++ compilers are available. PVM 3.3 can also be used for communication.

EPX provides a set of functions the use the communication network and define suitable routines managing data operations. A set of processors (a partition) is assigned to a user to run his/her program. Each processor of the EPX partition is arranged as a virtual 3D grid. In our system it is arranged as a 6x4x1 grid. So effectively 2D grid of size 6x4 is used for our system. This grid can be partitioned to subgrids of less number of processors by the system administrator.

The global information about nodes, consisting of id, location in grid, number of nodes, the dimensions of the grid, can be accessed with GET_ROOT routine.

In the EPX programming environment, communication between two nodes is performed via virtual links. A virtual link is a bidirectional synchronizing, non-buffering, point-to-point communication line between two threads. A set of virtual links can be combined to build a virtual topology. EPX provides some well-known virtual topologies like ring, grid, hypercube, star and tree.

EPX has three types of communication for message passing. The first type of communication is synchronous virtual link-bound communication. The communication processes are connected via virtual link and synchronize upon communication. The first process which reaches the point of communication waits the other one so that the communication takes place. This type of communication is blocking in that Send() function returns when the last byte of the buffer leaves the processor and Recv() returns when the last byte of the message enters the

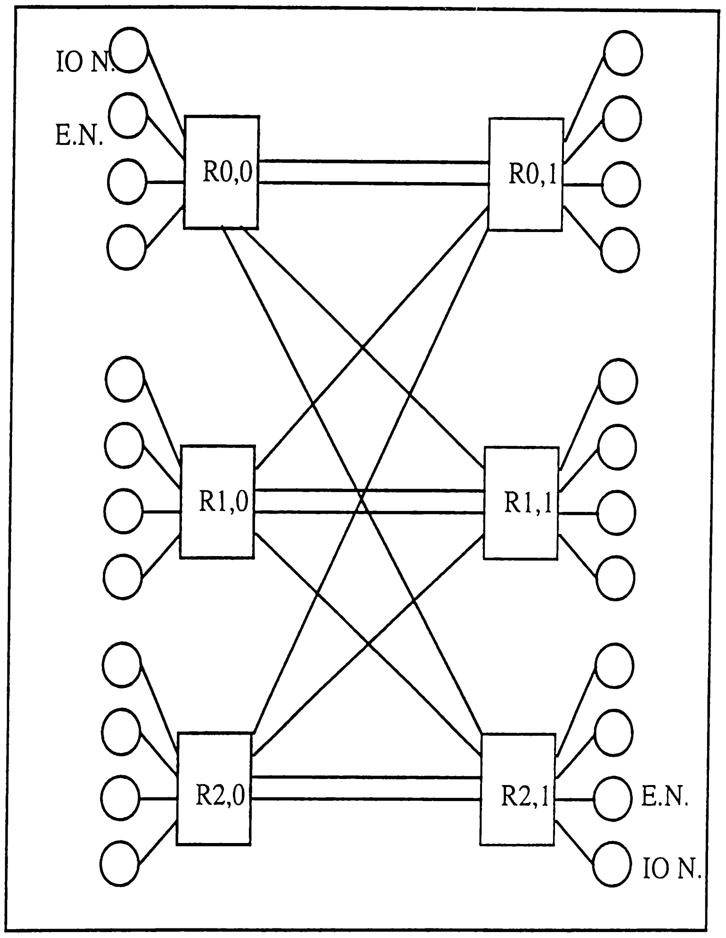


Figure A.1. The logical topology of our Parsytec CC system

receiving buffer. The routines provides by EPX for this type communication are Send(), Recv(), Select(), SendLink() and RecvLink().

The second type is the synchronous random communication where there is no need to define the virtual links. This is used for small message sizes. The routines for this are SendNode() and RecvNode(). This type is also blocking.

The third type is non-blocking link-bound communication. In this type, communication can concurrently continue with computation. A thread is created to perform communication concurrently. After the thread is created for communication, program continues from the next line. The library routines AInit(), ASend(), ARecv(), ASync(), AExit() are used for this type of communication. The routine AInit() must be called at least once before using this type of communication, afterwards any number of asynchronous communications can be performed. The routine ASync() waits for the completion of communication on a link.

The common virtual topologies can be created by calling related routines like Make2DGrid(), MakeHCube() or MakeRing(). Also AddTop() routine can be used to create user defined topologies. After creation of a virtual topology, each node gets a node id for the topology independent of the id given in the partition, and similarly each link gets a link id.

More information for communication on EPX can be found in its reference and programmers guides [40].

A.2 Basic Communication Operations

A.2.1 Ping-Pong

Ping-pong is a simple, basic communication operation in which two nodes communicate with each other to exchange data. We performed several experiments with simple Ping-Pong programs to analyze the behavior of EPX communication routines in our Parsytec CC system.

A.2.1.1 Program

The first ping-pong program uses blocking send and receive routines. Node 0 first sends a message to node 1 and then receives a message from that node. Similarly, node 1 first receives the message node 0 has sent and sends its message to node 0. The communication is blocking, because program does not continue until communication is completed. As blocking communication is used, the time to perform both send and receive operations is exactly twice the time spent for a one-sided communication between two nodes. A portion of the code for this program is shown in Figure A.2.

The second ping-pong program uses asynchronous communication, which utilizes non-blocking send routines. Both nodes 0 and 1 initiate asynchronous send (ASend() routine of EPX) operation followed by a blocking receive operations. This is asynchronous communication because the program continues working while send operation is performed in the background. The program waits for the finish of send operation at a call of ASync() function. The program is shown in Figure A.3.

```
if (ringData->id==0)
{
    Send(topId, PARTNER,(char *) send_buffer,MESSAGE_SIZE);
    Recv(topId, PARTNER,(char *) receive_buffer,MESSAGE_SIZE);
}
else
{
    Recv(topId, PARTNER,(char *) receive_buffer,MESSAGE_SIZE);
    Send(topId, PARTNER,(char *) receive_buffer,MESSAGE_SIZE);
}
```

Figure A.2. Blocking Ping-Pong Program

```
ASend(topId, PARTNER,(char *) send_buffer,MESSAGE_SIZE,&error);
Recv(topId, PARTNER,(char *) receive_buffer,MESSAGE_SIZE);
ASync(topId,PARTNER);
```

Figure A.3. Non-blocking Ping-Pong Program

A.2.1.2 Experimental Results and Discussion

We present experimental results for different message sizes which change from one byte to 4 MB. Six different data set have been used to see the behavior of the system. All results are shown in Table A.1. Timings in columns 2, 4 and 6 are the average of 100 iterations of the same ping-pong program. whereas timings in columns 1, 3 and 5 are taken for one iteration to exclude cache affect.

We used messages of one byte to measure the startup overhead of communications on Parsytec CC system. The blocking ping-pong operation takes 0.85 msec. So we can say that if communication is performed on virtual topology links with blocking send receive, the overhead is approximately 0.43 msec per communication. We performed the same blocking ping-pong operation, but this time we changed the Send() commands to ASend() commands but put ASync() commands right after sends. In this version, the communication is also blocking

Message Length(bytes)	(1) Blocked ping-pong (No- cache)	(2) blocked ping-pong (cache)	(3) Asynchronous ping-pong (No cache)	(4) Asynchronous ping-pong (cache)	(5) Asynchronous ping-pong, secondary buffering (No-cache)	(6) Asynchronous ping-pong, secondary buffering (cache)
1		0.85		1.18		1.24
1K	3	1.33	3	1.6	5	1.65
4K	6	2.78	5	2.44	6	3.5
64K	13	7.27	11	5.37	15	8.7
128K	26	13.9	18	9.8	26	17.4
256K	51	27	32	19	50	35
512K	100	52.5	66	37	95	70
1M	198	103	130	73	180	140
2M	300	205	195	143	355	280
4M	503	410	340	296	740	545

Table A.1. Timings for Ping-Pong programs

but only we add overhead of asynchronous communication. We measured that it took 1.14 msec which means 0.57 msec startup time for each communication, so the extra overhead due to extra operations, like thread creation, is equal to 0.57-0.43 = 0.14 msec. Finally, we performed one byte communication with asynchronous communication and measure 1.18 msec total time, which is higher than both of previous experiments.

In our experiments with larger message sizes to measure the communication bandwidth, we observed the importance of cache in communication. Also the effect of PCI bus congestion during concurrent communication is realized. Moreover, we see that usage of intermediate buffer with asynchronous communication brings extra overhead. In the next paragraphs these effects will be explained in more detail.

As it has been explained in Section A.1, Power PC 604 processors access the memory, L2 cache and PCI bus via a memory controller. We observe that memory controller directs the memory requests coming from either processor or PCI bus to L2 cache before main memory. This behavior of nodes makes difference in communication times according to existence of send or receive buffers in L2 cache.

The column 1 of Table A.1 shows the results for blocking ping-pong where the send and receive buffers are not in cache. Column 2 presents the case when send and receive buffers are accessed before communication. As we compare columns 1 and 2, we see that the difference until message of 1 MB is nearly double. After 1 MB the difference becomes constant due to size of the L2 cache in nodes. The same gain with cache can also be seen in columns 3 and 4.

Point to point communication speed of machine can be calculated using column 2. On the average, we measure $1/(102/2) = 20$MB/s speed.

The usage of intermediate buffering in asynchronous communication brings extra overhead to communication. We can see this affect by comparing columns 3 with 5 and 4 with 6. Columns 5 and 6 are the versions of 3 and 4 respectively when intermediate buffering is used in communication. It is advisable that if send buffer will not be used immediately after communication, intermediate buffering mechanism should not be used. This can be achieved by passing 0 to the buffer_size parameter of AInit() function.

Another observation with non-blocking ping-pong program is that asynchronous communication mechanism does not overlap the times of two communications at the same node. This is due to the multiplexing of PCI bus between two communications at the same node. In non-blocking ping-pong, nodes initiate send and receive operations concurrently. As seen in Table A.1. exchanging one MB with non-blocking ping-pong takes 73 msec. This is surely not half of the time of the blocking ping-pong program. PCI bus congestion causes performance decrease in performance of several basic communication operations.

A.2.2 Collect

The collect operation is one of the basic communication operations of parallel programming. It is also known as single node gather. In collect operation, initially every node has messages of size M bytes. After the termination of collect operation, data from all nodes is collected at a single destination processor.

A.2.2.1 Program

The simplest communication pattern for collect operation is on star topology. Each leaf node sends its message to the root of star and the root simply takes P-1 messages from leaves. The algorithm is illustrated in Figure A.4.

```
switch (starData->status) {
  case STAR_ROOT:
    for (i=1; i<starData->size; i++) {
      ARecv (topId, i,&(buf[(i+1)*BUF_SIZE]) , BUF_SIZE, &error );
    }
    ASync(topId,-1);
    break;
  case STAR_LEAVE:
    Send(topId,0,buf, SINGLE_BUF_SIZE);
    break;
}
```

Figure A.4. Collect operation on star topology

The communication pattern for ring topology is the inverse of one-to-all personalized communication. The algorithm for ring topology is shown in Figure A.5. At the ith step, the node whose node number is between $i + 1$ and $P - 1$ sends its message in forward direction to the next node. The node whose node number is $i + 1$ becomes idle after that step. Node 0 collects the received messages and does not issue any send calls.

In Figure A.6, the collect algorithm for hypercube topology is shown. At the ith step of the algorithm a node selects its partner by XORing the ith bit of its node number. Between two partners the node whose ith bit is 1 sends its message to its partner and becomes idle for the next steps. At the same step, the node whose ith bit is 0 receives the message. So at each step, the number of active nodes is halved, meanwhile, the message lengths are doubled.

The concurrent volume of communication for all three topologies is $(P-1)*M$. The startup overhead of ring and star are $(P-1)*T_s$ whereas it is $\log(P)*T_s$ for hypercube.

A.2.2.2 Experimental Results and Discussion

As seen in Table A.2, the hypercube topology is the best of three for collect operation. The problem with ring topology is the PCI bus congestion due to concurrent send and receive operations on the same node. The times of star

```
mynode=ringData->id;
nextnode=(mynode+1)%P;
prevnode=(mynode-1+P)%P;
for(q=0;q<P-1;q++)
{
  s=(mynode-q+P)%P;
  sptr=IX[s];
  r=(prevnode-q+P)%P;
  rptr=IX[r];
  msgsize=(IX[s+1]-IX[s])*sizeof(DATA_TYPE);
  rmsgsize=(IX[r+1]-IX[r])*sizeof(DATA_TYPE);
  if (mynode==0)
    Recv (topId, BACKWARD, &(X[rptr]),maxmsgsize);
  else
  if(mynode>q)
  {
    ASend(topId, FORWARD, (char *) &(X[sptr]),msgsize.&error);
    if (mynode!=q+1)
      Recv (topId, BACKWARD, &(X[rptr]),maxmsgsize);
    ASync(topId,FORWARD);
  }
}
```

Figure A.5. Collect operation on ring topology

```
myend=P;
size=1;
sindex=mynode;
for(q=0;q<dim;q++) {
  if (mynode&(1<<q))
    rindex=sindex-size;
  else
    rindex=sindex+size;
  msgsize=(IX[sindex+size]-IX[sindex])*sizeof(DATA_TYPE);
  rmsgsize=(IX[rindex+size]-IX[rindex])*sizeof(DATA_TYPE);
  sptr=IX[sindex]; rptr=IX[rindex];
  if (mynode&(1<<q)){
    Send (topId,q, (char *) &(X[sptr]),msgsize);
    break;
  } else
    Recv (topId,q, (char *) &(X[rptr]),rmsgsize);
  size*=2;
}
```

Figure A.6. Collect operation on hypercube topology

topology has small difference from that of hypercube. The difference comes from the higher startup overhead of star topology. The bus congestion problem does not occur in collect operation with hypercube or star topology. This is due to fact that, at any step a node initiates only one type of communication, namely send or receive.

A.2.3 Distributed Global Sum (DGS)

A.2.3.1 Program

Distributed global sum operation is the two phase version of global sum operation. The global sum operation is one of the basic communication operations used frequently in parallel programs. In the global sum operation, every node starts with a vector of length M and needs to know the vector sum. The simplest way to do this is to use reduction operation. The communication pattern is the same as of all-to-all broadcast. But, at each step the local vector and received vector

P	Message Length (bytes)	Star	Ring	Hypercube
4	64	1.25	3	0.88
	4K	1.8	4.4	1.5
	128K	7.2	9.4	5.6
	256K	13	14.5	10.4
	512K	21	26	20
	1M	40	48	39
	2M	78	94	77
8	64	2.9	8.3	1.4
	4K	4	11	2.2
	128K	10.5	18	7
	256K	17	24	12.5
	512K	30	36	23
	1M	48	67	45
	2M	91	131	88
16	64	6.3	18	2
	4K	8.7	23	2.9
	128K	15.5	34	8.2
	256K	23	42	14
	512K	36.5	54	26
	1M	64	81	50
	2M	102	152	97

Table A.2. Collect operation Timings

are added and the resulting vector is used as a send vector in the next step. The concurrent volume of communication in this operation is $M \log(P)$ for hypercube and $M(P-1)$ for ring topology. The startup overhead for ring is $(P-1)T_s$ and $(\log(P))T_s$ for hypercube. An alternative to the above algorithm with lower concurrent volume of communication is to use two phase distributed global sum operation. The first phase is called fold operation in which every processor starts with a vector and at the end has a M/P parts of vector whose sum has been computed. At the second phase, these parts are distributed to every processor using all-to-all broadcast operation. The concurrent volume of communication for fold and all-to-all broadcast on both hypercube and ring is $M * (P-1)/P$.

Therefore total concurrent volume of communication for distributed global sum is $2 * M * (P-1)/P$, which is less than both $M \log(P)$ and M(P-1). A program for distributed global sum is shown in Figure A.7.

```
mystart=0; myend=P;
for (q=dim-1;q>=0;q-){
  if (mynode & (1<<q)){
    sptr=IX[mystart];
  msgsize=(IX[(mystart+myend)/2]-IX[mystart])*sizeof(DATA_TYPE);
  mystart=(mystart+myend)/2;
  }
  else {
    msgsize=(IX[myend]-IX[(mystart+myend)/2])*sizeof(DATA_TYPE);
    myend=(mystart+myend)/2; sptr=IX[myend];
  }
  ARecv (topId,q, &(RB[0]),maxmsgsize,&error);
  Send (topId,q, (char *) &(X[sptr]),msgsize);
  ASync(topId,q);
  istart=IX[mystart]; iend=IX[myend]; j=0;
  for (i=istart;i¡iend;i++){
    X[i]+=RB[j]; j++;
}
size=1; sindex=mynode;
for(q=0;q<dim;q++) {
  if (mynode&(1<<q))
    rindex=sindex-size;
  else
    rindex=sindex+size;
  msgsize=(IX[sindex+size]-IX[sindex])*sizeof(DATA_TYPE);
  rmsgsize=(IX[rindex+size]-IX[rindex])*sizeof(DATA_TYPE);
  sptr=IX[sindex]; rptr=IX[rindex];
  ARecv (topId,q, (char *) &(X[rptr]),rmsgsize,&error);
  Send (topId,q, (char *) &(X[sptr]),msgsize);
  ASync(topId,q);
  if (mynode&(1<<q) ) sindex-=size;
  size*=2;
  }
}
```

Figure A.7. DGS on hypercube

A.2.3.2 Experimental Results and Discussion

We have implemented the distributed global sum operation for both hypercube and ring topologies. The results are taken for 4, 8, 16 processors by varying the vector size. These results are shown in Table A.3 and Figure A.8.

We see in Figure 8-a that for four processors, hypercube gives better results than ring. This is due to less startup overhead of hypercube topology. However, as seen in Figure 8-b and Figure 8-c, on 8 and 16 processors for message lengths greater than 256k, the ring becomes faster. This is due to the problem of congestion free mapping of hypercube to our system. Because of our router topology, it is impossible to map 16 processors congestion free to hypercube topology. Although it is possible to achieve congestion free communication with current system for 8 processors, EPX's mapping function cannot achieve it. In fact, EPX makes its mapping in a more general way, without regarding any communication pattern. The difference in total communication time becomes higher as the message sizes increase. Note that PCI bus congestion exists for both topologies in DGS. This is because of the fact that at each step every node performs send and receive operations concurrently.

A.3 Conclusion

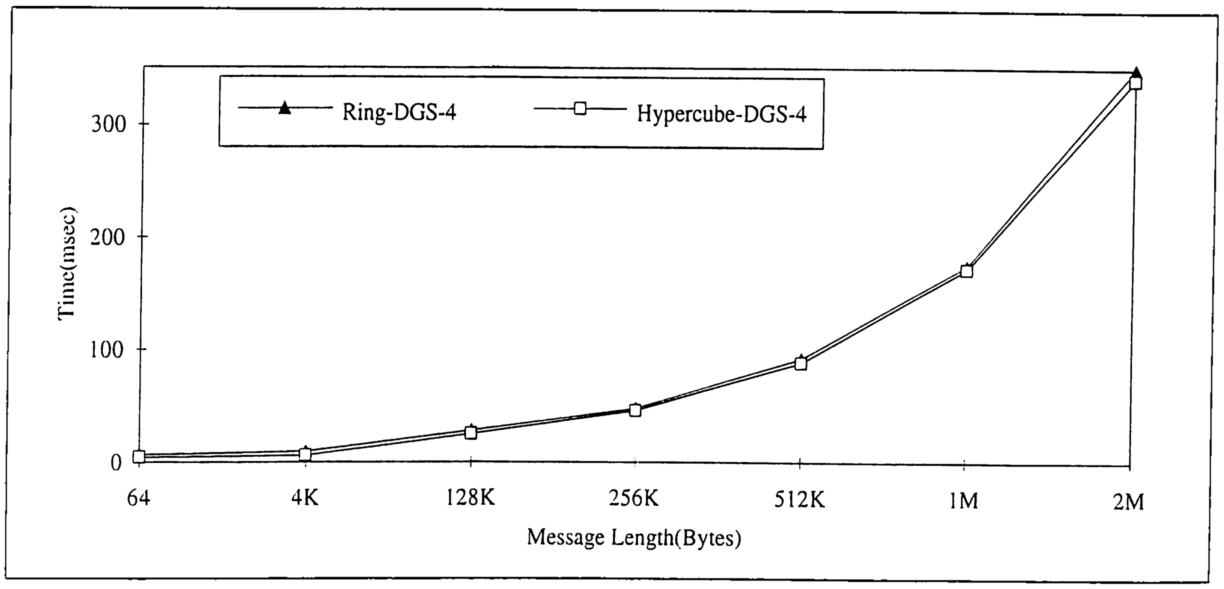
We have presented the results of several experiments that measure the key aspects of inter-processor communication on Parsytec CC system.

First, startup overhead is approximately 0.5 msec. So, in most cases the number of communication is also as important as the concurrent volume of communication. For point to point communication we achieve 20 MB/s speed. Second, the existence of data in cache has major affect in communication time. Therefore, parallel programs must be carefully designed to take advantage of cache. For example, communication can immediately be initiated after writing or reading the send buffer. Third, nodes do not entirely overlap the times of two communications initiated at the same node. This is due to PCI bus congestion at a node. Sometimes you may get longer time than you expect from your theoretical concurrent volume of communication calculations. Fourth, the use of intermediate buffering in asynchronous communication brings extra overhead. So if possible,

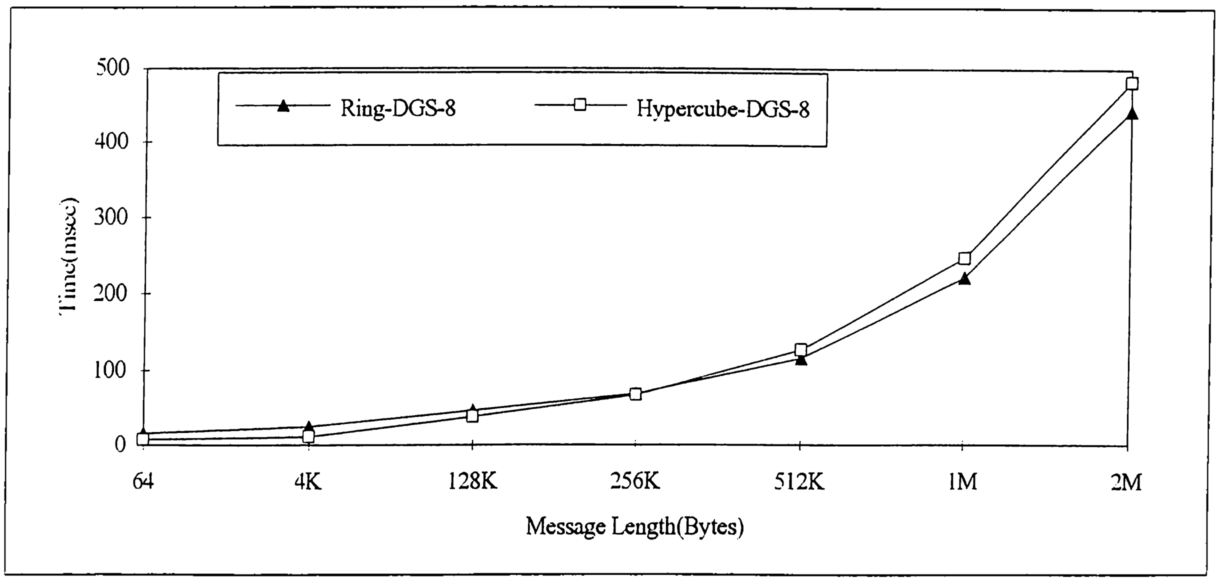
P	Message Length (bytes)	Ring-Fold	Ring-DGS	Hypercube-Fold	Hypercube-DGS
4	64	3.4	7	2.2	4.6
	4K	5	9.8	3.4	6.5
	128K	18	28	16	25
	256K	32	48	30	46
	512K	62	92	59	88
	1M	122	175	117	172
	2M	244	350	235	340
8	64	8	16	3.8	7.6
	4K	12	25	5.5	11
	128K	27	46	24	38
	256K	44	69	43	67
	512K	76	115	81	126
	1M	150	221	160	246
	2M	298	444	320	483
16	64	18	37	5	10
	4K	26	50	7	14
	128K	43	74	32	54
	256K	60	99	58	98
	512K	96	150	113	185
	1M	165	250	222	360
	2M	322	480	426	698

Table A.3. Timings For Fold and Distributed Global Sum Operations

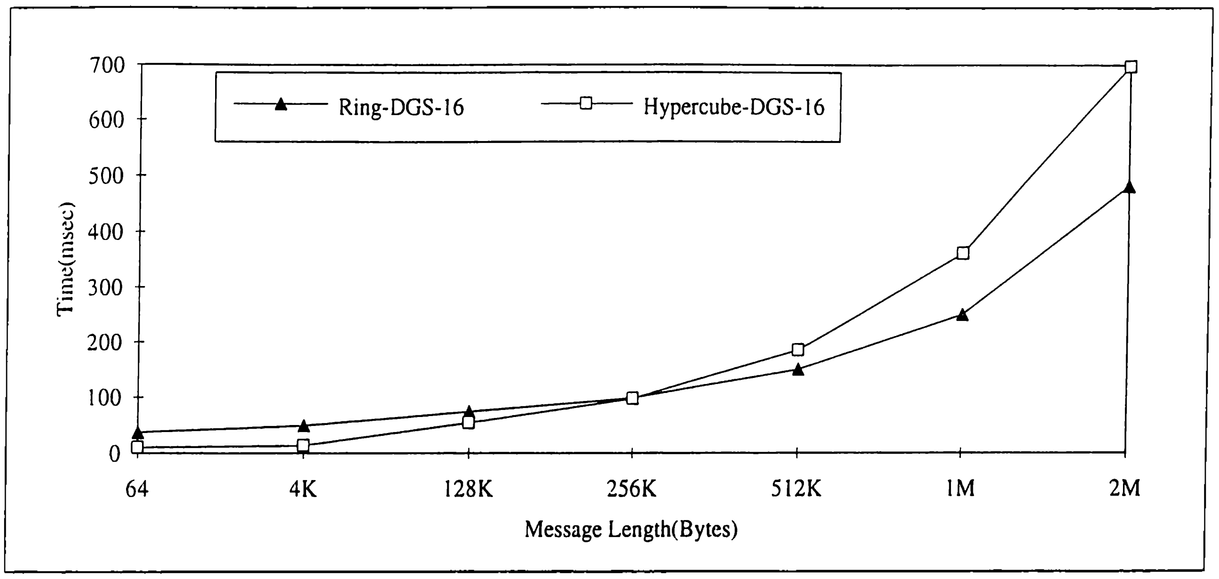
it is advisable not to use intermediate buffers. The fifth and final remark is, when embedding a well known topology to the Parsytec CC system one should be aware of possible link congestion.


Ring-DGS-4
Hypercube-DGS-4
Time(msec)
0
100
200
300
64
4K
128K
256K
512K
1M
2M
Message Length(Bytes)

(a)


Ring-DGS-8
Hypercube-DGS-8
Time(msec)
0
100
200
300
400
500
64
4K
128K
256K
512K
1M
2M
Message Length(Bytes)

(b)


Ring-DGS-16
Hypercube-DGS-16
Time(msec)
0
100
200
300
400
500
600
700
64
4K
128K
256K
512K
1M
2M
Message Length(Bytes)

(c)

Bibliography

[1] S. Anily and A. Federgruen. Structured partitioning problems. *Operations Research*, 13:130–149, 1991.

[2] C. Aykanat, V. İşler, and B. Özgüç. Efficient parallel spatial subdivision algorithm for object-based parallel ray tracing. *Computer-Aided Design*, 26(12):883–890, 1994.

[3] S. H. Bokhari. On the mapping problem. *IEEE Trans. Computers*, 3:207–214, 1981.

[4] S. H. Bokhari. Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Trans. Computers*, 1:48–57, 1988.

[5] T. Bultan and C. Aykanat. A new mapping heuristic based on mean field annealing. *J. Parallel and Distributed Computing*, pages 292–305, 1992.

[6] J. Challinger. Parallel volume rendering for curvilinear volumes. In *Proceedings of the Scalable High Performance Computing Conference*, pages 14–21. IEEE Computer Society Press, April 1992.

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

[8] J. Challinger. Scalable parallel volume raycasting for nonrectilinear computational grids. In *Proceedings of the 1993 Parallel Rendering Symposium*, pages 81–88. IEEE Computer Society Press, October 1993.

[9] H. Choi and B. Narahari. Algorithms for mapping and partitioning chain structured parallel computations. In *Proc. 1991 Int. Conf. on Parallel Processing*, pages I-625-I-628, 1991.

[10] T. W. Crockett. Parallel rendering. Technical Report 95-31, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, April 1995.

[11] F. C. Crow. Summed-area tables for texture mapping. *Computer Graphics*, 18(3):207–212, 1984.

[12] F. C. Crow. Parallelism in rendering algorithms. In *Proceedings of Graphics Interface 88*, pages 87–96, 1988.

[13] D. Ellsworth. A multicomputer polygon rendering algorithm for interactive applications. In *Proceedings of the 1993 Parallel Rendering Symposium*, pages 43–48. IEEE Computer Society Press, October 1993.

[14] M. P. Garrity. Raytracing irregular volume data. *Computer Graphics*, 24(5):35–40, 1990.

[15] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press, 1994.

[16] Genias Software GmbH, Germany. *PowerPVM/EPX for Parsytec CC systems: PowerPVM/EPX User's Guide*, 1996.

[17] M. Grigni and F. Manne. On the complexity of the generalized block distribution. In *Proc. 3rd Int. Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR'96)*, pages 319–326, 1996.

[18] P. Hansen and K. W. Lih. Improved algorithms for partitioning problems in parallel, pipelined and distributed computing. *IEEE Trans. Computers*, 6:769–771, june 1992.

[19] B. Hendrickson and R. Leland. The chaco user's guide (version 1.0). Technical Report SAND93-2339, Sandia National Labs., Albuquerque, NM, 1993.

[20] V. İşler. *Spatial Subdivision for Parallel Ray Casting/Tracing.* PhD thesis, Bilkent University, February 1995.

[21] M. A. Iqbal. Approximate algorithms for partitioning and assignment problems. Technical Report 86-40, ICASE, 1986.

[22] M. A. Iqbal and S. H. Bokhari. Efficient algorithms for a class of partitioning problems. Technical Report 90-49, ICASE, 1990.

[23] G. Karypis and V. Kumar. Metis: Unstructured graph partitioning and sparse matrix ordering system, version 2.0. Dept. of Computer Science, University of Minnesota. http://www.cs.umn.edu/~karypis.

[24] A. Kaufman. Volume visualization. In *Volume Visualization*, pages 1–18. IEEE Computer Society Press Tutorial, 1990.

[25] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing, Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc., California, USA, 1994.

[26] T. M. Kurç, H. Kutluca, C. Aykanat, and B. Özgüç. A comparison of spatial subdivision algorithms for sort-first rendering. In *Proceedings of HPCN Europe 1997, International Conference and Exhibition on High Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 137–146, Vienna, Austria, April 1997. Springer-Verlag.

[27] Tahsin M. Kurç. *Parallel Rendering on Multicomputers.* PhD thesis, Bilkent University, Department of Computer Engineering and Inf. Sci., 1997.

[28] H. Kutluca, T. M. Kurç, and C. Aykanat. Experimenting with the communication performance of parsytec cc system. Technical Report In preparation, Dept. of Computer Eng. and Information Sci., Bilkent University, 1997.

[29] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[30] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.

[31] B. Lucas. A scientific visualization renderer. In *Proceedings of IEEE Visualization '92*, pages 227–234. IEEE Computer Society Press, October 1992.

[32] K. Ma. Parallel volume ray-casting for unstructured-grid data on distributed-memory multicomputers. In *Proceedings of 1995 Parallel Rendering Symposium*, pages 23–30, October 1995.

[33] F. Manne and T. Sørevik. Partitioning an array onto a mesh of processors. In *Proc. 3rd Int. Workshop on Applied Parallel Computing (PARA '96)*, pages 467–476, 1996.

[34] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, July 1994.

[35] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of hilbert space-filling curve. Technical Report UMCP-CSD:CS-TR-3611, UMIACS, University of Maryland at College Park, 1996.

[36] C. Mueller. The sort-first rendering architecture for high-performance graphics. In *Proceedings of 1995 Symposium on Interactive 3D Graphics*, pages 75–84, 1995.

[37] D. M. Nicol. Rectilinear partitioning of irregular data parallel computations. *J. of Parallel and Disributed Computing*, 23:119–134, 1994.

[38] D. M. Nicol and D. R. O'Hallaron. Improved algorithms for mapping pipelined and parallel computations. *IEEE Trans. Computers*, 40(3):295–306, 1991.

[39] B. Olstad and F. Manne. Efficient partitioning of sequences. *IEEE Trans. Computers*, 11:1322–1326, 1995.

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

[41] J. R. Pilkington and S. B. Baden. Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):288–299, 1996.

[42] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[43] D. R. Roble. A load balanced parallel scanline z-buffer algorithm for the ipsc hypercube. In *Proceedings of Pixim' 88*, pages 177–192, Paris, France, October 1988.

[44] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *Computer Graphics*, 24(5):63–70, 1990.

[45] J. P. Singh, C. Holt, J. L. Hennessy, and A. Gupta. A parallel adaptive fast multipole method. In *Proceedings of Supercomputing 93*, pages 54–65, 1993.

[46] E. Tanın, T. M. Kurç, C. Aykanat, and B. Özgüç. Comparison of two image-space subdivision algorithms for direct volume rendering on distributed-memory multicomputers. In *Proceedings of the Workshop on Applied Parallel Computing in Physics, Chemistry, and Engineering Science (PARA95), August 1995*, volume 1041 of *Lecture Notes in Computer Science*, pages 503–512, Lyngby, Denmark, 1996.

[47] C. Upson and M. Keeler. Vbuffer: Visible volume rendering. *Computer Graphics*, 22(4):59–64, 1988.

[48] S. Whitman. *Multiprocessor Methods for Computer Graphics Rendering.* Jones and Bartlett Publishers, 1992.

[49] P. L. Williams. *Interactive Direct Volume Rendering of Curvilinear and Unstructured Data.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[50] P. L. Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, 1992.

[51] R. Yagel. Volume viewing: state of the art survey. In *Visualization '93, Tutorial #9, Course Notes: Volume Visualization Algorithms and Applications*, pages 82–102, 1993.