EXPLOITING REPLICATED DATA FOR COMMUNICATION LOAD BALANCING IN IMAGE-SPACE PARALLEL DIRECT VOLUME RENDERING OF UNSTRUCTURED GRIDS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BİLKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Erkan Okuyan

January, 2009

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.

Prof. Dr. Cevdet Aykanat (Advisor)

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

Assoc. Prof. Dr. Uğur Güdükbay

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

Assoc. Prof. Dr. Ezhan Karaşan

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

ABSTRACT

EXPLOITING REPLICATED DATA FOR COMMUNICATION LOAD BALANCING IN IMAGE-SPACE PARALLEL DIRECT VOLUME RENDERING OF UNSTRUCTURED GRIDS

Erkan Okuyan
M.S. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
January, 2009

The focus of this work is on parallel volume rendering applications in which renderings with different parameters are successively repeated over the same dataset. The only reason for inter-task interaction is the existence of data primitives that are inputs to several tasks. Both computational structure and expected task execution times may change during successive rendering instances. Change in computational structure means change in the data primitive requirements of tasks. Since the individual processors of a parallel system have a limited storage capacity, we can reserve a limited amount of storage for holding replicas at each processor. For the parallelization of a particular rendering instance, the remapping model should utilize the replication pattern of the previous rendering instance(s) for reducing the communication overhead due to the data replication requirement of the current rendering instance.

We propose a two-phase model for solving this problem. The hypergraph-partitioning-based model proposed for the first phase aims to minimize the total message volume that will be incurred due to the replication/migration of input data while maintaining balance on computational and receive-volume loads of processors. The network-flow-based model proposed for the second phase aims to minimize the maximum message volume handled by processors via utilizing the flexibility in assigning send-communication tasks to processors, which is introduced by data replication. The validity of our proposed model is verified on image-space parallelization of a direct volume rendering algorithm.

Keywords: parallel direct volume rendering, hypergraph partitioning, data replication, network flow, image-space parallelization.

ÖZET

DÜZENSİZ IZGARALARDA GÖRÜNTÜ-UZAYI PARALEL HACİM GÖRÜNTÜLEME İÇİN İLETİŞİM YÜKÜ EŞİTLEMEDE KOPYALANMIŞ VERİDEN FAYDALANMA

Erkan Okuyan
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Ocak, 2009

Bu çalışmanın hedef kullanım alanı, görüntülemelerin değişik parametreler ile aynı veri kümesi üzerinde defalarca tekrarlandığı hacim görüntüleme uygulamalarıdır. Görevler arası etkileşimin tek sebebi birden fazla görev için girdi olan veri primitiflerinin bulunmasıdır. Hem hesapsal yapı hem de görevlerin tahmini bitiş süreleri ardışık görüntüleme aşamalarında değişebilirler. Hesapsal yapıdaki değişim, görevlerin veri girdi gereksinimlerindeki değişimi ifade eder. Paralel sistemdeki her bir işlemcinin sınırlı bellek kapasitesi olduğundan, sınırlı miktarda saklama alanını kopyaların saklanması için ayırabiliriz. Eldeki görüntüleme aşamasının paralelleştirilmesi için, dağıtım modelinin daha önceki görüntüleme aşamalarının kopya dağıtım yapısından, eldeki görüntüleme aşamasının gerektirdiği haberleşme gereksinimlerini azaltmak için, yararlanması gereklidir.

Bu problemin çözümü için iki aşamalı bir model öneriyoruz. İlk aşama için önerilen hiperçizge parçalama temelli modelin amacı girdi verilerin kopyalanması/taşınmasından kaynaklanan toplam haberleşme hacmini asgariye indirirken işlemcilerin hesapsal dengelerini ve girdi alış yükleri arası dengelerini korumaktır. İkinci aşama için önerilen ağ akışı temelli model, işlemcilerin ele aldığı en büyük mesaj hacmini, işlemcileri gönderi görevleri ile ilgili görevlendirme konusunda veri kopyalama sonucu oluşmuş esnekliği kullanarak, asgariye indirmeyi amaçlar. Önerilen modelin geçerliliği doğrudan hacim görüntüleme algoritmasının görüntü-uzayı paralelleştirilmesi aracılığı ile doğrulanmıştır.

Anahtar sözcükler: paralel doğrudan hacim görüntüleme, hiperçizge parçalama, veri kopyalama, ağ akışı, görüntü-uzayı paralelleştirme.

Acknowledgement

I would like to express my gratitude to my thesis supervisor Prof. Dr. Cevdet Aykanat for his encouragement, support and guidance throughout the development of this thesis.

I am grateful to Assoc. Prof. Dr. Uğur Güdükbay and Assoc. Prof. Dr. Ezhan Karaşan for reading and commenting on the thesis.

I thankful to Tayfun Küçükyılmaz, Enver Kayaaslan and Erhan Okuyan for taking their time to review the draft copy. I thank Tayfun Küçükyılmaz and Ata Türk for their comments and suggestions.

Finally, I would like to thank my parents Gülseren and Mehmet for their invaluable support and trust in me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the massively parallel computer systems of modern era, physical simulations in many disciplines started to gain more importance. In these simulations, there is vast amount of input and many constraints to consider (such as physical laws). Solving these kinds of complex systems analytically is nearly impossible. Although usage of optimization techniques elevates this problem to some degree still there is vast amounts of computation, which necessitates parallel systems. Usage of powerful parallel systems for computationally heavy problems produce vast amount of numeric data, which in essence is not understandable with its' most basic form: numbers. So scientists needs visual analysis tools of these numeric data to have better understanding of the problem at hand.

The main aim of this thesis is to present faster methods for volume rendering. To further narrow the scope of the thesis we can comment thesis focuses on parallel volume rendering algorithms. Volume rendering is simply defined as producing 2-D projection of a 3-D space data. With ever growing sizes of 3-D space data, faster methods for rendering and possible use of hardware become necessary. In this thesis we will introduce some improvements, mainly for parallel processing environments, over existing rendering methods. Although we had to choose a specific type of algorithm and method to show validity of our contributions, we feel applying proposed methods for other type of rendering methods and other applications is possible and helpful.

1.1 Scientific Visualization

Scientific visualization is a field of research that produces representations of data for better understanding and insight. Growing sizes of data produced by computer systems and inability of humans to fully understand the raw data as it is makes scientific visualization a hot topic for research. Human brain is especially well adapted to perceive data presented via images or sounds. So representations produced by scientific visualization methods usually are images and sounds. Scientific visualization is important for scientists because scientific research both produce data and is affected by the produced data. Usually scientists produce data via simulations. Next steps are understanding, analyzing the data and drawing conclusions from it. Then they need to implement corrections or change parameters of simulation, induced by the conclusions drawn in previous step. Scientific visualization eases the process of understanding and analyzing the data so accelerates the process of scientific research. For instance a scientist working on an airplane wing will need to do simulations and calculate the amount of stress on different parts of the wing. With help of a visualization tool, scientist will easily identify the problematic areas and take precautions. However, in most cases data to visualize has 3 (or more) dimensions which makes production of understandable and helpful representations difficult. Sub-field of scientific visulization, which visualization 3-D spatial data is called Volume Rendering.

1.1.1 Volume Rendering

Volume Rendering can be defined as the process of generating a representative 2-D image of a 3-D volumetric dataset. Volumetric datasets are difficult in nature for human beings to understand easily without any representation extraction. So good representations of 3-D datasets let scientists and engineers gain better understanding of the dataset at hand. Achieving interactive speeds are important for volume rendering tools since faster visualization tools enables scientists to deliver feedback to data generation algorithms quickly thus increasing the effectivity of research. However, we can't say volume rendering studies are at a level

Cartesian		
Non-Cartesian	Regular	
	Irregular	Rectilinear
		Curvilinear
		Unstructured

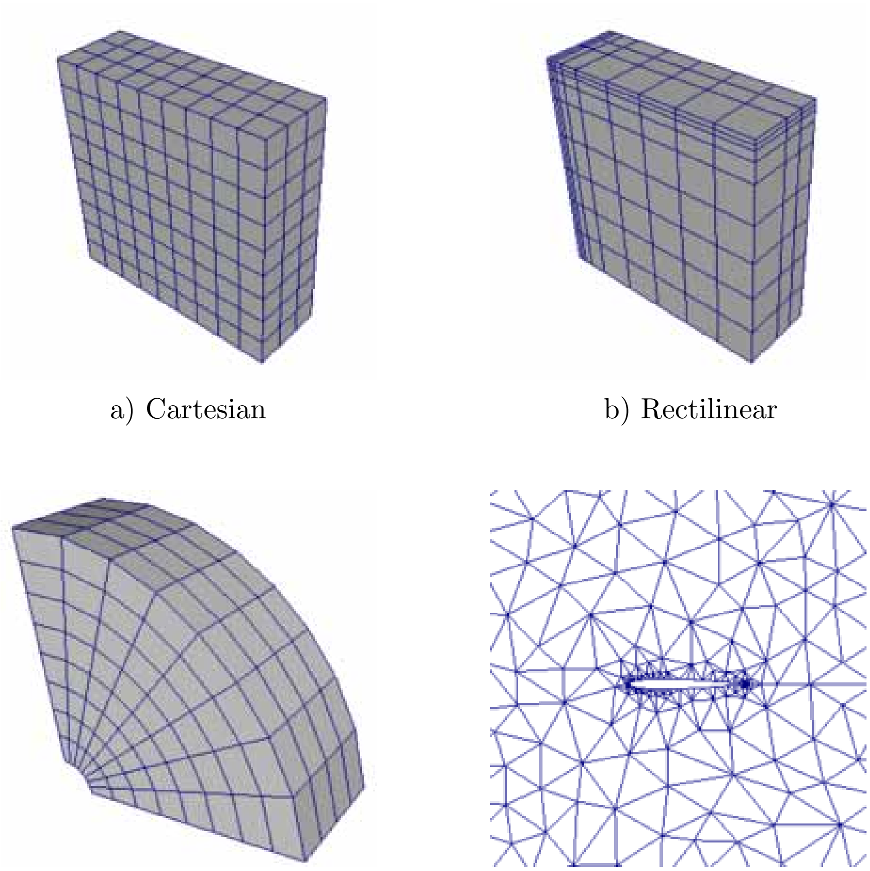
Table 1.1: Grid Types

that interactive speeds are achievable for big datasets and detailed visualizations.

There are several types of grids that are used to represent volumetric datasets. Most known and used ones are cartesian grids, regular grids, rectilinear grids, curvilinear grids and unstructured grids. Table 1.1 shows the relation between these grids and Figure 1.1 shows sample images of these grid types.

Cartesian grids are axis aligned and points of the grid are evenly spaced in each direction. Basically they are formed from uniform cubes. Regular grids are cartesian grids with a single difference; they are formed from rectangular prisms. Rectilinear grids have similar structure to regular grids but in this type of grids volumetric primitives do not have to be evenly spaced. Curvilinear grids has similar structure to rectilinear grids but unlike above, curvilinear grids are not axis aligned. So, as opposed to rectilinear grids, for curvilinear grids points may have different values in all three of the dimensions with neighbor points. Volumetric primitive used in curvilinear grids are non-uniform hexahedra. Unstructured grids have no implicit structure about connectivity of volumetric primitives. Several types can be used as volumetric primitives: tetrahedra, hexahedra, etc. But every primitive can be decomposed to tetrahedrals so usage of single type of volumetric primitive is possible.

Regular grids present us with a very basic and easy to work decomposition of the volume but they don't represent the data very effectively for some datasets. So, irregular grids are most reasonable choice for most applications. Unstructured grids have no implicit connectivity information but they have most flexibility to represent dataset with high effectiveness. And it is possible to explicitly present the connectivity information for unstructured grids and use the most effective



a) Cartesian

b) Rectilinear

c) Curvilinear

d) Unstructured

Figure 1.1: Grid Types [48]

representation available. In this work, we use unstructured grids with tetrahedrals as volumetric primitives and we explicitly present connectivity information to the system.

Inputs of volume rendering process are: a set of volumetric primitives defined by the grid imposed, viewing position and orientation. Viewing position and orientation define the image planes position in coordinate system and volume rendering algorithm calculates the contributions of volumetric primitives to pixels. Final values of pixels in image plane will correspond to pixel values of desired image. To calculate the contributions of volumetric primitives to pixels in image plane, every volumetric primitive should have some data associated with it. At this point there are two choices about where to store the data: Volumetric primitives will have data associated with it as a whole, points of volumetric primitives have the data and some interpolation of these values will be effecting the final color. In this work we store the data with points. Another discussion would be the type of data stored. For different purposes different types of data should be used. For example, if the data to visualize shows temperature values in a volume a single scalar will be enough for visualization purposes. If both the magnitude and direction is important in dataset as in physical experiments pertaining magnetic fields or velocity, vectors will be suitable as data. Or in the case of a stress test a tensor should be more appropriate than other data types. Some other visualization experiment may require different types also. In our work, single scalar seemed enough for our purposes. We associate a single scalar with every point defined in 3-D coordinate system.

Volumetric primitives are called voxels and in some research they are referred to as cells. In this thesis we will use the terms voxels and cells interchangeably.

1.1.2 Direct Volume Rendering

There are two types of volume rendering method [44]: Direct Volume Rendering (DVR) and indirect volume rendering. DVR methods are characterized by direct mapping between image space primitives and volumetric data so no intermediate representation, such as polygons and surfaces, is used. DVR methods are more suited to application where no apparent surfaces are present or of no interest. So visualization of cloud, fluid and gas like structures usually use DVR algorithms. Since direct mapping between image space primitives and volumetric data is present change in viewing parameter causes traversal of whole dataset for rendering. Indirect volume rendering methods are characterized by using surface primitives for rendering purposes. Execution starts with user defining a threshold value and cells interacting with surface primitive according to this threshold value are considered for extracting surface representation of volume data. Extracted intermediate representations can be rendered using standard graphics pipeline in a fast manner. Indirect volume rendering methods are more suited to application where apparent surfaces are present and are the interest. Such applications mainly arise in medical imaging where organ/tumor/vessel boundaries are needed. Furthermore, extraction of intermediate representation can be done as preprocessing in these methods so change in viewing parameters won't necessitate traversal of whole dataset but only rendering of intermediate representation. However, problems of false positives and false negatives may arise in indirect methods, mainly because intermediate representation only approximate the volume data. In case of false positives or false negatives scientist/doctor may incorrectly view small features of the dataset. Marching Cubes [43] is an well known indirect method, that has the problem of false positives and false negatives.

Generally indirect methods are considered faster than direct methods and direct methods are considered more accurate than indirect methods. Slower nature of direct volume rendering methods makes these methods a good candidate for parallelization. In this thesis we have worked on a parallel direct volume rendering algorithm.

1.1.3 Parallel Direct Volume Rendering

A common approach for parallel volume rendering has been transferring the dataset to a single graphics workstation from the source parallel machine and then rendering the dataset on a single machine. This is an undesirable approach because of the time and bandwidth spent for transferring the dataset, especially if the dataset is large. If the dataset is big in size, it is much preferable to render on a parallel machine instead of a single graphics workstation because of the memory constraints of the single machine. It is also beneficial to divide the workload among several processors.

High quality parallelization of direct volume rendering algorithms is a convenient method for achieving interactive speeds. Dataset sizes are growing and becoming fairly big for a single processor to handle with high performance especially for computationally heavy direct volume rendering methods. So parallel volume rendering became a hot topic of research. However, parallelization comes with some considerations:

- Efficient parallelization of rendering algorithms requires balanced distribution of work between processors.
- Decomposition and distribution of data if parallel processing environment is distributed-memory architecture.

Running the parallel volume rendering tools on the same parallel environment, which is used to produce simulation data, seems somewhat convenient. Each processor in parallel environment produces some simulation results and is supposed to store the produced data. So after simulation, there is initial data distribution on the system. Moreover, for many applications this data distribution is a high quality distribution since data produced by the same processor tends to be used together. After simulation, each processor has a set of data which has high probability to be used together. Meaning, data distribution produced by simulation allows easy assignment of rendering jobs to processors, because for each processor there is a partial rendering job that mainly uses the data assigned to the processor. This decreases the data replication on the system. Alos note that, data distribution may be improved to ensure the better balance in size or higher quality task-data coherency for some applications.

Coupling of visualization tools and scientific simulations on the same machine will be supported by two factors. First one is, increasing use of parallel machines for scientific simulations. Second factor is convenient data and task distribution for visualization enforced by data production pattern of simulation. This will allow scientists to run their computations with the feedback received from visualization tools in a fast manner. We believe the trend to combine scientific simulations and visualization tools on parallel machines will continue.

1.1.4 IS versus OS Parallelism

Parallelization of volume rendering algorithms can be done in two ways; Image-space parallelization (IS parallelization) and object-space parallelization (OS parallelization). In OS parallelization, decomposition of total workload is done in object space via assigning rendering operations of sub-volumes to processors. However, minimizing rendering load imbalance is important so that assignment of sub-volumes should be in a manner that rendering load imbalance is minimal. Furthermore, after rendering of sub-volumes compositing the intermediate results is needed in OS methods so that final pixel colors can be produced. Compositing stage may require excessive amount of communication, especially for highly irregular unstructured grids, and this high communication volume stands to be the main disadvantage for OS parallelization. Aykanat et al. [1], propose a graph partitioning based method utilizing OS parallelization for rendering problem which presents promising results.

In IS parallelization, decomposition of total workload is done in image space via assigning rendering operations of sub-screens to processors. So, a processor responsible for rendering sub-screen, only process the data projecting onto the sub-screen assigned to processor. If every processor stores the data needed for performing assigned rendering operation prior to rendering, no image compositing step is needed. However, in IS methods, achieving low load imbalance is important while avoiding excessive communication prior to rendering. Cambazoğlu and Aykanat [2] and Kutluca et al. [45] propose IS based methods which are taken as performance wise reference throughout this research.

1.2 Motivation and Contributions

Parallel volume rendering is a good approach for fast rendering; this way considerable speedups can be gained without sacrificing image quality. There are two kinds of architectures in which parallelism can be achieved; in shared memory architectures every processing unit can reach, read or write to global memory. Memory can be built as a single module or it may be divided between processor units. However, every memory operation affects the processors view of global memory. This heightens the usage of memory bus, such that the applications run over shared memory architectures are not very scalable because of memory bottlenecks. Because of scalability problems of shared memory architectures, parallel volume rendering on distributed memory architectures is a good choice especially for big datasets. Parallel volume rendering on distributed memory architectures requires

- the distribution of partial rendering jobs, and
- the communication of data imposed by job distribution.

Mentioned task repartitioning and data communication operations makes the following metrics important.

Work Load Imbalance: If every processor do not have equal amount of work, some processor finishes early and have idle time. Utilization of this idle time may increase the performance of the system therefore balancing workload is important.

Storage Volume: Every processor should be responsible for storing nearly equal amount of data. If some processor has high amount of data to store and limited memory, rendering tool may slow down or even crash. Furthermore, unequal assignment of store may cause imbalance on communication volumes of processors.

Communication Load Imbalance: High send and receive volume loads of processors may have adverse effect on system performance, via adversely effecting communication performance. In some systems maximum of send or receive volume of any processor has dominating effect and this maximum value should be decreased while in others total volume handled by any processor may have dominating effect. In general for distributed systems, one port and two port communication systems corresponds to above schema respectively.

Previous hypergraph based methods primarily reduces the total amount of communication while maintaining work load balance. This method indirectly reduces the send volumes and receive volumes of processors. However, in the trade off between high communication volume load of a processor and total communication volume in the system, hypergraph partitioning tools may have opt to produce a partition that has lower total communication volume but very high send or receive volume loads for a processor in the system. This fact may cause the application to work slower for some architectures. On the other hand, state of the art hypergraph partitioning tools PaToH and H-Metis presents the multi-constraint approaches so that while minimizing total volume of communication partitioning tool balances workload of processors, send volumes of processors and receive volumes of processors. However, this multi-constraint approaches are not present with fixed vertex formulation to the best of our knowledge. Thus, fixed vertex model based formulation for remapping under data replication does not use multi-constraints. Balance on volume loads of processors is achieved by proposed network flow based method and heuristics.

Radical increase in computation power presented by GPUs and usage of this power in GPU clusters makes the communication step in volume rendering methods a bottleneck. This thesis mainly focuses on communication patterns for volume renderers and general trend in technology increases the importance of methods presented in this thesis. The contributions in this thesis are as follows:

- Incorporation of replication to parallel rendering algorithm and proof of HP model with multiple fixed vertices per net model to encode the minimization of actual total communication volume under replicated data.
- Adaption and extention of [4] to balance communication volumes loads of processors.
- Proposal of a network flow based solution for data deletion to increase flexibility in the system.
- Proposal of INSH (inverse net size heuristic) to balance the receive volume loads of processors.
- Implemention and experimention of these general algorithms for a real volume rendering application.

1.3 List of Symbols and Abbreviations

$\mathcal{P}$	Set of processors
K	Number of processors
$\mathcal{R}$	Rendering application
$\mathcal{PB}$	Set of pixel blocks to be rendered
$time(pb_i)$	Rendering time of pixel block i
$Input(pb_i)$	Set of clusters needed to render pixel block i
$\mathcal{D}$	Set of clusters
$size(C_j)$	Size of storage used to store cluster j
$PixelBlock(C_j)$	Set of pixel blocks that need cluster j
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	Clusterization graph
$w_{\mathcal{G}}(v_i)$	Weight of ith vertex of clusterization graph
$w_{\mathcal{G}}(e_{ij})$	Cost of edge (i,j) of clusterization graph
$Home(C_j)$	The processor of cluster j using single home approach
$\mathcal{H}_{\mathcal{I}}$	Interaction hypergraph - without replication
$w(v_i)$	Weight of vertex i in $\mathcal{H}_{\mathcal{I}}$
$c(n_j)$	Cost of net j in $\mathcal{H}_{\mathcal{I}}$
Π	K way vertex partition of $\mathcal{H}_{\mathcal{I}}$
W_k	Total weight of vertices in part k induced by Π
$\Lambda(n_j)$	Set of parts that connects net j
$\lambda(n_j)$	Number of parts that connects net j
$TotalCommVol(\Pi)$	Total communication induced by Π
$\mathcal{I}$	Set of successive rendering instances
$WS^h(P_k)$	Working set of processor k at rendering instance h
$eRS^{h-1}(P_k)$	Set of clusters that is stored by processor k prior to $\mathcal{I}^h$
$eHome^h(C_j)$	Set of processors that stores cluster j prior to $\mathcal{I}^h$

$rc(C_j)$	replication count of cluster j
M	Memory limit of processors
$\mathcal{H}_{\mathcal{R}}$	Remapping hypergraph - with replication
$\Psi(n_j)$	Set of parts connected by net j through processor vertices
$\psi(n_j)$	Number of parts connected by net j through processor vertices
$INSH$	Inverse net size heuristic
$RecvSet(C_j)$	Set of processors to receive cluster j
$SendSet(C_j)$	Set of processors that can send cluster j
$Recv(\mathcal{D})$	Receive requirement pattern
$SendFlex(\mathcal{D})$	Send flexibility pattern
$RecvCnt(P_k)$	Number of clusters to be received by processor k
$SendCnt(P_k)$	Number of clusters to be sent by processor k
$\mathcal{F}$	Assignment flow network
$\mathcal{P}_R$	Processors vertices of F that will involve in receive operations
$\mathcal{P}_S$	Processors vertices of F that may involve in send operations
$\mathcal{U}_D$	Cluster vertices of F
B	Upperbound on maximum send/total volume handled by a processor
Ξ	Complete communication-task-to-processor assignment
$MaxSendCnt(\Xi)$	Maximum number of clusters sent by a processor
W_T	Number of total send operations
W_r	Number of failed send operations for a failed probe
$SatCnt$	Number of saturated terminal edges for a failed probe
$MaxTotalCnt(\Xi)$	Maximum number of clusters sent and received by a processor
$\mathcal{F}_{cont}$	Replica contraction network
$\mathcal{P}_{cont}$	Processor vertices of replica contraction network
$\mathcal{D}_{cont}$	Cluster vertices of replica contraction network
$\mathcal{E}_{cont}$	Edge set of replica contraction network
B_{cont}	Lower bound on replication count of minimally replicated cluster

Chapter 2

Background and Previous Work

In this chapter we will give details about previous work on volume rendering in three stages. In the first stage will give design options on volume rendering in an organized way with related previous work. In the second stage we will give details about some important publications on volume rendering in a less organized way. In the third part we will give details about some publication that are closely related to this thesis and we will discuss the relation between in detail.

There are several methods on how to achieve volume rendering. Throughout this stage we will demonstrate design options in a well classified manner. But in some cases there are possible hybrid options. In general we are not inclined to detail these options, but we will mention about some well studied hybrid methods. Furthermore, topics and design options given in this section has main concern of general rendering system performance in terms of interactivity and throughput.

Most basic decision option on design of a volume renderer can be said as direct [1][2][45]/indirect [8][18] volume rendering. As mentioned above direct volume rendering methods treats the volume as a whole while indirect methods extracts intermediate representations like polygons which usually rendered by graphics hardware.

Another decision option on design of a volume renderer would be the decision on serial [6][10][13][35][41] or parallel [1][2][7][8][19][33][34][45] volume rendering. From its earliest stages, volume rendering was defined as a computation intensive operation. Also nature of the work suggests that renderers should produce images at interactive rates (at least 1 image/second), which also hightens the stakes performance wise. If we consider the general trend on dataset size growth vs. hardware improvements stated in [35], it is safe to say that for the next decade, the status of volume rendering as a computational intensive operation will not change. There are several serial methods proposed for fast volume rendering. However, radical increases in data sizes and limited memory for serial algorithms forced researchers to consider parallel rendering methods. Most notable benefit of a parallel volume rendering method is the use of better processing-unit-power and memory of underlying architecture, especially for bigger datasets but parallel volume renderers can not use the spatial coherency in data as good as serial algorithms. Generally parallel volume renderers perform better on bigger datasets while serial volume renderers perform better on smaller data sets. Additionally parallel volume renders will have a cost disadvantage because of higher cost of underlying parallel architecture.

Assuming a parallel renderer is being designed, there are 3 more options on how to design a parallel renderer. Molnar et al. [17] present us with a good and well accepted classification of parallel renderers. Proposed 3 options are: sort-first, sort-middle, sort-last. Parallel rendering consists of 2 main parts: geometry processing (transformation, clipping, lighting, etc.) and rasterization (scan-conversion, shading, visibility determination, etc.). Nature of parallel renderers requires communication of data at some point of execution which is called sorting. Classification is done according to where the sorting occurs from object space to image space. If sorting is done at the beginning of execution with raw data in a fashion that every processor receives necessary data to complete geometry processing and rasterization without any further communication, it is called sort-first. If sorting takes place after geometry processing but before rasterization it is called sort-middle. In sort-middle systems, transformed data (according to viewing parameters) is communicated instead of raw data. The last class of sort-last algorithms aims to defer the communication step as much as possible and pixel data is communicated after rasterization stage takes place.

Without getting into much detail we should mention about advantages and disadvantages about each approach related to this thesis. Sort-first approaches have the advantage of exploiting frame-to-frame coherence where several viewing stages with little changes in viewing parameters take place. In these situations initial distribution of dataset among processors can be changed in an organized fashion which will decrease the communication time. Sort-middle approaches can do the same but it requires a clever implementation. Sort-last approaches do not have the ability to exploit such frame-to-frame coherence. Sort-first and sort-middle approaches are generally susceptible to load imbalance mainly because calculation for some pixels may require much more data than others. And in some cases, there are not as good balanced solutions as sort-last approaches. However, there are some work that have achieved very good load-balance between processors, hypergraph partitioning based remapping model proposed by Aykanat and Cambazoğlu [2] being one of them. Still sort-first and sort-middle approaches may cause imbalances between processors in receive volume size and storage volume sizes, but we will go into detail for them in later stages of this thesis. Sort-last approaches are less prone to load imbalances but pixel traffic at the communication step may be extremely high, particularly when oversampling.

Further classification can be done as image-space rendering methods and object-space rendering methods. In general, sort-first and sort-middle approaches tend to couple with image-space algorithms while sort-last methods uses object-space iteration model. This thesis is based on image-space methods while sort-first being the specific method.

Use of additional hardware is another option on design of volume renderers. While purely software-based methods are capable of volume rendering, use of additional hardware may improve the speed of volume renderer greatly. The reason for that is, hardware based methods benefits from the pipeline of the graphics hardware. Keeping in mind that using additional hardware has cost disadvantages, there are several promising works that utilize graphics hardware and obtain good results. There are 2 options on using hardware: Using special hardware or using commodity graphics accelerators. Two examples of special hardware are SGI's InfiniteReality and UNC's PixelFlow machines. These type of hardware solutions are usually rather costly compared to using standard graphics accelerators. Although there are important works for both types of hardware usage, using commodity graphics accelerators is the more popular choice for the last decade. Reason of this is the increasing programmable nature GPUs and the radical increase in graphics card speed of the last decade (faster than Moore's law).

Vilanova and Ruijters [24] discusses various bottlenecks about GPU-based volume rendering. Most important two of them being: Limited bus speed between system memory and GPU memory and texture memory limitation of graphics hardware. They propose a novel approach to relieve the problems on determined bottleneck areas for a GPU-based rendering system. However, they don't discuss that some of the bottlenecks found in the GPU-based systems can be effectively addressed by GPU clusters.

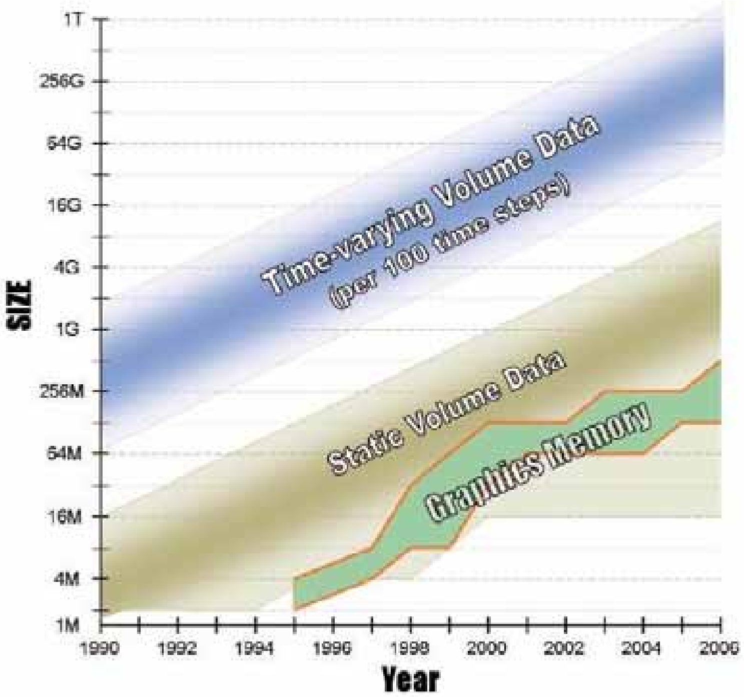


Figure 2.1: Texture Memory Problem [35]

Figure 2.1 graphically represents the memory problem. As seen in the figure dataset sizes are much bigger than texture memory sizes of commodity off-the-shelf graphics cards. Although GPU based rendering methods are fast alternatives, capabilities of a single GPU is far from interactive rates just because of the limited texture memory for larger datasets. GPU clusters present much more flexibility, much larger computation power and larger overall system texture memory. Therefore GPU clusters seem to be the viable option for interactive rendering for now.

For the second stage important previous publications are given: Roettger et al. [6] propose an important work in the sense that many optimization techniques proposed earlier has been collected together. This paper employs techniques like slicing approach, where several slices parallel to image plane is formed for rendering purposes, volumetric clipping, where parts in volume which are out of view frustrum is omitted, and advanced lightning approaches. Furthermore, in this work, capabilities of graphics hardware are used for efficient rendering of slices via an underlying ray casting approach. A problem of slicing approach, ring artifacts, is prevented via use of slabs instead of slices. Also it should be stated that this is a serial algorithm and for bigger datasets parallel algorithms with same capabilities may be of need. Ino et al. [7] propose a sort-last parallel approach with a new image compositing scheme. They show proposed algorithm especially behaves well for large number of processors.

Cullip and Neumann propose one of the pioneer work [15] and use trilinear interpolation hardware as an acceleration technique for volume rendering. In this work, necessary sampling schemes with object-space and image space sample planes are discussed. Using one of the special hardware Silicon Graphics RealityEngine workstations they prove the effectivity of hardware usage. They also acknowledges the problem, may be the biggest problem of hardware-based rendering, of limited texture memory. Texture memory is a limited source and for many datasets this memory is not enough to store all the dataset. So some sort of replacement policy is needed and dataset has to be brought to graphics memory part by part. Cabral et al. [16] also use 3D texture acceleration techniques and show availability of volume reconstruction along with volume rendering with help of 3D texture hardware. They show validity of their system on advanced medical problems. Krüger and Westermann [13] integrates standard acceleration techniques to standard graphics hardware implementations and shows the importance of such methods for interactive rendering. These techniques are: Early ray termination and empty space skipping. Both in [13] and [16] special hardware solutions are employed.

Müller et al. propose a GPU-based parallel rendering method [30]. Proposed algorithm is an ray casting based object space method. They propose a k-d tree based load balancing scheme and empty space skipping method to increase the performance of the system. For middle sized datasets they achieve interactive speeds on a GPU-cluster system with 8 rendering nodes. However, concept of bricking (volume data is divided into uniform texture bricks and each brick is treated as an object space primitive) is used which tends to increase the interaction between object-space primitives so causing swaps in and out of the memory. Work of Marchesin et al. [32] has similar structure. It is also a GPU-based sort-last parallel method where bricking techniques are employed along with k-d tree based load balancing. Main contribution of this work is dynamic load balancing algorithm on situations where initial data distribution causes imbalance in rendering load. Such situations arise in level-of-detail methods and/or zooming. For that a hierarchical cache structure is used where video ram, system ram and system disk forms the hierarchy. Assumption of whole dataset replication in every processing node is the biggest weakness of this work. Cavin et al. [36] also employ the use of high end graphics cards on commodity off-the-shelf PC clusters. In this work, implementation allows fully overlapped CPU(s), GPU(s), and network usage all along the execution. Samanta et al. [18] present us with a dynamic sort-first/sort-last parallel hybrid method with polygon rendering. The main idea is to dynamically partition the 2D screen into tiles and partition 3D primitives into groups in order to balance rendering loads via 3D grouping and balance communication load required for compositing tiles via 2-D partitioning. Furthermore, system employs the use of commodity graphics accelerators for further performance and authors presents some reasoning for sort-first usage instead of sort-middle with graphics clusters. Acknowledged shortcomings of system such as client-side bottleneck and dynamic data management overhead is labeled as, likely to diminish with net trends in technology. Bachthaler et al. [31] also propose a GPU-based parallel sort-first/sort-last hybrid method. The main topic is texture-based visualization of flow fields on curved surfaces. This work is similar in nature to work of Samanta et al. [18]. A bricking technique for 3-D data partitioning is used for overcoming texture memory problem and a dynamic 2-D partitioning scheme is used for work load balancing. Garcia and Shen propose a hybrid image-space/object-space method [19] and there are two modes of implementation, one using texturing hardware and one is software-based. Necessary comparisons are given. Samanta et al. [9] propose a sort-first multi-projector rendering system. We think this is a promising work because it employs the best possible techniques for interactive volume rendering with a group of hardest parameters. Most notable one of these parameters is very big image sizes (over 6 million pixels). For interactive rendering, load balancing methods is proposed, usage of high speed GPUs and a parallel architecture is used. Validity of image-space methods for big image sizes and big datasets has been shown.

One other type of work for volume rendering is level-of-detail volume rendering. The idea in level-of-detail volume rendering is to render the region of interest in greater detail and other parts in less detail. Lamar et al. [12] use an octree to decompose 3-D data and form several levels of coarser 3-D representation to exploit the benefits of level-of-detail volume rendering. At the time of rendering, according to users interest to a region, system may render data in greater detail or may opt to render in lesser detail or not at all. Rendering in less detail is faster so overall system performance increases with small loss of detail, especially in point of interest. Artifacts introduced, especially when rendering two different but adjacent detail levels, is the biggest problem of level-of-detail rendering. A method that utilizes spherical shell geometry is introduced and low-resolution data is processed with a different transfer function in order to minimize the artifact introduced. Work of Weiler et al. [11] is similar in nature, but a hierarchy is introduced to effectively interpolate between detail levels to reduce the artifacts. This method uses slicing method, which is more allowing for algorithms that reduce rendering artifacts of level-of-detail rendering. In this work data is divided into bricks which are then represented in lesser detail. To address artifacts arises for rendering two different but adjacent detail levels, boundary voxels of bricks are replicated which enables smooth transaction between level-of-details.

Another area of research in volume rendering via usage of GPUs aims to relieve the very limited texture memory problem. Fout and Ma [35] propose a asymmetric transform compression scheme which facilitates the real time decompression of data on GPU. Asymmetric nature of algorithm utilizes not so critical compression time at the time data is produced to facilitate real-time nature of volume rendering without sacrificing compression quality. Increasing the effectiveness of texture memory enables real-time volume rendering via GPU-based methods. However, compression methods did not start to take importance with the usage of GPUs. There is a group of research that exploits the usage of wavelets to increase the performance of the system, but main aim of these compression schemes always have been relieving the bottlenecks of communication or memory originating from data size. Work of Gao et al. [33] is such an example. The main idea in this work is to convert raw data into a multiresolution wavelet tree and design algorithms to partition the wavelet tree into pieces considering data dependencies and workload balance among processors. At run time wavelet tree is traversed according to a user specified error tolerance and then rendered after decompression. Guthe and Straßer [28] and Guthe et al. [14] use hierarchical wavelet representations and at rendering time necessary detail levels are decompressed and sent to texturing hardware. Typically 30:1 compression rates are reported without noticeable artifacts. Also usage of wavelet compression scheme with GPUs is an important contribution considering texture memory problem. Wang et al. propose an important method [34]. In this work, wavelet based methods are used in a multiresolution context over large-scale time-varying data sets which is big problem for rendering systems if interactivity is important. Methods proposed by Muraki [23], Kim and Shin [22] and Ihm and Park [21] are some earlier work on wavelets showing the validity of wavelet usage for managing, rendering and compressing 3D data.

Weiskopf et al. [10] propose a novel approach to maintain constant frame rates when using 3D textures when rendering. The main problem is the nature of modern graphics cards that have optimized texture cache for fast 2D texture retrieval. While using 3D textures have some benefits, like smaller memory usage and ease of implementation, slow retrieval of textures is a problem. For that they propose a bricking approach which then stored in 3D texture memory in different orientations. So different viewing conditions won't cause major change in rendering rates.

As far as we know, only Samanta et al. [8] investigated the effects of replication on volume rendering. Their contention is, with a replication amount of k << n (n being processor count) primitives over nodes in parallel machine, rendering performances closer to replication n is achievable while respecting memory constraints closer to replication 1. In this sense a subset of our work is very similar to this work. While they are extending their work from their previous work [18] and using a hybrid sort-first/sort-last and investigating the effects of replication, we are investigating the effects of replication on a purely image-space context. Furthermore, their replication pattern is not changing throughout visualization while our replication pattern changes on each iteration. Their contention is supported by our findings and benefit of replication (usage of excess memory) is established both in image-space and object-space environments.

Viola et al. [26] propose an importance driven volume rendering system, which remove or suppress less important parts of volumetric data in order to magnify the effects of more important volume parts. For that an importance metric is defined for volumetric objects where volume of interest has high importance and occluding objects have less importance. Although main idea of this work is to increase the usefulness of image produced, not rendering some parts of volume while increasing the quality of final image has performance benefits in terms of interactiveness. Work of Wang et al. [29] is somewhat related to previously mentioned importance driven volume rendering system, but extensions in this work also allows user to non-linearly magnify the high importance objects via defining volumetric lenses for better inspection. Performance benefits are similar in both of these works. Hadwiger et al. [25] use the explicit segmentation information and selectively enable objects of interest for detailed graphics hardware based rendering. Furthermore, within proposed method different transfer functions can be given to different objects and different rendering methods (direct volume rendering, iso-surfacing etc.) can be applied to different segments in dataset.

Work of Bordoloi and Shen [27] is an interesting work in the sense that they suggest a minimal set of viewing angles to user that captures the entire scene. For that several views are considered according to their representativeness of the scene and a clustering approach is applied. At the end, N best representing viewing angles are selected for user to render. This work aims to decrease the time spent for rendering purposes by doing some preprocessing and allow user to gain intuition about the whole volumetric data.

For the third stage, our work is closely related with three publications [2][4][8]. We develop our work by taking [2] as a basis. Cambazoğlu and Aykanat present a parallel image-space volume rendering method based on remapping with hypergraph partitioning to obtain good load balancing while minimizing total communication. Our work is very similar to [2] considering rendering algorithms and general structure. Also note that, our work extends and improves [2] in several ways. First, we introduce the replication of data to facilitate faster rendering, and prove hypergraph partitioning remapping method minimizes total communication under replication. A proposed heuristic to reduce maximum receive volume improves the system performance dramatically especially for very irregular data. In general we think, this work is a significant improvement over [2].

As far as we know [8] is the only work to investigate the effect of replication on volume rendering. However, they replicate the data in a static manner, where replication occurs prior to visualization and does not change throughout visualization. In our approach, dynamic replication is allowed where natural communication of dataset between processors is used for replication. So dynamic replication is achieved with zero communication cost. Furthermore, data coherency between spatially close pixel blocks can be exploited with this method, as opposed to [8], even though it is out of scope of this thesis. In this aspects, we think our work surpasses [8]. Also Samanta et al. present the results of replication on volume rendering on a hybrid indirect volume rendering context, whereas our work presents the effect of replication on a direct image-space parallel context.

Therefore, we think our work seems to be completing the findings of [8].

Final publication closely related to this thesis is [4]. In this work, a network flow based load balancing algorithm is presented. In this work tasks to assign to processors has the flexibility to be assigned to more than one processor. So a task-to-processor mapping can be found that balances the computation loads of processors. We have used and extended the proposed method for our communication needs.

Chapter 3

General HP-based IS Parallelization

3.1 Framework

The target parallel computing platform is a homogenous distributed-memory architecture. In this platform, there exists a set $\mathcal{P}=\{P_1, P_2, \ldots, P_K\}$ of K identical processors each having its own local memory. These processors are connected to each other via an interconnection network. Interprocessor coordination and communication is performed via message passing. A PC-cluster constitutes a typical example for the target parallel platform.

The rendering method to be parallelized is represented as a two tuple $\mathcal{R} = (\mathcal{PB}, \mathcal{D})$. Here $\mathcal{PB} = (pb_1, pb_2, \ldots, pb_n)$ represents the set of pixel blocks to be rendered. Pixel blocks can have different rendering times. The expected rendering time of pixel block pb_i is denoted as $time(pb_i)$. The dataset is divided into pairwise disjoint sub-parts called clusters where $\mathcal{D} = (C_1, C_2, \ldots, C_m)$. Rendering of each pixel block pb_i needs a subset of the data set, namely a set of clusters, as input. The set of clusters needed by rendering operation of pb_i is denoted as $Input(pb_i)$. There is no dependency between distinct pixel block rendering operations. The only reason for interaction between pixel block rendering operations is

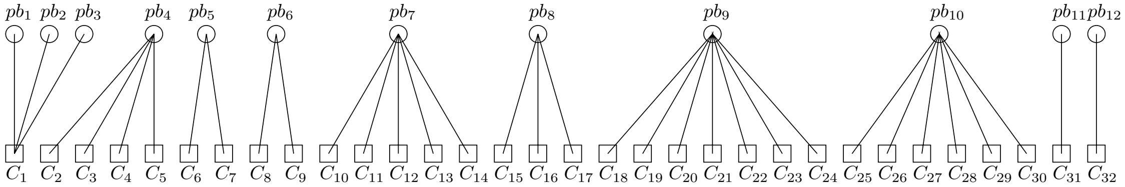

pb_1
pb_2
pb_3
pb_4
pb_5
pb_6
pb_7
pb_8
pb_9
pb_{10}
pb_{11}
pb_{12}
C_1 C_2 C_3 C_4 C_5 C_6 C_7 C_8 C_9 C_{10} C_{11} C_{12} C_{13} C_{14} C_{15} C_{16} C_{17} C_{18} C_{19} C_{20} C_{21} C_{22} C_{23} C_{24} C_{25} C_{26} C_{27} C_{28} C_{29} C_{30} C_{31} C_{32}

Figure 3.1: A rendering operation $\mathcal{R} = (\mathcal{PB}, \mathcal{D})$ with $\mid \mathcal{PB} \mid$ = 12 tasks and $\mid \mathcal{D} \mid$ = 32 clusters.

the existence of the clusters that are inputs to several rendering operations. The set of rendering operations that need cluster C_j is denoted as $PixelBlock(C_j)$. Clusters can have different sizes; the size of a cluster C_j is denoted as $size(C_j)$ and measured by the size of storage used to store C_j. Figure 3.1 shows the interaction between pixel block rendering operations and clusters, it illustrates a sample rendering with $n = 12$ pixel blocks and $m = 32$ clusters.

The focus of this work is parallel volume rendering applications in which similar type of renderings are successively repeated over the same dataset instance for many times with different parameters. So the overall rendering can be considered as a sequence of computations $\{\mathcal{R}^1 = (\mathcal{PB}^1, \mathcal{D}), \mathcal{PB}^2 = (\mathcal{PB}^2, \mathcal{D}), \ldots\}$ performed over the same dataset $\mathcal{D}$. Both computational structure and expected pixel block rendering times may change during successive viewing instances. Change in computational structure means change in the input set of pixel block rendering operation.

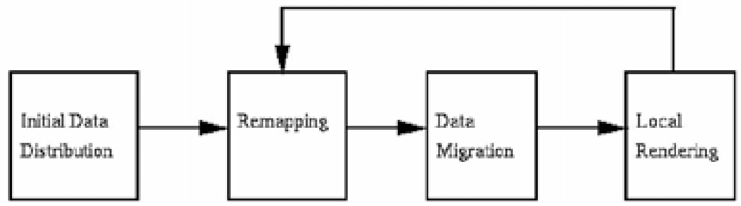


Figure 3.2: General Execution of Rendering Algorithm

Figure 3.2 shows the general execution of the rendering system. In the first step of initial data distribution, processors divide the dataset into clusters which then are assigned to processors thus each processor is holding a pairwise disjoint subset of the whole dataset. This step does not have to be repeated for every successive rendering iteration so this step is called view independent step. Then there is the remapping step in which assignment of rendering operations of pixel blocks to processors takes place. However assignment of pixel blocks pb_i to P_k may require communication for a subset of $Input(pb_i)$ by P_k. This subset consists of clusters that is required by rendering pb_i but have not stored by P_k. After data migration step every processor is free to render its' assigned sub-screen. This step produces the final image by communicating sub-screens to a final destination. Rendering may be redone with different parameters, starting from the step remapping.

3.2 Concepts in Image-Space Parallelization

Ray casting based rendering algorithms can be parallelized using both object-space methods and image-space methods. An image-space parallel method has been implemented for this work. However load imbalances between processors and data-to-processor mappings are important concepts for implementing an efficient image-space parallel method. In this chapter, we will explain some concepts regarding our image-space parallel method and give reasoning about some decisions made for performance before giving details about our main contribution in the next chapter. We urge reader to keep in mind that, concepts in this chapter, like clustering, screen subdivision, workload calculation etc., are used by both old HP-based, newly proposed HP-based and to some extent other rendering algorithms (Some sort of clustering, workload calculation and screen subdivision is necessary for image space parallelization of a volume rendering algorithm. Algorithms to achieve these may change however used concepts and reasoning to use them are same.).

3.2.1 Ray-Casting

Ray-casting is a method where color of a pixel is calculated with the help of a ray shot from the viewpoint through pixel into the volume. Final color is a product/composite of samples taken over the ray at some regular intervals until

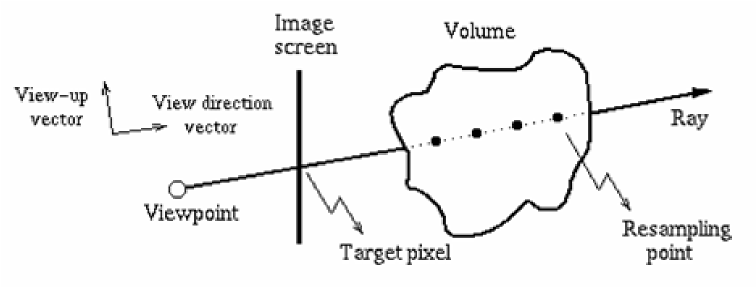


Figure 3.3: Ray-Casting [5]

the end of volume. If sample taken over the ray does not coincide with a grid vertex point then some interpolation of neighbor grid vertices is needed. Transfer functions are used at each resampling point in order to compute color and opacity contributions and each resampling point accumulates the color and opacity values according to some weighting scheme into the final color of the pixel. All pixels should be processed to get the final colors of all pixels. Figure 3.3 explains the process in more detail.

We have used Koyamada's ray-casting algorithm [42]. For more detail on implementation and optimizations please refer to [5] and [41].

3.2.2 Clustering

IS parallelization requires distribution of pixel rendering jobs among processors. To achieve load balance among processors in parallel system, initial calculation of work amount for rendering each pixel is needed. Rendering time for each pixel is directly proportional to ray face intersection count. For the most accurate calculation of loads for each pixel, every tetrahedral should be processed if it causes a ray-face intersection. However processing every tetrahedral individually produces unacceptable amount of preprocessing calculations therefore a clustering method is used.

Applying a global clustering method that takes the whole volume as input gives us more accurate results. However volumetric data usually produced by scientific simulations produces the data distributed among processors. Such clustering method requires communication to obtain better clustering of the data. We think applying a local clustering approach gives us good enough clustering quality while avoiding communication costs. In local clustering, every processing unit processes the data assigned to produce clusters.

Clusterization of data is important for two reasons. First, producing limited number of clusters is handy for bookkeeping purposes. Second and more importantly, formed clusters are the primitives to work load calculation process so clusters should be formed to ease work load calculation. To ease the work load calculation process, the idea of generating minimum surface area clusters is used. Using this idea produces sphere-like clusters therefore less scan-conversion is performed in work load calculations. Furthermore sphere-like clusters decreases the data dependency (same cluster usage) between pixels rendering tasks so, whatever task-to-processor assignment method is used, search space is increased for remapping.

One other requirement for clusterization can be stated as: Every cluster should be nearly equal in size. It is necessary because production a mix of very big and very small clusters tends to increase rendering time. Also it may incur an additional unnecessary communication because big clusters tend to be communicated much and receiving processor usually don't need to use all data in a big cluster.

3.2.2.1 Graph Partitioning

Graph partitioning is a method for grouping vertices of a graph into P parts while considering some predetermined constraints and optimizing an objective function. Its been known to be used in many areas, one being VLSI design. For clustering purposes a graph partitioning method has been used. In this partitioning, applied constraint corresponds to maintaining balance between clusters and objective function corresponds to total surface area of clusters.

Undirected graph $\mathcal{G}=(\mathcal{V},\mathcal{E})$ where $\mathcal{V}$ denote the set of tetrahedral cells and $\mathcal{E}$ denote the set of shared faces between tetrahedral cells has been partitioned, as

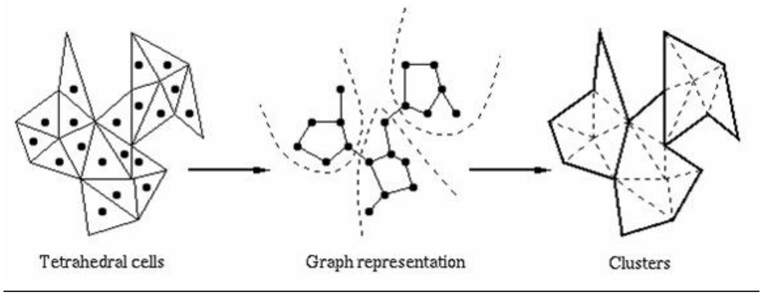


Figure 3.4: Cell clusterization using METIS

seen in Figure 3.4. For each $v_i \in \mathcal{V}$ there is an associated cost $w_{\mathcal{G}}(v_i)$ and for each $e_{ij} \in \mathcal{E}$ there is an associated cost $w_{\mathcal{G}}(e_{ij})$. A vertex v_i is connected to v_j with edge e_{ij} in $\mathcal{G}$ if there is a shared face between tetrahedrals denoted by v_i and v_j.

We have partitioned $\mathcal{V}$ into P pairwise disjoint nonempty sets

$$\mathcal{V} = \mathcal{V}(C_1) \cup \mathcal{V}(C_2) \cup \ldots \cup \mathcal{V}(C_P) \tag{3.1}$$

where vertices in each partition i corresponds to cluster C_i and $\mathcal{V}(C_i)$ corresponds to vertex set of C_i. Total weight of vertices belonging to C_i is defined as $W(C_i) = \sum_{v_i \in \mathcal{V}(C_i)} w_{\mathcal{G}}(v_i)$. Partitioning constraint enforces $W(C_1) \cong W(C_2) \cong \ldots \cong W(C_P)$ meaning every cluster has nearly the same size. In graph partitioning an edge is said to be cut if endpoints of the edge lies in different partitions and the set of all cut edges is denoted with $\mathcal{E}'$. Total weight of cut edges is defined as $W(\mathcal{E}') = \sum_{v_i \in \mathcal{V}(C_i) \& v_j \notin \mathcal{V}(C_i)} w_{\mathcal{G}}(e_{ij})$ and partitioning objective is to minimize $W(\mathcal{E}')$. Minimizing $W(\mathcal{E}')$ means, total interaction between clusters are minimized.

There are six possible weighting schemes proposed by Cambazoğlu [5] to determine vertex and edge weights of clusterization graph. It is given in Table 3.1 below. The symbols CV, CA, FA denotes cell volume, cell area and face area respectively.

We think usage of FA as edge weights is natural since it represents the interaction between tetrahedrals naturally. A ray leaving a cluster has a chance of entering another cluster directly proportional to shared face area, therefore we use FA weighting scheme for edge weights. Furthermore usage of FA scheme

$VertexWeight$	$EdgeWeight$
1	1
1	FA
CA	1
CA	FA
CV	1
CV	FA

Table 3.1: Cluster generation weighting schemes

also produces spherical shaped clusters, which does not cause drastic change in rendering time for changing viewing angles [10]. For the vertex weighting there are three possible options. Vertex weight one produces clusters nearly equal in storage size however rendering times of clusters can be different. Schemes CA and CV tend to produce clusters equal in rendering time but not storage size. Vertex weight CA and CV can be used if communication in parallel system is not the bottleneck area, vertex weight one should be the choice if prevention of very big clusters (in storage size) hence avoiding additional communication overhead is important. Findings of Cambazoğlu [5] supports our decisions. Furthermore proposed network flow approach needs clusters to have same storage size so we have chosen one as vertex weights. Clusters may be referred to as data primitives in this thesis since clusters are used as primitives to handle the dataset as opposed to data primitives of the actual dataset, tetradedral.

3.2.3 Load Balancing

Load balancing is an important issue for volume renderers. To achieve good load balance and avoid degrading performance a good screen subdivision algorithm is needed. Work load calculations are necessary to calculate the rendering times of pixels for assigning near equal amount of work to each processor. Furthermore communication operations of processors should be handled to avoid assigning great amount of network related operations to a single processor, in order to achieve better computational load balance.

3.2.3.1 Screen Subdivision

Image-space parallel volume renderers work by distributing pixels of desired image to processors for rendering purposes. Earlier work on screen subdivision includes quad-trees, recursive bisection and jagged partitioning. The shortcomings of these algorithms lie on the notion that sub-screens should be in the shape of a rectangle. However rectangular subsections reduce the freedom for assigning pixels to processors in an optimal manner. Therefore more flexible techniques such as [2] is needed. However calculation of pixel-to-processor assignment with actual image pixels causes too much preprocessing. Therefore a grouping method of actual pixels should be used for practicality purposes. In this work, screen is divided into nxn pieces and for further computation each pixel block has been treated as a pixel. Number of pixel blocks is a parameter for the program but an engineering formula has been proposed by Cambazoğlu [5] to determine the appropriate number of pixel blocks.

3.2.3.2 Workload Calculation

In order to achieve good load balance, work needed for a pixel block to be rendered should be estimated correctly. This is called workload calculation. In this work, we calculate the work needed to render a cluster with

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

where f denotes the face, $\mathcal{F}_{ff}^{C}$ denotes the set of front facing faces of cluster C and $Area(f)$ is the projection area of face f with current viewing parameters. $Load(C)$ is then distributed among pixel blocks that intersect with the projection area of the cluster C to find the work loads of pixel blocks.

3.2.4 Remapping

For an efficient parallelization, pixel blocks should be partitioned among processors in such a way that communication overhead is minimized while computational load balance is maintained. An initial high quality pixel-to-processor mapping may become a low quality mapping as the viewing parameters change. Therefore remapping of pixels to processors is needed as the viewing parameters change. As mentioned earlier this pixel-to-processor remapping should assign balanced loads of computations while minimizing the communication of the parallel system. This problem is NP-hard and HP-based method presents a high quality solution to remapping problem.

3.3 HP-based Remapping Solution without Replication

HP-based remapping problem without replication is addressed in the work of Cambazoglu and Aykanat [2]. In this work every processor is assigned to store a set of clusters, which are pairwise disjoint. Therefore no two processors store the same cluster at the start of the rendering instance. So notation $Home(C_j)$ is used for the one and only processor that is storing the cluster C_j. Furthermore $Home(C_j)$ is responsible for replicating cluster C_j in case the proposed remapping algorithm induces a partition that requires replication of C_j. At the end of rendering instance, if a processor P_k is storing a cluster C_j and $p_k \neq Home(C_j)$, processor P_k simply deletes C_j.

3.3.1 Introduction to Hypergraph Partitioning Problem

A hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$ consists of a set of vertices $\mathcal{V}$ and a set of nets $\mathcal{N}$. Each net n_j in $\mathcal{N}$ connects a subset of vertices in $\mathcal{V}$, which are said to be the pins of n_j. Let $Pins(n_j)$ denote the set of pins of net n_j. Let $Nets(v_i)$ denote the set of nets that connect vertex v_i. We extend the $Nets(\cdot)$ notation to a subset of vertices, that is $Nets(\mathcal{U}) = \bigcup_{v_i \in \mathcal{U}} Nets(v_i)$ for any $\mathcal{U} \subseteq \mathcal{V}$.

Each vertex v_i has a weight $w(v_i)$, and each net n_j has a cost $c(n_j)$. $\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ is a K-way vertex partition if each part $\mathcal{V}_k$ is non-empty, parts are pairwise disjoint, and the union of parts gives $\mathcal{V}$. In Π, a net is said to connect a part if it has at least one pin in that part. The connectivity set $\Lambda(n_j)$ of a net n_j is the set of parts connected by n_j. The connectivity $\lambda(n_j) = |\Lambda(n_j)|$ of a net n_j is equal to the number of parts connected by n_j. If $\lambda(n_j) = 1$, then n_j is an internal net. If $\lambda(n_j) > 1$, then n_j is an external net and is said to be at cut. In Π, the weight W_k of a part $\mathcal{V}_k$ is equal to the sum of the weights of vertices in $\mathcal{V}_k$, i.e.,

$$W_k = \sum_{v_i \in \mathcal{V}_k} w(v_i). \tag{3.3}$$

The K-way hypergraph partitioning problem is defined as finding a vertex partition Π for a given hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ such that part weights are balanced while a cost defined on nets is optimized. In this work, the connectivity-1 metric

$$\text{Cutsize}(\Pi) = \sum_{n_j \in \mathcal{N}} c(n_j)(\lambda(n_j) - 1) \tag{3.4}$$

is used as the cost to be minimized. In this metric, which is frequently used in VLSI [51] and recently used in scientific computing [52][53][54] communities, each net n_j contributes $c(n_j)(\lambda(n_j) - 1)$ to the cost $\chi(\Pi)$ of a partition Π.

3.3.2 HP-based Remapping Model

In this section, we show that pixel-block-to-processor assignment problem, in order to achieve remapping, can be described as a K-way hypergraph partitioning with fixed vertices. In this model, the computational structure of the rendering instance is represented as an interaction hypergraph $\mathcal{H}_{\mathcal{I}} = (\mathcal{V}, \mathcal{N})$, where

$\mathcal{V} = \mathcal{PB} \cup \mathcal{P}$ and $\mathcal{N} = \mathcal{D}$. That is, nets represent clusters while vertices represent pixel blocks and processors such that there exists a vertex v_i for each pixel block pb_i, there exists a vertex p_k for each processor P_k and there exist a net n_j for each cluster C_j. Net n_j connects the set of vertices which represent the pixel blocks that need cluster C_j as input and the processor $Home(C_j)$ that is asigned to store the cluster C_j.

$$Pins(n_j) = \{pb_i : C_j \in Input(pb_i)\} \cup Home(C_j) \qquad (3.5)$$

In other words

$$\begin{aligned} Nets(v_i) &= \{C_j : C_j \in Input(pb_i)\} && (3.6) \\ Nets(p_k) &= \{C_j : P_k = Home(C_j)\}. && (3.7) \end{aligned}$$

The expected rendering time of each pixel block is assigned as the weight of the respected vertex, i.e., $w(v_i) = time(pb_i)$. Weight of processor vertex p_k is assigned zero. The size of each data primitive is assigned as the cost of the respected net, i.e., $c(n_j) = size(C_j)$. Here, $size(C_j)$ is assumed to be the number of bytes needed to store or send the data primitive C_j.

Consider a K-way vertex partition $\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ of $\mathcal{H}_I$. Note that each part $\mathcal{V}_k$ contains a single fixed vertex, which is processor vertex p_k, due to the partitioning with K-fixed vertices model. So partition Π is decoded as assigning the set of pixel blocks corresponding to the set of pixel block vertices in part $\mathcal{V}_k$ to processor P_k. So Π will be used interchangeably to refer to vertex partition of $\mathcal{H}_\mathcal{I}$ and pixel block-to-processor mapping for for the current rendering instance. That is, vertex partition Π will denote the pixel block-to-processor mapping $\{\mathcal{PB}_1, \mathcal{PB}_2, \ldots, \mathcal{PB}_K\}$, where $\mathcal{PB}_k = \{pb_i : pb_i \in \mathcal{V}_k\}$.

A processor P_k can complete the rendering of pixel block pb_i, which is assigned to P_k by vertex partition Π, only if all clusters in $Input(pb_i)$ are replicated in the local memory of P_k. Here we will refer to the set of clusters needed by a processor

P_k as its working set $WS(P_k)$, where

$$WS(P_k) = \bigcup_{pb_i \in \mathcal{PB}_k} Input(pb_i) = \{C_j : \mid PixelBlock(C_j) \cap \mathcal{PB}_k \mid \neq 0\} \tag{3.8}$$

So all clusters in $WS(P_k)$ should be replicated in processor P_k for P_k to complete its assigned workload $\mathcal{PB}_k$.

K-way vertex partition Π of $\mathcal{H}_\mathcal{I}$ induces a pixel block-to-processor mapping. Since vertices in part $\mathcal{V}_k$ of Π correspond to the pixel blocks assigned to processor P_k for rendering, the weight W_k of $\mathcal{V}_k$ corresponds to the total expected rendering time for processor P_k. So the partitioning constraint corresponds to maintaining computational load balance among the processors.

Any part $\mathcal{V}_k$ in $\Lambda(n_j) - Home(C_j)$ corresponds to processor P_k that needs C_j but does not hold a replica of C_j. So the set $\Lambda(n_j) - Home(C_j)$ of parts corresponds to the set of processors that should receive a replica of C_j for the current rendering instance. That is, data primitive C_j should be communicated $|\Lambda(n_j) - Home(C_j)| = \lambda(n_j) - 1$ times, because every clusters has only one home processor an only stored once. Hence, total communication volume incurred by Π will be

$$\begin{aligned}
TotalCommVol(\Pi) &= \sum_{n_j \in \mathcal{N}} (\lambda(n_j) - 1)c(n_j) && (3.9)\\
&= \sum_{n_j \in \mathcal{N}} \lambda(n_j)c(n_j) - \sum_{n_j \in \mathcal{N}} c(n_j) && (3.10)\\
&= \sum_{n_j \in \mathcal{N}} (\lambda(n_j) - 1)c(n_j) + \sum_{n_j \in \mathcal{N}} c(n_j) - \sum_{n_j \in \mathcal{N}} c(n_j) && (3.11)\\
&= Cutsize(\Pi) && (3.12)
\end{aligned}$$

In (3.12), the partitioning objective of minimizing the cutsize according to the connectivity-1 metric given in (3.4) corresponds to minimizing the total volume of communication. Note that, $Cutsize(\Pi) = TotalCommVol(\Pi)$ is the exact volume of communication on the parallel system.

Chapter 4

HP-based IS Parallelization under Replication

This chapter focuses on modifications of general HP-based image-space parallelization to facilitate use of data replication for better performance. Volume rendering applications are a good candidate for make use of rendering because similar type of renderings are successively repeated over the same dataset instance for many times with different parameters. During successive instances of rendering data migration, thus data replication, naturally occurs to facilitate rendering.

As opposed to previous single home processor approach for general HP-based image-space parallelization, this work makes use of replication and extends the home processor idea to support more than one home processor for each cluster. Therefore at the beginning of each instance clusters are allowed to be stored in more than one processor, thus decreasing the total communication volume. Furthermore, with the use of replication, the flexibility of assigning replication operation (sending) of a cluster to more than one processor arises, so we can use this flexibility to balance send message volume loads of processors or total communication volume loads of processors. Remapping problem and corresponding HP-based method changes to some extent to support replication and we give the following remapping method and we show proposed method minimizes the total communication volume.

Overall rendering operation can be thought as a series of rendering instances. Therefore rendering application can be defined as $\mathcal{I} = \{I^1, I^2, \ldots\}$. Furthermore working set notation $WS(P_k)$ for processor P_k is extended to $WS^h(P_k)$ to specify working set of P_k for the hth rendering instance.

For the parallelization of a particular rendering instance I^h, the remapping model should utilize the replication pattern of the previous rendering instances for reducing communication overhead due to the replication pattern of the current rendering instance. For example, all (some of) clusters in $WS^{h-1}(P_k)$ already resides in the replica buffer of processor P_k under the assumption that processors have sufficient (insufficient) memory. Furthermore, the replica buffer of P_k may contain replicas of other data primitives due to the available storage capacity. These additional replicas depend on the replication patterns of the computational phases before I^{h-1} and the replica replacement policy used. We define the extended replica set $eRS^{h-1}(P_k)$ of processor P_k to denote the set of clusters that reside in the replica buffer of processor P_k just before the parallel execution of phase I^h. The extended replica sets of individual processors determine the extended replication pattern:

$$eRS^{h-1}(\mathcal{P}) = \{eRS^{h-1}(P_1), eRS^{h-1}(P_2), \ldots, eRS^{h-1}(P_k)\}. \tag{4.1}$$

In this setting, the replication of a cluster C_j in processor P_k for parallel computation of I^h incurs a communication only if $C_j \notin eRS^{h-1}(P_k)$. In a similar manner extended home processor set for a cluster C_j can be defined as

$$eHome^h(C_j) = \{P_k : C_j \in eRS^h(P_k)\}. \tag{4.2}$$

So the repartitioning model should take the 2-tuple $(I^h, eRS^{h-1}(\mathcal{P}))$ as input for reducing communication overhead of phase h due to the replication.

Since the individual processors of the parallel system have a limited storage capacity, we can reserve a limited amount of storage for holding replicas for the next instance. Depending on the nature of the application, we might have the requirement that $eRS^{h-1}(P_k)$ has an upper bound M. That is, $Size(eRS^h(P_k)) \leq M$ where M denotes the upper bound on replica buffers of processors. The total size of the data primitives replicated at any processor P_k should not exceed this capacity.

In this section, we propose a two phase repartitioning approach for efficient parallelization of any viewing instance I^h given the previous extended replication pattern $eRS^{h-1}(\mathcal{P})$. In the first phase we determine the assignment of pixel blocks to processors while we determine the assignment of communication tasks to processors in the second phase.

4.1 Remapping with Replication

In this section, we show that pixel-block-to-processor assignment problem encountered in the first phase can be described as a K-way hypergraph partitioning with fixed vertices. For this purpose we construct a remapping/repartitioning hypergraph $\mathcal{H}_R(I^h, eRS^{h-1}(P)) = (\mathcal{V}, \widetilde{\mathcal{N}})$ by augmenting the interaction hypergraph $\mathcal{H}_I(\mathcal{I}) = (\mathcal{V}, \mathcal{N})$, proposed earlier in Section 3.3.2, with some pin additions.

New pins are added to the pin set of each net of $\mathcal{H}_I$ to encode the extended replication pattern $eRS^{h-1}(\mathcal{P})$ in $\mathcal{H}_R$. That is, for each net n_j of $\mathcal{H}_I$ we have:

$$Pins(\widetilde{n}_j) = Pins(n_j) \cup \{p_k : d_j \in eRS^{h-1}(P_k)\} \tag{4.3}$$

As seen in (4.3), a processor vertex p_k is added to the pin list of net n_j if cluster C_j is currently in the extended replica set of processor P_k. Note that the number of new pins added to net n_j is equal to replication count $rc(d_j)$ of data primitive d_j where $rc(C_j) = |\,eHome(C_j)\,|$ denotes the replication count of C_j.

Figure 4.1 illustrates the repartitioning hypergraph constructed for the sample application in Figure 3.1 for a given extended replication pattern on a

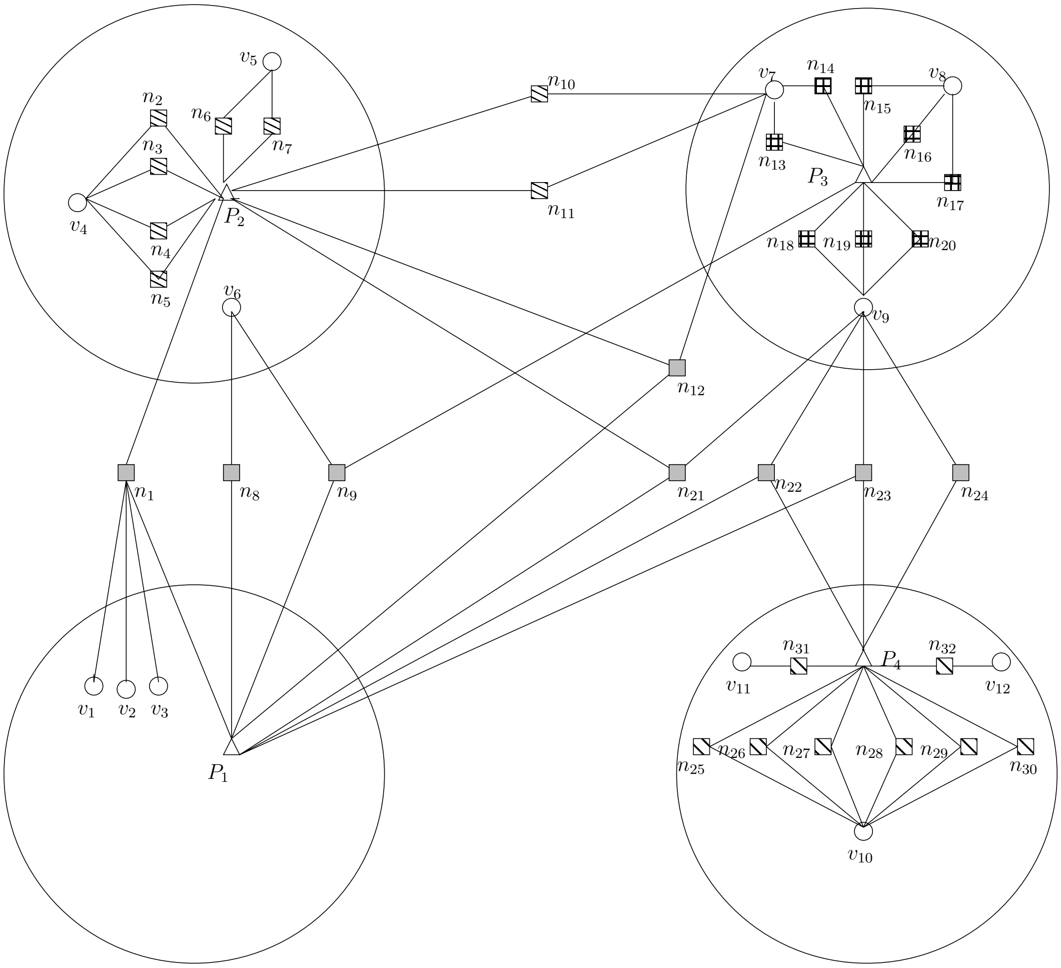


Figure 4.1: Repartitioning hypergraph of the sample application given in Figure 3.1 for the extended replication pattern $eRS^{h-1}(\mathcal{P}) = \{\{C_1, C_{8-9}, C_{12}, C_{21-23}\}, \{C_{2-7}, C_{10-12}, C_{21}\}, \{C_9, C_{13-20}\}, \{C_{22-32}\}\}$ on a 4-processor system.

four-processor system. In the figure, circles and triangles respectively represent task and processor vertices, whereas squares represent nets. As seen in the figure, net n_{12} connects to processor vertices p_1 and p_2 since cluster C_{12} is replicated on processors P_1 and P_2. Figure 4.1 also shows a four-way vertex partition which denotes the pixel-block-to-processor mapping $\{\{pb_1, pb_2, pb_3\}, \{pb_4, pb_5, pb_6\}, \{pb_7, pb_8, pb_9\}, \{pb_{10}, pb_{11}, pb_{12}\}\}$.

Consider a K-way vertex partition $\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ of $\mathcal{H}_R$. Note that each part $\mathcal{V}_k$ contains a single fixed vertex, which is processor vertex p_k, due to the partitioning with K-fixed vertices model. So partition Π is decoded as assigning the set of pixel blocks in part $\mathcal{V}_k$ to processor P_k. That is, vertex partition Π will denote the pixel-block-to-processor mapping $\{\mathcal{PB}_1, \mathcal{PB}_2, \ldots, \mathcal{PB}_K\}$, where $\mathcal{PB}_k = \{pb_i : v_i \in \mathcal{V}_k\}$.

Consider a K-way vertex partition Π of $\mathcal{H}_{\mathcal{R}}(I^h, eRS^{h-1}(P)))$. Since vertices in part $\mathcal{V}_k$ (except processor vertex p_k) correspond to the pixel blocks assigned to processor P_k and $w(p_k) = 0$, the weight W_k of $\mathcal{V}_k$ corresponds to the total expected computation time for processor P_k. So the partitioning constraint corresponds to maintaining computational load balance among the processors during the rendering instance I^h.

Extended replication pattern $eRS^{h-1}(\mathcal{P})$ determines the anchored connectivity set $\Psi(n_j)$ for each net n_j, for any partition Π.

$$\Psi(n_j) = \{\mathcal{V}_k : C_j \in eRS^{h-1}(P_k)\} \tag{4.4}$$

In hypergraph theoretic notation, $\Psi(n_j)$ denotes the set of parts connected by net n_j through processor vertices, i.e., $\mathcal{V}_k \in \Psi(n_j)$ if $p_k \in Pins(n_j)$. Thus $\Psi(n_j)$ represents the set of parts corresponding to the processors that hold a replica of cluster C_j. However, any part $\mathcal{V}_k$ in $\Lambda(n_j) - \Psi(n_j)$ corresponds to processor P_k that needs C_j but does not hold a replica of C_j. So the set $\Lambda(n_j) - \Psi(n_j)$ of parts corresponds to the set of processors that should receive a replica of C_j for the current rendering instance. That is, cluster C_j should be communicated $\mid \Lambda(n_j) - \Psi(n_j) \mid = \lambda(n_j) - \psi(n_j)$ times. Hence, total communication volume incurred by Π will be

$$
\begin{aligned}
TotalCommVol(\Pi) &= \sum_{n_j \in \mathcal{N}} (\lambda(n_j) - \psi(n_j))c(n_j) \qquad (4.5)\\
&= \sum_{n_j \in \mathcal{N}} \lambda(n_j)c(n_j) - \sum_{n_j \in \mathcal{N}} \psi(n_j)c(n_j) \qquad (4.6)\\
&= \sum_{n_j \in \mathcal{N}} (\lambda(n_j) - 1)c(n_j) + \sum_{n_j \in \mathcal{N}} c(n_j) - \sum_{n_j \in \mathcal{N}} \psi(n_j)c(n_j) \qquad (4.7)\\
&= Cutsize(\widetilde{\Pi}) + \sum_{n_j \in \mathcal{N}} c(n_j) - \sum_{n_j \in \mathcal{N}} \psi(n_j)c(n_j) \qquad (4.8)
\end{aligned}
$$

In (4.8), both the second and third summation terms are constants since they, respectively, represent the total dataset size $Size(\mathcal{D})$ and total data replication size $Size(eRS^{h-1}(\mathcal{P})) = \sum_{P_k \in \mathcal{P}} eRS^{h-1}(P_k)$. Hence, the partitioning objective of minimizing the cutsize according to the connectivity-1 metric given in (3.4) corresponds to minimizing the total volume of communication.

Consider the 4-way vertex partition shown in Figure 4.1. For vertex part $\widetilde{\mathcal{V}}_3$, $Nets(\widetilde{V}_3) - Nets(p_k) = \{n_{9-24}\} - \{n_9, n_{13-20}\} = \{n_{10-12}, n_{21-24}\}$. So 7 clusters $\{C_{10}, C_{11}, C_{12}, C_{21}, C_{22}, C_{23}, C_{24}\}$. Simmilarly, P_2 is to receive 2 clusters C_8 and C_9, whereas P_1 and P_4 do not receive any clusters. So the total communication volume is equal to $7 + 2 = 9$.

As seen in the figure, the cutsize is equal to 15 assuming unit-size data primitives. The total data set size and total data replication size are 32 and 38 respectively. So Equation 4.8 correctly computes the total communication volume since $15 + 32 - 38 = 9$.

For vertex part $\widetilde{\mathcal{V}}_3$, since $Nets(\widetilde{V}_k - p_k) = \{n_{10-24}\}$ the peak replica buffer size of processor P_3 because of the working set $WS(P_3) = \{C_{10-24}\}$.

4.2 Communication-Task-to-Processor Assignment

Recall that, for a given partition Π of $\mathcal{H}_R$, $\Psi(n_j)$ denotes the set of parts corresponding to the processors that hold a replica of cluster C_j, whereas $\Lambda(n_j)-\Psi(n_j)$ represents the set of parts corresponding to the processors that should receive a replica of cluster C_j. For the sake of clarity of the presentation of this subsection, we introduce the following parallel application specific notation

$$\begin{aligned} \mathit{RecvSet}(C_j) &= \{P_k : \mathcal{V}_k \in \Lambda(n_j) - \Psi(n_j)\} && (4.9) \\ \mathit{SendSet}(C_j) &= \{P_k : \mathcal{V}_k \in \Psi(n_j)\}. && (4.10) \end{aligned}$$

Here $\mathit{RecvSet}(C_j)$ denotes the set of processors on which the cluster C_j should be replicated via communication. So the number of communication tasks to be completed regarding cluster C_j is equal to $\mid \mathit{RecvSet}(C_j) \mid \; = \; \mid \Lambda(n_j) - \Psi(n_j) \mid$. Any processor in $\mathit{SendSet}(C_j)$ can achieve the communication task(s) of sending a copy of cluster C_j to any processor(s) in $\mathit{RecvSet}(C_j)$. We will exploit this flexibility in the assignment of communication tasks in order to minimize the maximum message volume (send message volume) handled by a processor.

The receive sets of clusters determine the pattern of processors' receive requirements, whereas the send sets of clusters determine the flexibility pattern on the send communication tasks. That is,

$$\begin{aligned} \mathit{Recv}(\mathcal{D}) &= \{\mathit{RecvSet}(C_1), \mathit{RecvSet}(C_2), \ldots, \mathit{RecvSet}(C_m)\} && (4.11) \\ \mathit{SendFlex}(\mathcal{D}) &= \{\mathit{SendSet}(C_1), \mathit{SendSet}(C_2), \ldots, \mathit{SendSet}(C_m)\}. && (4.12) \end{aligned}$$

Given $\mathit{Recv}(\mathcal{D})$ and $\mathit{SendFlex}(\mathcal{D})$ determined in the first phase (remapping), we propose a network flow based formulation for finding optimal communication-task-to-processor assignment for the second phase assuming equal sized clusters. Our formulation adapts and extends the model and methods recently proposed by Pinar and Hendrickson [4] for flexibly assignable unit tasks to improve load balance. In case of equal-sized clusters, the receive volume for processors can be represented in terms of the number of clusters to be received. That is, receive volume can be shown as

$$RecvCnt(P_k) =\mid \{d_j : P_k \in RecvSet(d_j)\} \mid \tag{4.13}$$

in terms of cluster count.

We construct a 3-layered flow network $\mathcal{F}(Recv(\mathcal{D}), SendFlex(\mathcal{D})) = (\mathcal{P}_R \cup \mathcal{U}_D \cup \mathcal{P}_S, \mathcal{E})$ which is referred to here as the assignment flow network. The set $\mathcal{P}_R = \{p_1^R, p_2^R, \ldots, p_K^R\}$ of vertices in the first layer represent the set of processors that will involve in receive-communication operations, where p_k^R corresponds to processor P_k. The set $\mathcal{U}_D = \{u_1, u_2, \ldots\}$ of cluster vertices in the middle layer represents the subset of clusters that are involved in at least one communication operation. That is, $u_i \in \mathcal{U}_D$ only if $\mid RecvSet(d_i) \mid \neq 0$. The set $\mathcal{P}_S = \{p_1^S, p_2^S, \ldots, p_K^S\}$ of vertices in the third layer represents the set of processors that may involve in send-communication operations, where p_k^S corresponds to processor P_k.

Edges are introduced only between the vertices of successive layers of $\mathcal{F}$, that is

$$\mathcal{E} = \mathcal{E}_{sR} \cup \mathcal{E}_{RU} \cup \mathcal{E}_{US} \cup \mathcal{E}_{St}. \tag{4.14}$$

The set $\mathcal{E}_{sR}$ of source edges is constructed by connecting the source vertex s to all receive vertices in the first layer. The capacity of each source edge is set to be equal to the number of clusters to be received by the respective processor, i.e., $cap\,(s, p_k^R) = RecvCnt(P_k)\ \ \forall\,(s, p_k^R) \in \mathcal{E}_{sR}$.

The set of receive edges connecting the vertices of first and second layers are determined as follows: There exists an edge from the receive-processor vertex p_k^R to the cluster vertex u_j if cluster C_j should be replicated on processor P_k via communication. That is,

$$\mathcal{E}_{RU} = \{(p_k^R, u_j) : P_k \in RecvSet(C_j)\}. \tag{4.15}$$

The capacity of each receive edge in $\mathcal{E}_{RU}$ is set to be equal to 1. The set of send-task-assignment edges connecting the vertices of second and third layers are determined as follows: There exists an edge from the cluster vertex C_j to the send-processor vertex p_k^S if the task of sending cluster C_j can be assigned to processor P_k. That is,

$$\mathcal{E}_{US} = \{(u_j, p_k^S) : P_k \in SendSet(C_j)\}. \tag{4.16}$$

The capacity of each assignment edge in $\mathcal{E}_{US}$ is set to be equal to infinite. Finally, the set $\mathcal{E}_{St}$ of terminal edges is constructed by connecting all send-processor vertices in the third layer to the terminal vertex t. Here we will use the notation $InAdj(\cdot)$ and $OutAdj(\cdot)$ to denote the set of vertices adjacent to a vertex through incoming and outgoing edges, respectively. The capacity of each assignment edge in $\mathcal{E}_{US}$ is determined as a function of parameter B. B is the parameter that will be used in network probing and this capacity assignment will be detailed in the next subsection.

Figure 4.2 ilustrates the general structure of assignment flow network. The three receive edges (p_x^R, u_j), (p_y^R, u_j) and (p_z^R, u_j) show that cluster C_j is to be received by processors P_x, P_y and P_z. The two assignment edges (u_j, p_k^S) and (u_j, p_l^S) show that cluster C_j can be sent by processors P_k and P_l. In other words, three receive edges and two assignment edges present the flexibility of assigning the three communication tasks among two processors P_x and P_y.

Consider assignment flow network $\mathcal{F}(Recv(\mathcal{D}), SendFlex(\mathcal{D}))$. Communication-task-to-processor assignment problem can be formulated as a maximum flow problem on assignment flow network $\mathcal{F}(Recv(\mathcal{D}), SendFlex(\mathcal{D}))$. If a complete flow f exists on $\mathcal{F}$, then flow f induces a complete communication-task-to-processor assignment.

Consider a complete flow f with $\mid f \mid = \sum_{k=1}^{K} cap(s, p_k^R)$ on $\mathcal{F}$. Note that each receive vertex p_x^R has $cap(s, p_x^R)$ outgoing edges of unit capacity, i.e., $\mid OutAdj(p_x^R) \mid = cap(s, p_x^R)$. Therefore saturation of all source edge capacities by f implies the saturation of all receive edge capacities. This in turn means that

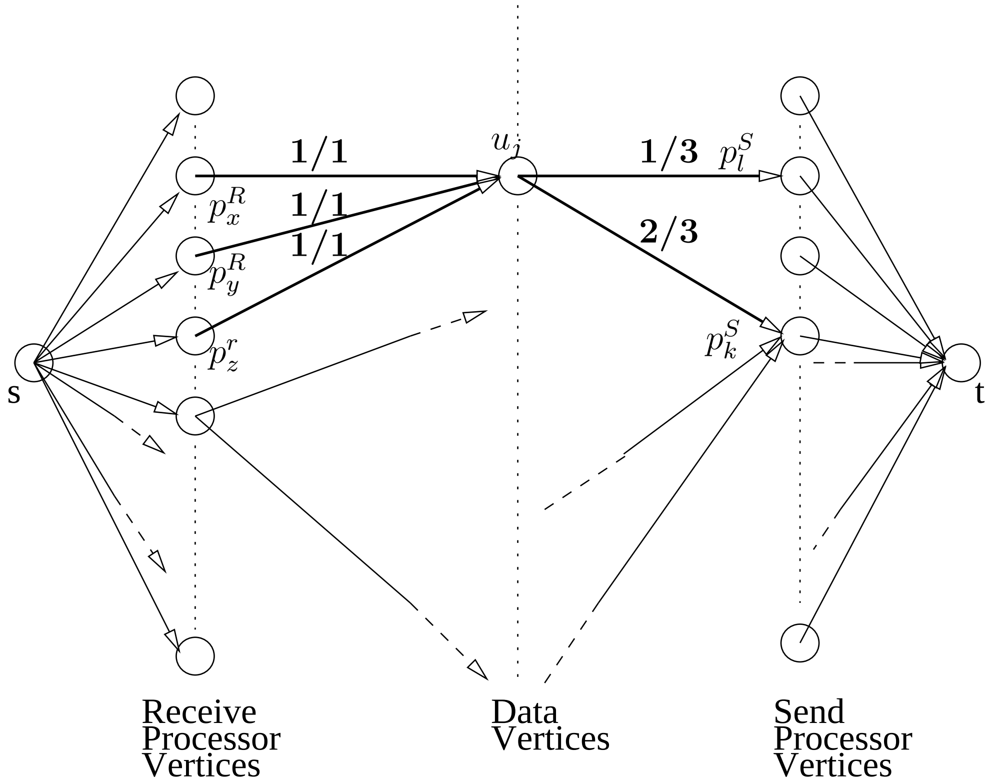


Figure 4.2: General Structure of the Assignment Flow Network

all receive requirements are satisfied by the send-task assignment to be induced by flow f. The flow on the assignment edges between the cluster vertices and send-processor vertices is decoded as send-task-to-processor assignment as follows: A flow $f(u_j, p_k^S)$ from cluster vertex u_j to send-processor vertex p_k^S means that processor P_k becomes responsible for sending cluster C_j to any $f(u_j, p_k^S)$ processors corresponding to the vertices in $InAdj(u_j)$. Since the capacities of all incoming receive edges of C_j are saturated and flow conservation holds on vertex u_j, $inDegree(u_j)$ send tasks associated with C_j will be shared among the processors corresponding to the vertices in $OutAdj(u_j)$. Therefore, send volume of a processor is

$$SendCnt(P_k) = f(p_k^S, t); \tag{4.17}$$

in terms of cluster count.

Consider a sample flow distribution regarding vertex u_j of flow network given in Figure 4.2. The flows $f(u_j, p_k^S) = 2$ and $f(u_j, p_l^S) = 1$ on the two outgoing edges of u_j assign two and one send-communication tasks to processors P_k and P_l, respectively, thus satisfying the three send communication operations associated with the data primitive d_j.

4.2.1 Assignment Flow Parametric Search

A parametric search algorithm can be used to find a solution to a problem that has minimal associated cost. Thus, in this part we will explain the details of a method that uses parametric search algorithm, similar to one proposed by Pinar and Hendrickson [4], that sets an upper bound on send volume loads of processors or total volume loads of processors. A parametric search algorithm has two components. First component is a probing function that determines if there is a solution that has associated cost less than a specified value. Second component is an algorithm that searches among a set of candidate optimal solution.

In this work we use maximum-flow algorithms as probe function. Therefore our claim is, a maximum-flow algorithm run on previously created assignment network $\mathcal{F}(Recv(\mathcal{D}), SendFlex(\mathcal{D}))$ can set an upper bound on maximum send volume of processors or maximum total communication volume of processors. For the sake of simplicity, we first present usage of maximum-flow algorithm (same as [4] but on different flow network) as a probe function to set an upper bound on maximum send volume of processors. Then we extend this method to set an upper bound on total communication volume handled by a processor.

Consider a complete communication-task-to-processor assignment Ξ. Ξ induces a maximum send volume load

$$MaxSendCnt(\Xi) = max(SendCnt(P_1), SendCnt(P_2), \ldots, SendCnt(P_K)) \tag{4.18}$$

in terms of cluster count. $MaxSendCnt(\Xi) \leq B$ if there exists a complete flow on $\mathcal{F}(Recv(\mathcal{D}), SendFlex(\mathcal{D}))$ where all terminal edges have capacity B. This is true because decoding of flow f on $\mathcal{F}(Recv(\mathcal{D}), SendFlex(\mathcal{D}))$ ensures that no processor is assigned more than $f(p_k^S, t)$ send operations. Capasity constraint $cap(p_k^S, t) = B$ for maximum flow algorithm ensures $f(p_k^S, t) \leq B$. Therefore $MaxSendCnt(\Xi) \leq B$.

For finding minimal $MaxSendCnt(\Xi)$ for a complete communication-task-to-processor assignment Ξ, we must find minimal value of B. Pinar and Hendrickson [4] propose two methods, bisection search and incremental search, which only incremental search is suitable (because of low number of probes) for our application.

Incremental search starts with the lowest possible solution and increases the probing value at each iteration until the optimal solution is found. However choosing increment size is important because very small increments increase execution time whereas very large increments cause the optimal value to be missed. Thus increments should be just large enough that after a failed probe increment should increase the probe value to smallest value that possibly has a solution.

Let W_T be the total send operations,i.e.,

$$W_T = \sum_{k=1}^{K} RecvCnt(P_k) \tag{4.19}$$

and B be is the probe value. Incremental search should start execution with lowest possible $B = \lceil W_T/K \rceil$ where K is the total processor number and every processor is assigned equal number of send operations. Then after a failed probe with probe value B, let W_r be the failed flow,i.e.,

$$W_r = W_T - \sum_{k=1}^{K} f(p_k^S, t) \tag{4.20}$$

$SatCnt$ be the number of saturated terminal edges,i.e.,

$$SatCnt = \mid \{(p_k^S, t) : f(p_k^S, t) = cap(p_k^S, t)\} \mid \tag{4.21}$$

and Inc be the increment on probe value B,i.e.,

$$Inc = \lceil W_r/SatCnt \rceil \tag{4.22}$$

then no probe value smaller than $B + Inc$ can have a complete communication-task-to-processor assignment. This comes from the fact that current topology of assignment network $\mathcal{F}(Recv(\mathcal{D}), SendFlex(\mathcal{D}))$ does not allow enough flow to come to vertex p_k^S for an unsaturated (p_k^S, t) edge (if it were to allow, additional unused terminal edge capacity would be used and more flow would be passed, which is a contradiction to maximum flow algorithm), therefore only additional flow can be passed over saturated edges. Pseudocode of classical IncrementalSearch algorithm is given below.

```
IncrementalSearch(){
        $B \leftarrow \lceil W_T/K \rceil$
        **foreach** unsuccessful probe **do**
                $Inc \leftarrow \lceil W_r/SatCnt \rceil$
                $B \leftarrow B + Inc$
        **end**
}
```

Figure 4.3 shows the assignment flow network and the flow obtained via balancing on send volumes. There are four processors and nine clusters subject to communication as shown in Figure 4.1. For the sake of simplicity in presentation no cluster is received by more than one processor, so bold arrows between data vertices and send processor vertices means the assignment of single send operation to the processor. There are cluster to graph point pairings such as: u_1-C_{10}, u_2-C_{11}, u_9-C_{12}, u_3-C_{21}, u_4-C_8, u_5-C_9, u_6-C_{22}, u_7-C_{23} and u_8-C_{24}. Note that every edge between receive vertices and data vertices are shown in bold, meaning every receive request is satisfied with the current flow. There are 9 send operations therefore best possible $B = \lceil 9/4 \rceil = 3$ and maximum flow found has been found for $B = 3$ value. As shown in the figure, maximally (communication task) loaded processors are assigned 3 send operations. As opposed to single home processor approach, this method decreases the send volume handled by a processor to 3 from 7.

We have shown how to use maximum flow algorithm as a probe function to

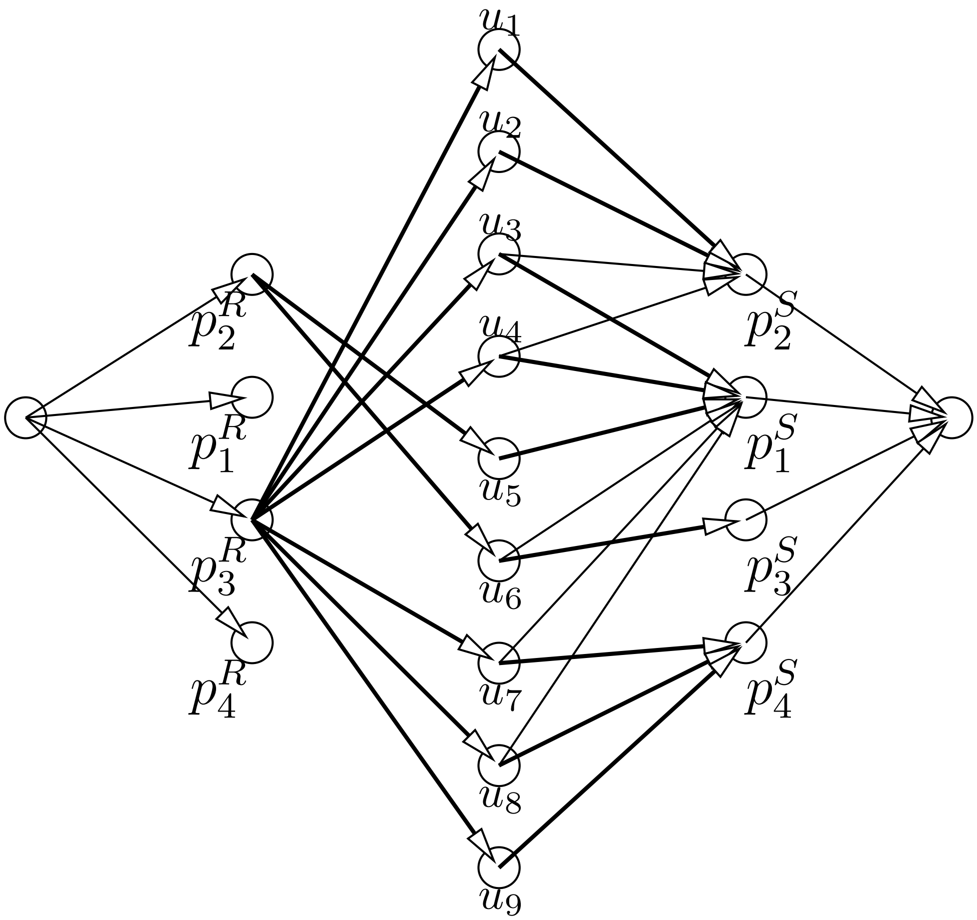


Figure 4.3: Assignment Flow Network Balanced by Send Volumes

set an upper bound on maximum send volume of processors and explained incremental search algorithm. Now we extend this method to set an upper bound on total communication volume handled by a processor. Consider a complete communication-task-to-processor assignment Ξ. Ξ induces a maximum total communication volume load $MaxTotalCnt(\Xi)$,i.e.,

$$TotalCnt(P_k) = SendCnt(P_k) + RecvCnt(P_k) \tag{4.23}$$

$$MaxTotalCnt(\Xi) = max(TotalCnt(P_1), \ldots, TotalCnt(P_K)) \tag{4.24}$$

in terms of cluster count.

$MaxTotalCnt(\Xi) \leq B$ if there exists a complete flow on $\mathcal{F}(Recv(\mathcal{D}), SendFlex(\mathcal{D}))$

where,

$$cap(p_k^S, t) = B - RecvCnt(P_k). \tag{4.25}$$

This is true because decoding of flow f on $\mathcal{F}(Recv(\mathcal{D}), SendFlex(\mathcal{D}))$ ensures that no processor is assigned more than $f(p_k^S, t)$ send operations. Capacity constraint $cap(p_k^S, t) = B - RecvCnt(P_k)$ for maximum flow algorithm ensures $f(p_k^S, t) \leq max(0, B - RecvCnt(P_k))$. Therefore following assertions are true

$$\begin{aligned}
SendCnt(P_k) &= f(p_k^S, t) \leq B - RecvCnt(P_k) && (4.26)\\
TotalCnt(P_k) &= SendCnt(P_k) + RecvCnt(P_k) && (4.27)\\
&\leq RecvCnt(P_k) + B - RecvCnt(P_k) && (4.28)\\
TotalCnt(P_k) &\leq B && (4.29)
\end{aligned}$$

for receive processor vertices' terminal edges with $B \geq RecvCnt(P_k)$. Then $TotalCnt(P_k) \leq B$ for processors with $RecvCnt(P_k) \leq B$.

For finding minimal $MaxTotalCnt(\Xi)$ for a complete communication-task-to-processor assignment Ξ, we must find minimal value of B. As in the case of minimization of $MaxSendCnt(\Xi)$ incremental search is used for our application. Incremental search starts with the lowest possible solution and increases the probing value at each iteration until the optimal solution is found.

Incremental search should start execution with lowest possible B. If a selection of B where B is small enough that total terminal edge capacity

$$\begin{aligned}
TotalTermCap(\mathcal{F}) &= \sum_{k=1}^{K} (B - RecvCnt(P_k)) && (4.30)\\
&= \sum_{k=1}^{K} B - \sum_{k=1}^{K} RecvCnt(P_k) && (4.31)\\
&= KB - W_T \qquad \leq \quad W_T. && (4.32)
\end{aligned}$$

is not feasible. However to make $TotalTermCap(\mathcal{F}) \geq W_T$, minimum B could be calculated using 4.32 as

$$B = \lceil 2W_T/K \rceil. \tag{4.33}$$

Furthermore we define

$$MaxRecvCnt(\Pi) = max(RecvCnt(P_1), RecvCnt(P_2), \ldots, RecvCnt(P_K)) \tag{4.34}$$

where $MaxTotalCnt(\Xi) \geq MaxRecvCnt(\Pi)$ using 4.23 and 4.24. Since B is an upperbound for $MaxTotalCnt(\Xi)$, following assertion is true.

$$B \geq MaxRecvCnt(\Pi) \tag{4.35}$$

Using 4.33 and 4.35 lowest possible starting B is

$$B = max(\lceil 2W_T/K \rceil, MaxRecvCnt(\Pi)). \tag{4.36}$$

After a failed probe calculation of Inc is the same as balance on send volumes of processors case. Pseudocode of extended IncrementalSearch algorithm is given below.

```
IncrementalSearchExtended(){
        $B \leftarrow max(\lceil 2W_T/K \rceil, MaxRecvCnt(\Pi))$
        **foreach** unsuccesful probe **do**
                $Inc \leftarrow \lceil W_r/SatCnt \rceil$
                $B \leftarrow B + Inc$
         **end**
}
```

Figure 4.4 shows the assignment flow network and the flow obtained via balancing on total volumes handled by processors. There are four processors and nine clusters subject to communication as shown in Figure 4.1. For the sake of simplicity in presentation no cluster is received by more than one processor, so bold arrows between data vertices and send processor vertices means the assignment of single send operation to the processor. There are cluster to graph point pairings such as: u_1-C_{10}, u_2-C_{11}, u_9-C_{12}, u_3-C_{21}, u_4-C_8, u_5-C_9, u_6-C_{22}, u_7-C_{23} and u_8-C_{24}. Note that every edge between receive vertices and data vertices are shown in bold, meaning every receive request is satisfied with the current flow. There are 9 send operations therefore best possible $B = \lceil 2 * 9/4 \rceil = 5$. However processor 3 is already receiving 7 clusters so best possible $B = 7$. As shown in the figure, if a processor is already scheduled to receive clusters then smaller amount of send operations are assigned to that processor. For example, processor 3 is already receiving 7 clusters so no send operation is assigned to processor 3.

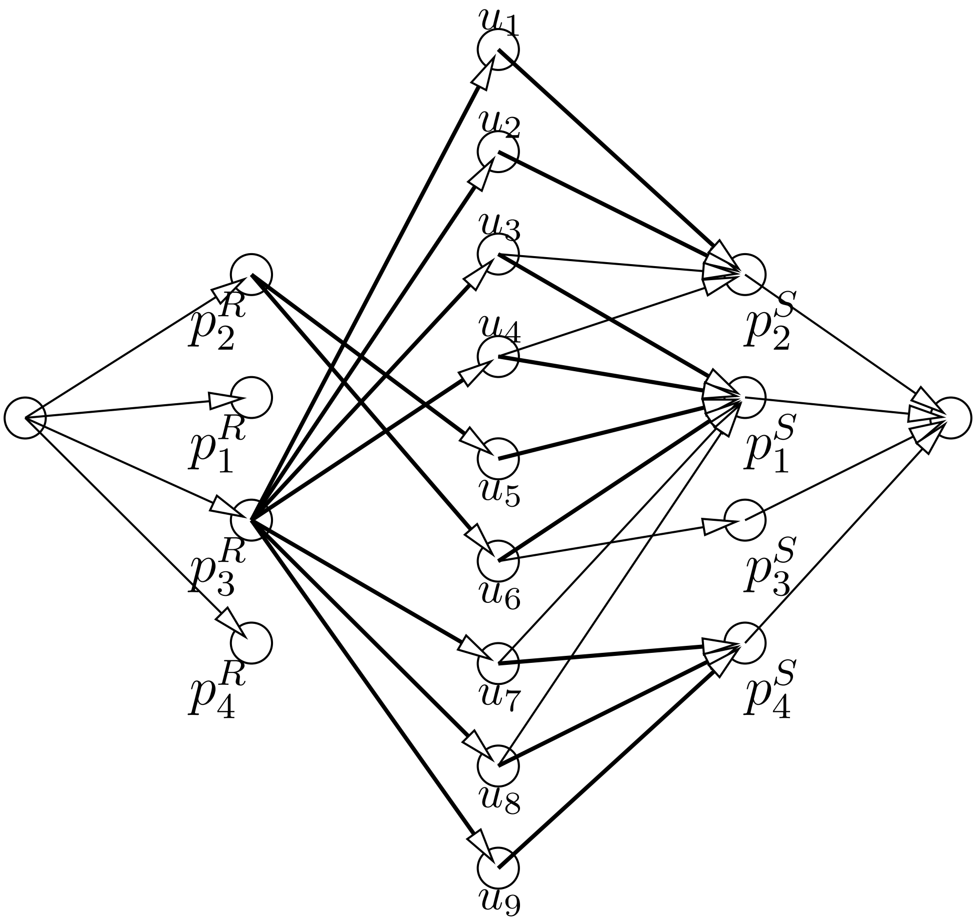


Figure 4.4: Assignment Flow Network Balanced by Total Volumes

As opposed to single home processor approach, this method decreases the send volume handled by a processor to 4 from 7.

4.3 Replication Contraction

We allow clusters to replicate in more than one processor via natural replication process of the renderer. However, monitoring and necessary deletion on replicas is needed as the execution continues. At each iteration, replication count of any cluster may increase and total replicated clusters on a processor may exceed the memory limit M of the processors. We have employed deletion algorithm at the end of each iteration, making memory space for the remapping algorithm.

Allowing replication has the main goal of utilizing excess memory in the system. Therefore algorithms utilizing replication should allow replication until some point where excess system memory is used but not exceeded. So after each iteration of rendering algorithm a portion of the replicas will be deleted. This section mainly focuses on clever contraction of replication buffer so that system has the most flexibility for fast rendering in the next iteration.

We proposed and implemented several algorithms for replica contraction, namely: LRRF (least recently rendered first), LCF (least connected first), SVF (smallest volume first), SAF (smallest area first), SDPF (smallest data points first) ,LDPF (largest data points first). From these algorithms LRRF mainly aims to utilize frame to frame coherence, SVF and SAF aims to increase the replication of clusters most likely to be replicated, SDPF and LDPF aims to exploit the trade off between memory usage and bandwidth usage. However none of them produces good results mainly because they restrict the flexibility used by assignment flow network.

We have proposed 2 more replica contraction algorithms, namely: RAND (random) and DELNET (replica contraction network). Test results were given by using these methods since they produce more flexible patterns than previously proposed ones. RAND algorithm randomly deletes clusters from candidates until some previously define value reached while DELNET tries to maximize the flexibility used by assignment flow network via finding a contraction pattern in the parallel system so that each cluster is replicated in at least B_{contr} processors. We will give details about DELNET algorithm in the next section.

4.3.1 DELNET

DELNET is a maximum flow network based parametric search algorithm (just like assignment flow network). The main idea behind DELNET algorithm is to increase the flexibility of assignment network via enforcing a lower bound on replication count of minimally replicated cluster while deleting clusters. Construction and parametric search of DELNET is explained below.

We construct a 2-layered flow network $\mathcal{F}_{contr} = (\mathcal{P}_{contr} \cup \mathcal{D}_{contr}, \mathcal{E}_{contr})$ which is referred to here as the replica contraction network. The set $\mathcal{P}_{contr} = \{p_1, p_2, \ldots, p_K\}$ of vertices in the first layer represent the set of processors that needs to delete at least one cluster, where p_k corresponds to processor P_k. The set $\mathcal{D}_{contr} = \{c_1, c_2, \ldots c_m\}$ of cluster vertices in the second layer represents the clusters that are possible candidates to be deleted. That is, C_i is a candidate of deletion only if $\mid rc(C_i) \mid \neq 1$.

Edges are introduced only between the vertices of successive layers of $\mathcal{F}_{contr}$, that is

$$\mathcal{E}_{contr} = \mathcal{E}^{sP}_{contr} \cup \mathcal{E}^{PD}_{contr} \cup \mathcal{E}^{Dt}_{contr}. \tag{4.37}$$

The set $\mathcal{E}^{sP}_{contr}$ of source edges is constructed by connecting the source vertex s to the processor vertices in the first layer if processors associated with the processor vertices have to delete at least 1 cluster. The capacity of each source edge is set to be equal to the number of clusters to be deleted by the respective processor, i.e., $cap\,(s, p_k) = \max(eRS(P_k) - M, 0)$.

The set of delete edges connecting the vertices of first and second layers are determined as follows: There exists an edge from the processor vertex p_k to the cluster vertex c_j if cluster C_j is a candidate to be deleted by processor P_k The capacity of each delete edge in $\mathcal{E}^{PD}_{contr}$ is set to be equal to 1.

Finally, the set $\mathcal{E}^{Dt}_{contr}$ of terminal edges is constructed by connecting all cluster vertices in the second layer to the terminal vertex t. The capacity of each terminal edge in $\mathcal{E}^{Dt}_{contr}$ is determined as a function of parameter B_{contr}. B_{contr} is the parameter that will be used in network probing and it is the replication count of minimally replicated cluster after deletion. The capacity of each terminal edge in $(c_i, t) = rc(C_i) - B_{contr}$. However as opposed to previous assignment flow network where parametric search value is slowly increased in replica contraction network parametric search value is slowly decreased. We employ such technique because higher lower bound on minimally replicated cluster presents us with more flexibility.

Parametric search of lower bound on minimally replicated cluster begins from the highest possible value, which is $B_{contr} = \min(rc(C_1), rc(C_2), \ldots, rc(C_m))$. Then as the parametric search goes we decrease the B_{contr} value until a complete flow is found (here we won't go into detail about how to decrease the B_{contr}, however it is similar in nature to the assignment flow parametric search). Then complete flow on replica contraction network should be decoded as, if unit flow exists on edge (p_k, c_j), then processor P_k should delete cluster C_j. This way, B_{contr} value resulting in complete flow on replica contraction network becomes the lower bound for minimally replicated cluster.

Chapter 5

Experimental Results

This chapter presents experimental results using three different datasets and we will show that findings of these experiments validate the proposed algorithms.

5.1 Datasets and Environment

Proposed algorithms are implemented on a 32-node cluster interconnected by a Fast Ethernet switch using LAM implementation of the MPI message passing interface. Each node has an Intel Pentium IV 3.0 GHz processor, 1 GB of RAM and runs Mandrake Linux, version 10.1.

We have used three dataset in this thesis for experiments. These datasets were obtained from NASA-Ames Research Center [38] and they represent the results of CFD simulations. All these datasets were originally curvilinear in structure and consisted of hexahedral cells. We have converted the initial format into unstructured tetrahedral format by dividing each hexahedral cells into five tetrahedral cells using tetrahedralization techniques presented in [39][40]. Alco connectivity information is kept during this operation, which then used to speed up the rendering process.

Table 5.1 presents some related information about datasets Blunt Fin

Data Set	Vertices	Cells	CSV
BLUNT	40960	187395	5.50
POST	109744	513375	4.26
COMB	47025	215040	0.42

Table 5.1: Data Set Properties

(BLUNT), Combustion Chamber (COMB) and Oxygen Post (POST). In this table, number of vertices, number of cells and an irregularity metric is given. CSV value gives us the cell size variation within the cells of a data set, so bigger CSV values means there is more irregularity in the dataset.

We have used a wide range of changing parameters for the experiments. One parameter that we haven't changed during our experiments is image size. We have used 400 × 400 as our final image size. We have used a fixed and somewhat smaller image size because fast development of graphics hardware and usage of GPUs for rendering purposes increase the actual rendering speed so make the communication step for parallel volume rendering applications the bottleneck. In this work we have used previously implemented CPU based rendering method [41] which does not utilize the fast GPU based rendering. However, simply reducing the image sizes, so reducing rendering time, enables us to show validity of our algorithm for current state of rendering algorithms and architectures. As mentioned above reader should refer to [41] for details of sequential rendering algorithm.

Parallelization of sequential rendering algorithm have been previously carried out in [2][45]. This work is an extension of [2][5]. In this work we have tried to optimize the communication pattern with the help of replication. We have investigated the effect of replication on performance and tried to overcome the irregularity of the datasets and reader should refer to [5] for experimental results that does not directly relate to replication, such as view independent preprocessing, workload calculation etc.

All timings are in seconds and communication volumes are given in MBytes. Imbalance metrics are given as the ratio of maximum value to average value.

5.2 Inverse Net Size Heuristic

Irregular structure of the datasets may cause imbalance in rendering loads or imbalance in communication loads. Balancing rendering loads is guaranteed by hypergraph partitioning tool PaToH [46] but imbalance on communication loads are not explicitly monitored by the partitioning (remapping) tool so a processor may have to receive much more data than the average or a processor may have to send much more data than the average. Irregular structure of the data we have used nearly guarantees that unacceptable communication imbalance ratios will be obtained. In our rendering application case, for datasets BLUNT and POST, we have high receive communication load imbalances. To speed up the migration process we have to address this problem.

A multi-constraint partitioning method is needed where one constraint monitors the balance in rendering loads where other monitors the receive volume loads of processors (it is possible for receive volume loads but not possible for send volume loads when replication is present). For remapping/repartitioning hypergraph $\mathcal{H}_R(I^h, eRS^{h-1}(P)) = (\mathcal{V}, \widetilde{\mathcal{N}})$ and the partition Π, balance on $\mid \ Nets(\mathcal{PB}_k) - Nets(p_k) \ \mid$ infer balance on receive volumes of processors. In other words, nets that connect the part k but not processor vertex p_k, corresponds to clusters that processor k needs but does not have. So balance on $\mid Nets(\mathcal{PB}_k) - Nets(p_k) \mid$ means balance on receive volume loads' of processor where clusters are equal sized. However, most of the current HP tools do not support the feature of defining partitioning constraint(s) on the nets of parts. To our knowledge only PaToH support defining constraint on maintaining balance all nets of every part but this feature of PAToH can not be used directly since the nets of a part should be differentiated for satisfying the constraints regarding receive volume loads.

For achieving an effect of balance on receive volume loads of processors we propose a heuristic which we call Inverse Net Size Heuristic (INSH) where communication loads induced by transfer of a cluster is equally divided and added to vertex weights of pixel blocks that use the cluster at hand. So vertex weight of a pixel vertex in the hypergraph should be redefined as

$$w(v_i) = time(pb_i) + e \times \sum_{n_j \in Nets(v_i)} \frac{c(n_j)}{\mid Pins(n_j) - rc(C_j) \mid}. \qquad (5.1)$$

Here $time(pb_i)$ is the original weight of the vertex, e is the scaling factor between rendering loads and communication loads and $\mid Pins(n_j) - rc(C_j) \mid$ is the number of pixel vertices associated with net n_j. Scaling factore e is needed because, we are combining 2 components which are not the same type. We have used scaling factors ranging from 0 to 3120 for the experiments. For the readability of this thesis, these values are normalized so e values ranges from 0 to 80 with intervals of 10 in the presented results.

A single constraint partition with INSH do not find the perfect partition, which both balances the receive volume loads of processors and rendering loads of processors. Instead it forces partitioning tool to assign less rendering load to a partition that is already assigned to receive a high volume, which avoids additional receive volume load associated with not assigned rendering load. Therefore, INSH may lead to imbalances in rendering loads of processors while balancing the communication volumes of processors. Sample statistical data is given in Table 5.2. Data presented in this table is produced using 32 processors and the POST dataset with different replication factors (0-50-100-150-200%) and the average values are presented. These particular parameters are chosen because they produce the most representative results to see the effect of INSH scaling factor. However, a complete set of experimental results can be found in Section 5.6.

Total time, migration time and rendering time is defined as the maximum time spent by any processor during respective phases. Rendering imbalance between processors is defined as a ratio of maximum rendering load of a processor to average rendering load. Migration imbalance is defined as the ratio of maximum effective communication volume to average of effective communication volume where effective communication volume is defined as the maximum of send and receive volumes of processors. Selection of such migration imbalance metric is needed because in two port communication model maximum of send or receive volume dominates the communication time. As presented in Table 5.2, maximally receiving processor is receiving about 20 MB of data whereas maximally sending

	e	tt	mt	rt	mi	ri	msv	mrv
32-Processors - POST	0	2.56	2.09	0.20	2.39	1.20	7.37	20.58
	10	2.02	1.46	0.29	2.35	1.83	9.64	8.96
	20	2.12	1.46	0.38	2.39	2.51	10.17	8.37
	30	2.18	1.45	0.45	2.38	3.05	10.18	8.33
	40	2.26	1.45	0.53	2.37	3.56	10.23	8.32
	50	2.32	1.44	0.60	2.39	4.03	10.27	8.27
	60	2.40	1.44	0.67	2.39	4.50	10.30	8.23
	70	2.43	1.43	0.72	2.39	4.83	10.23	8.32
	80	2.48	1.43	0.77	2.40	5.13	10.30	8.24

Table 5.2: 32-Processor statistics with changing INSH scaling factor for the dataset POST. Reported parameters are INSH scaling factor (e), total time (tt), migration time (mt), rendering time (rt), migration imbalance (mi), rendering imbalance (ri), maximum send volume handled by a processor (msv) and maximum receive volume handled by a processor (mrv).

processors is sending 7 MB of data without INSH. For the architectures, such as ours, using two port communication model, dominating factor for a processors communication time is the maximum of send volume of a processor and receive volume of a processor. So decreasing the receive volume of maximally receiving processor is important even if this causes imbalance in rendering times. Table 5.2 shows even usage of small scaling factors causes a radical decrease in maximum receive volume in the system. Furthermore, we expect, and observe, that usage of increasing scaling factors of INSH also causes increase in rendering time and rendering load imbalance between processors. Also there is a proportion in increasing rates of rendering time and rendering imbalance. It is because rendering imbalance between processors is defined as a ratio of maximum rendering load of a processor to average rendering load. Since average rendering load is constant no matter what the scaling factor is, we see the proportion between rendering time and the rendering imbalance but same proportion is not present between migration times and migration imbalance metrics for changing scaling factors. Change in communication pattern enforced by change in scaling factor also changes the average of effective communication volume. For the selection of optimal INSH scaling factor, total times should be considered because the trade-off between communication times and rendering times best can be observed with this metric.

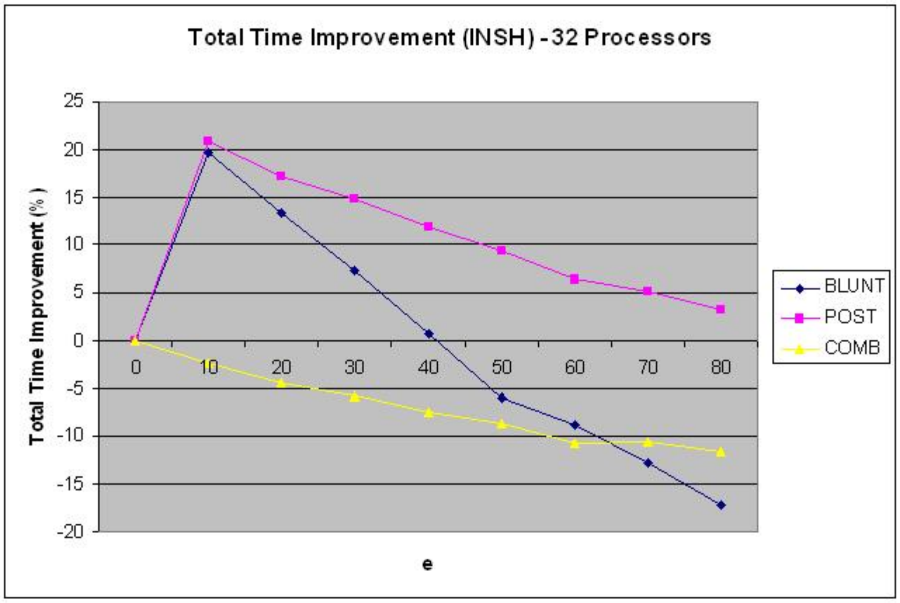


Figure 5.1: Total time with changing INSH scaling factor - 32 Processor

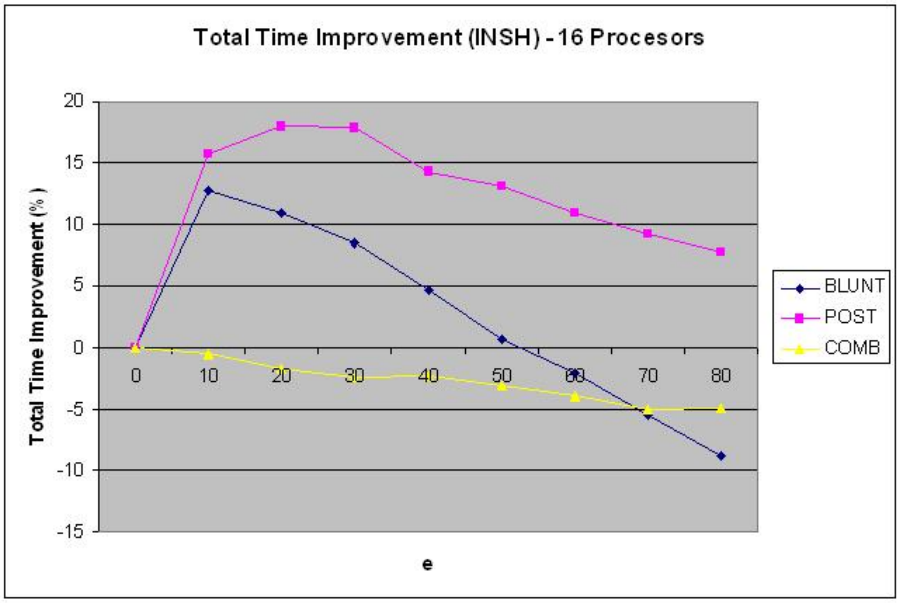


Figure 5.2: Total time with changing INSH scaling factor - 16 Processor

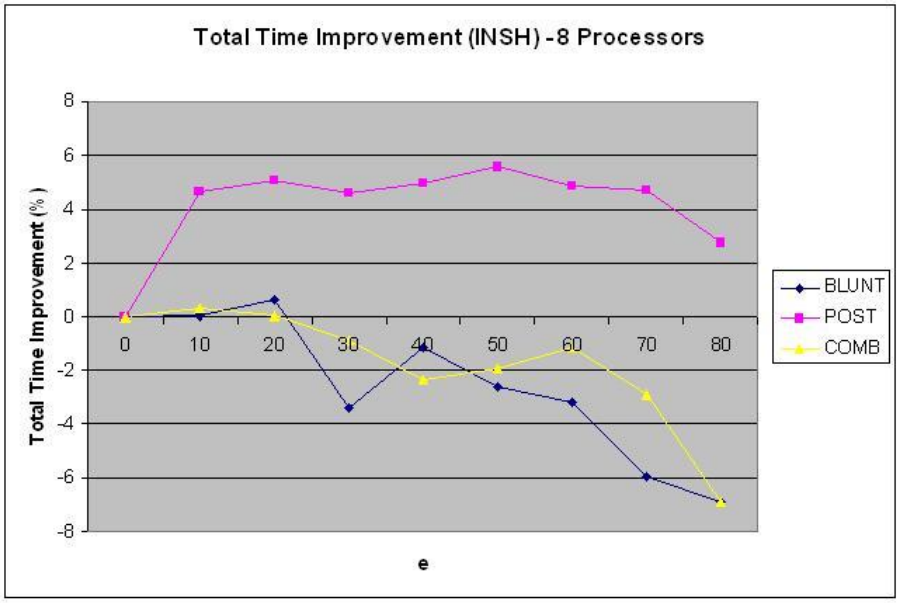


Figure 5.3: Total time with changing INSH scaling factor - 8 Processor

Figures 5.1, 5.2, 5.3 show the relation between total rendering times and INSH scaling factor for processor counts 32, 16 and 8. Figures show that INSH scaling factor usage should be directly proportional with irregularity of the dataset (CSV) because fairly regular dataset COMB always shows decrease in performance with higher scaling factors, whereas fairly irregular datasets BLUNT and POST shows increase in performance up to some point and then shows decrease in performance. Optimal INSH scaling factor for irregular datasets seems to be 10. From this point we will show experimental results using INSH scaling factor 10 for datasets BLUNT and POST while using results INSH scaling factor 0 for the dataset COMB.

5.3 Effect of Replication

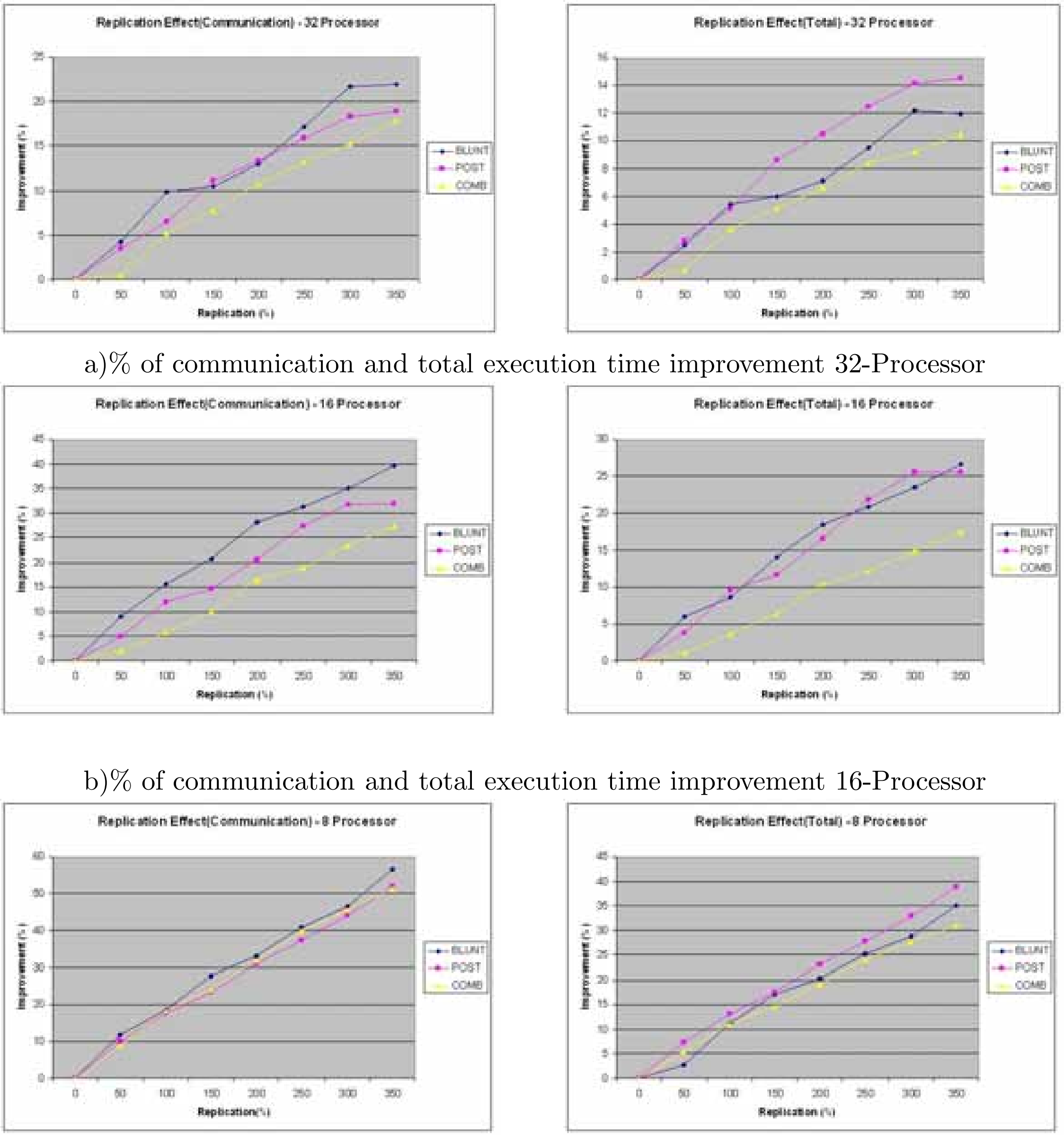
Replication of clusters will improve the communication time even though no other optimization technique is employed. Consider a processor is assigned a set of pixel blocks so it needs a set of clusters for rendering purposes. If required clusters are already replicated by the processor, no communication is needed as opposed to the

	R(%)	tt	mt	mtv	ttimp	mtimp	mtvimp
32-Processors POST	0	1.99	1.57	16.60	-	-	-
	50	1.94	1.51	16.00	2.79	3.49	3.57
	100	1.89	1.47	15.42	5.13	6.50	7.09
	150	1.82	1.39	14.62	8.63	11.17	11.90
	200	1.78	1.36	14.09	10.47	13.29	15.11
	250	1.74	1.32	13.30	12.43	15.86	19.89
	300	1.71	1.28	12.73	14.12	18.27	23.32
	350	1.70	1.27	12.73	14.52	18.82	23.31

Table 5.3: 32-Processor statistics showing improvement with different replication percentages for the POST dataset. Reported parameters are percentage of replication (R), total execution time (tt), migration time (mt), maximum volume handled by a processor (mtv), total execution time improvement (ttimp), total migration time improvement (mtimp), improvement for maximum message volume handled by a processor (mtvimp).

case that clusters are retrieved from other processors. So most basic function of data replication is avoidance of communication. This basic improvement should be stated with experiments and excluded from the improvements of optimization techniques that use replication. Table 5.3 shows the effect of replication for 32 processors and dataset POST. R denotes the percentage of the replication. For example, 100% replication means processor stores two times its original data size. tt (total execution time) and mt (migration time) were explained before and are measured in MB. mtv is maximum total volume handled by a processor and it is also measured in MB. ttimp, mtimp and mtvimp are the improvements in total execution time, total migration time and maximum total volume handled by a processor with increasing replication. All of the improvements were given in percentages.

As seen in Table 5.3 with increasing amount of replication, results shows fairly regular improvements in total execution time, total migration time and maximum volume handled by a processor. Further data can be found in Section 5.6. Graphical representations of these data is shown in Figure 5.4.



a)% of communication and total execution time improvement 32-Processor

b)% of communication and total execution time improvement 16-Processor

c)% of communication and total execution time improvement 8-Processor

Figure 5.4: Effect of replication on total execution time and communication time for 32, 16 and 8 processors

5.4 Network Flow Algorithm Improvement

As presented in Chapter 4, network flow formulation has 2 balancing options. First option balances send volumes of processors while second options balances the total communication loads of processors. Balancing send volumes, when receive volumes are already determined, effectively corresponds to minimizing maximum of send or receive volumes of processors, i.e., minimizing effective communication load for two port communication model. Balancing total communication loads of processors corresponds to balancing communication between processors for single port communication model. Since our architecture uses two port communication model, we have balanced send volumes of processors for the experiments. A detailed experiment set is given in Section 5.6. Partial results for 32-processors with POST dataset are shown in Table 5.4. R (% of replication), mt (migration time) and mtimp (% of migration time improvement) are explained before. net specifies if network flow based algorithm is used or not, msv is maximum send volume handled by a processor and msvimp is the percentage of improvement achieved for maximum send volume loads of processors with assignment network flow algorithm. As seen in Table 5.4 for 32 processor improvements of 15% to 27% is obtained for cluster migration phase.

Figures 5.5 and 5.6 present good improvement rates whereas Figure 5.7 shows somewhat unpredictable improvement rates. We think that deciding replication relative to processors initial data size is the problem. For example, 8-processors replication of 250% nearly replicates half of the dataset on a single processor, which decrease the necessity to improve communication and makes the application behave more like a serial rendering algorithm. Therefore, we think proposed network flow based algorithm is more suitable to be used with large processor numbers and big datasets. We have used maximum flow algorithm first proposed by Goldberg and Tarjan [49][50], which has a computational complexity of $O(V^2E)$. Even for high processor numbers time spent for maximum network flow algorithm is negligible, so this data is omitted in tables.

	R(%)	net	mt	msv	mtimp	msvimp
32-Processors POST	50	0	1.51	10.06	15.26	36.13
		1	1.28	9.32		
	100	0	1.47	9.57	23.57	50.10
		1	1.12	8.93		
	150	0	1.39	9.12	23.90	55.84
		1	1.06	8.48		
	200	0	1.36	8.93	25.05	55.59
		1	1.02	8.31		
	250	0	1.32	8.50	27.49	56.18
		1	0.96	8.08		

Table 5.4: 32-Processor statistics showing improvements of network flow algorithm (while balancing send volumes of processors) for POST dataset with different replication percentages. Reported parameters are percentage of replication (R), network flow present bit (net), migration time (mt), maximum send message volume handled by a processor (msv), migration time improvement (mtimp), improvement for maximum send message volume handled by a processor (msvimp).

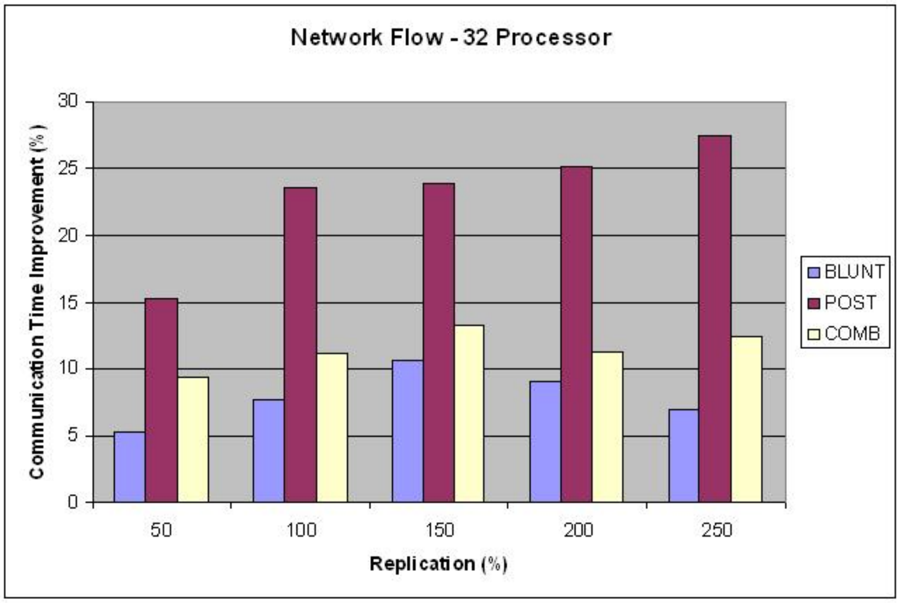


Figure 5.5: Communication time Improvement with Proposed Network Flow Algorithm - 32 Processor

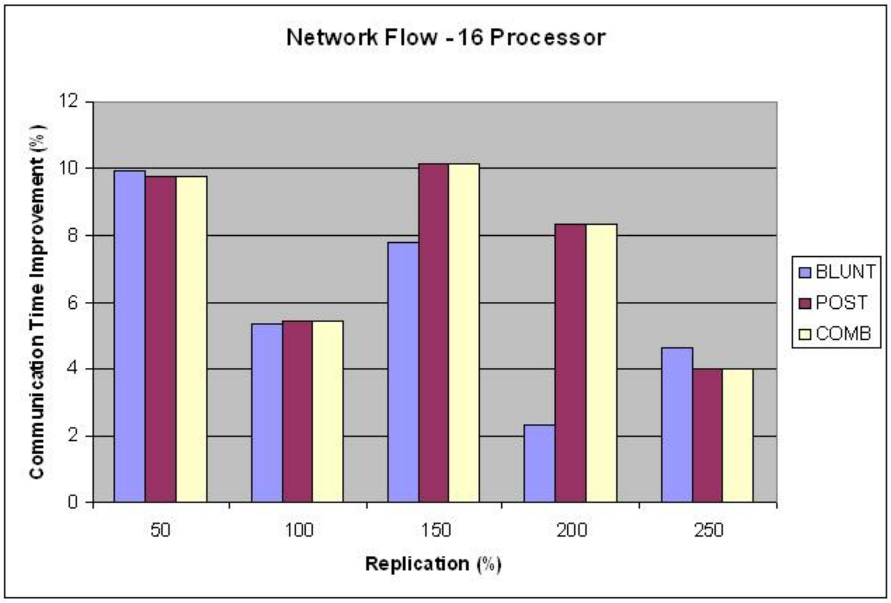


Figure 5.6: Communication time Improvement with Proposed Network Flow Algorithm - 16 Processor

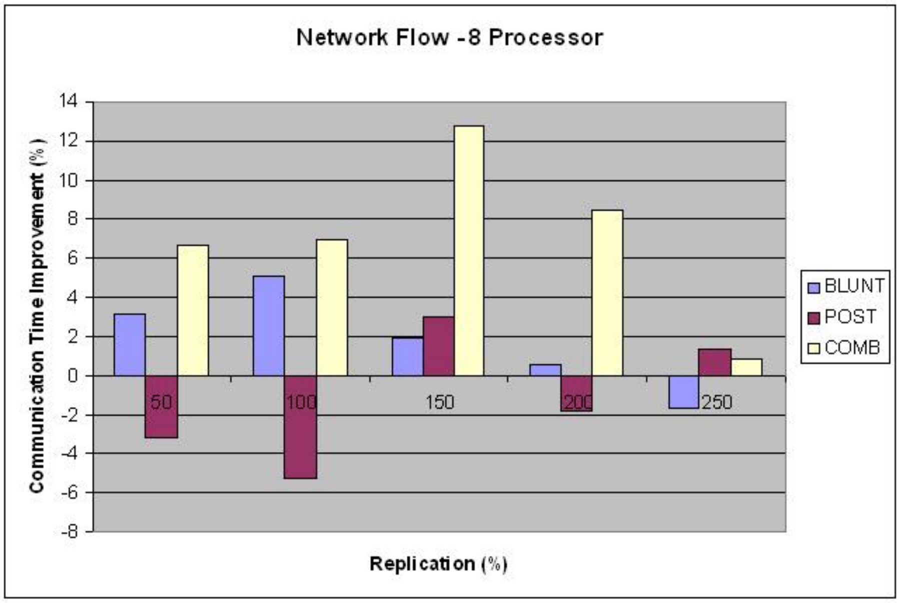


Figure 5.7: Communication time Improvement with Proposed Network Flow Algorithm - 8 Processor

5.5 Total System Improvement

Total system improvement consists of 2 components. First component is the improvements obtained via usage of INSH and the second component is the improvements obtained via usage of network flow algorithm. Table 5.5 shows the improvement results on total execution time and migration time. A complete set of results can be found in Section 5.6.

	R(%)	mod	tt	mt	ttimp	mtimp
32-Processors-POST	50	0	2.51	2.18	36.04	41.15
		1	1.61	1.28		
	100	0	2.41	2.07	40.15	46.03
		1	1.44	1.12		
	150	0	2.35	2.02	41.16	47.51
		1	1.38	1.06		
	200	0	2.31	1.98	42.94	48.56
		1	1.32	1.02		
	250	0	2.23	1.89	43.38	49.45
		1	1.26	0.96		

Table 5.5: 32-Processor statistics showing total system improvements consisting of network flow algorithm improvements and INSH improvements for POST dataset with different replication percentages. Reported parameters are percentage of replication (R), network flow/INSH present bit (mode), total execution time (tt), migration time (mt), total execution time improvement (ttimp), migration time improvement (mtimp).

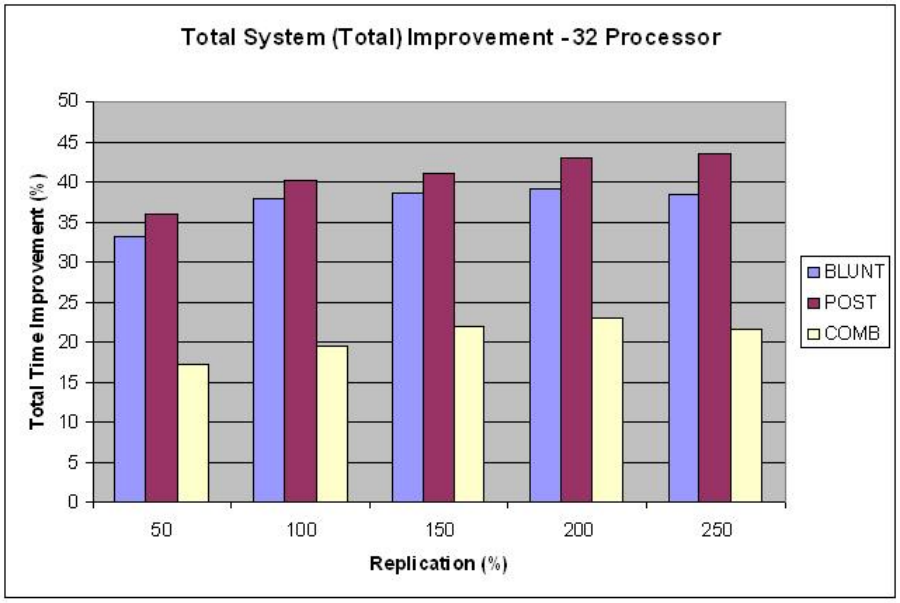


Figure 5.8: Migration Time Improvement with Proposed Network Flow Algorithm and INSH - 32 Processor

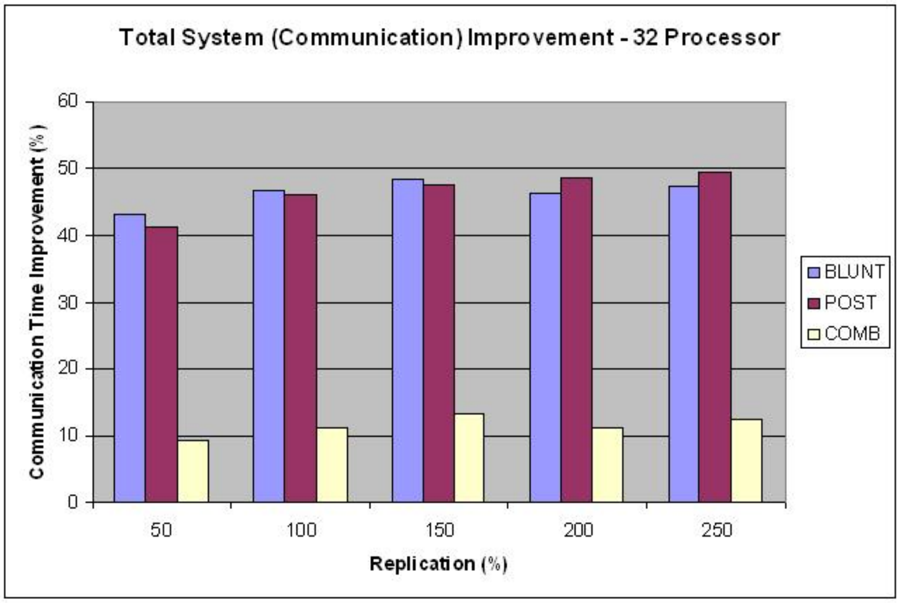


Figure 5.9: Total Execution Time Improvement with Proposed Network Flow Algorithm and INSH - 32 Processor

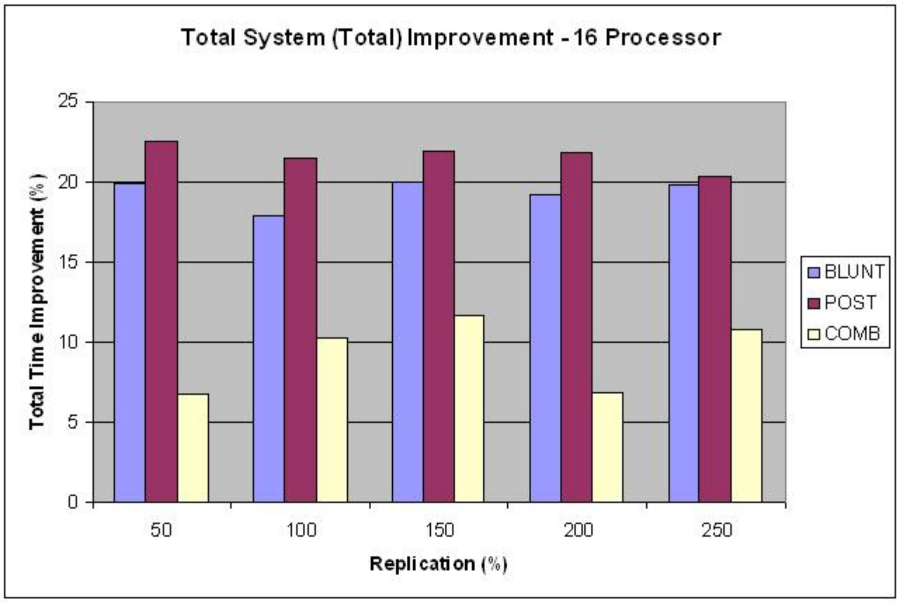


Figure 5.10: Total Execution Time Improvement with Proposed Network Flow Algorithm and INSH - 16 Processor

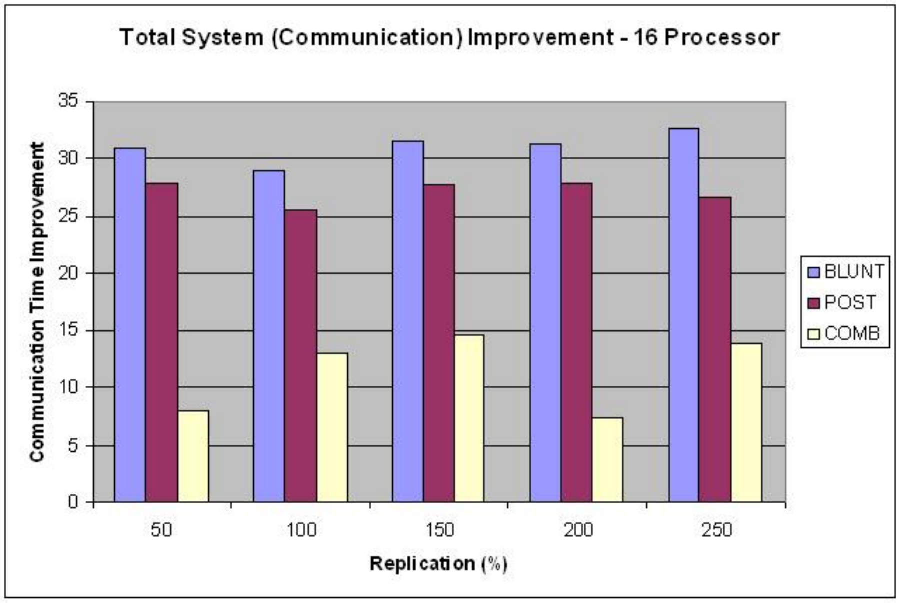


Figure 5.11: Migration Time Improvement with Proposed Network Flow Algorithm and INSH - 16 Processor

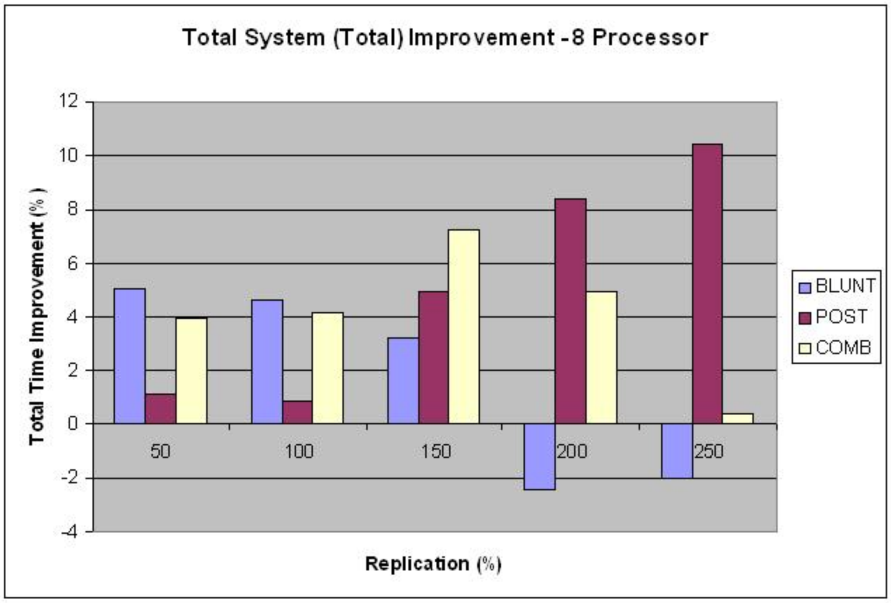


Figure 5.12: Total Execution Time Improvement with Proposed Network Flow Algorithm and INSH - 8 Processor

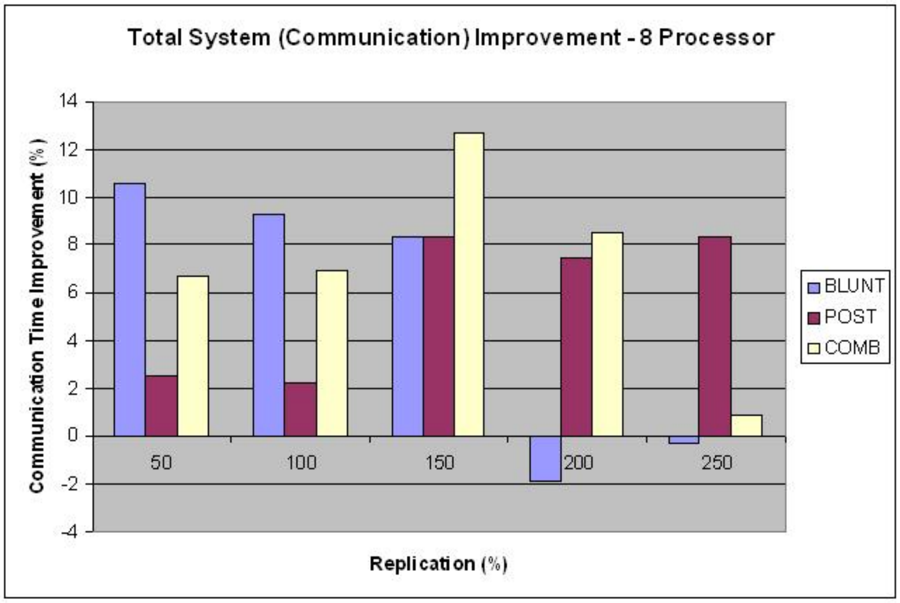


Figure 5.13: Migration Time Improvement with Proposed Network Flow Algorithm and INSH - 8 Processor

5.6 A Complete Set of Experimental Data

Total time, migration time and rendering time is defined as the maximum time spent by any processor during respective phases. Rendering imbalance between processors is defined as a ratio of maximum rendering load of a processor to average rendering load. Migration imbalance is defined as the ratio of maximum effective communication volume to average of effective communication volume where effective communication volume is defined as the maximum of send and receive volumes of processors. Selection of such migration imbalance metric is needed because in two port communication model maximum of send or receive volume dominates the communication time.

Abreviation used in this section are:

e INSH scaling factor

R Replication percentage

net Network flow present bit

mode Network flow + INSH present bit

tt Total execution time (second)

mt Migration time (second)

rt Rendering time (second)

mi Migration imbalance

ri Rendering imbalance

msv Maximum send volume handled by a processor (MB)

mrv Maximum receive volume handled by a processor (MB)

mtv Maximum total volume handled by a processor (MB)

ttimp Improvement on total execution time (%)

mtimp Improvement on migration time (%)

msvimp Improvement on maximum send volume handled by a processor (%)

mtvimp Improvement on maximum total volume handled by a processor (%)

5.6.1 Inverse Net Size Heuristic

		e	tt	mt	rt	mi	ri	msv	mrv
32-Processors	BLUNT	0	1.26	0.80	0.15	2.94	1.20	4.31	7.99
		10	1.02	0.48	0.22	2.18	1.79	3.68	3.54
		20	1.09	0.48	0.30	2.04	2.43	3.47	3.36
		30	1.17	0.48	0.37	1.99	3.08	3.40	3.29
		40	1.25	0.49	0.45	1.99	3.69	3.38	3.25
		50	1.34	0.49	0.54	1.99	4.42	3.37	3.27
		60	1.38	0.49	0.58	1.98	4.76	3.37	3.24
		70	1.43	0.49	0.63	1.98	5.22	3.36	3.22
		80	1.48	0.48	0.69	1.98	5.73	3.35	3.24
	POST	0	2.56	2.09	0.20	2.39	1.20	7.37	20.58
		10	2.02	1.46	0.29	2.35	1.83	9.64	8.96
		20	2.12	1.46	0.38	2.39	2.51	10.17	8.37
		30	2.18	1.45	0.45	2.38	3.05	10.18	8.33
		40	2.26	1.45	0.53	2.37	3.56	10.23	8.32
		50	2.32	1.44	0.60	2.39	4.03	10.27	8.27
		60	2.40	1.44	0.67	2.39	4.50	10.30	8.23
		70	2.43	1.43	0.72	2.39	4.83	10.23	8.32
		80	2.48	1.43	0.77	2.40	5.13	10.30	8.24
	COMB	0	0.88	0.46	0.15	1.51	1.09	2.93	3.20
		10	0.90	0.47	0.16	1.54	1.17	3.05	3.16
		20	0.92	0.48	0.18	1.56	1.27	3.12	3.11
		30	0.93	0.48	0.19	1.57	1.37	3.14	3.05
		40	0.94	0.48	0.20	1.58	1.44	3.16	3.04
		50	0.96	0.48	0.21	1.60	1.50	3.20	3.05
		60	0.97	0.49	0.22	1.59	1.56	3.22	3.07
		70	0.97	0.49	0.22	1.61	1.60	3.24	3.09
		80	0.98	0.49	0.23	1.60	1.62	3.23	3.06

Table 5.6: 32-Processor statistics with changing INSH scaling factor

		e	tt	mt	rt	mi	ri	msv	mrv
16-Processors	BLUNT	0	1.29	0.89	0.27	2.40	1.13	4.81	8.85
		10	1.13	0.68	0.33	2.11	1.35	4.58	5.60
		20	1.15	0.65	0.38	2.00	1.56	4.47	4.85
		30	1.18	0.63	0.43	1.91	1.79	4.42	4.51
		40	1.23	0.63	0.48	1.90	2.00	4.44	4.30
		50	1.28	0.63	0.53	1.90	2.21	4.50	4.24
		60	1.32	0.62	0.57	1.89	2.42	4.43	4.14
		70	1.36	0.62	0.62	1.87	2.60	4.48	4.14
		80	1.40	0.62	0.66	1.87	2.79	4.46	4.11
	POST	0	2.86	2.36	0.37	1.91	1.15	8.19	23.02
		10	2.41	1.87	0.41	1.92	1.30	10.07	15.16
		20	2.34	1.75	0.46	1.95	1.51	10.94	12.88
		30	2.35	1.70	0.52	1.97	1.72	11.34	11.81
		40	2.45	1.74	0.59	2.00	1.94	11.77	11.25
		50	2.48	1.73	0.63	2.03	2.12	12.22	11.04
		60	2.54	1.74	0.68	2.07	2.28	12.32	10.92
		70	2.60	1.75	0.73	2.04	2.46	12.39	10.57
		80	2.64	1.75	0.77	2.04	2.60	12.56	10.53
	COMB	0	1.05	0.66	0.29	1.56	1.06	4.13	4.52
		10	1.06	0.66	0.29	1.57	1.06	4.17	4.56
		20	1.07	0.66	0.31	1.57	1.11	4.22	4.45
		30	1.08	0.66	0.31	1.58	1.14	4.23	4.50
		40	1.07	0.66	0.31	1.58	1.14	4.28	4.50
		50	1.08	0.66	0.32	1.58	1.17	4.28	4.49
		60	1.09	0.66	0.33	1.59	1.19	4.32	4.49
		70	1.10	0.67	0.33	1.60	1.21	4.35	4.48
		80	1.10	0.66	0.34	1.59	1.22	4.36	4.43

Table 5.7: 16-Processor statistics with changing INSH scaling factor.

		e	tt	mt	rt	mi	ri	msv	mrv
8-Processors	BLUNT	0	1.47	0.86	0.52	1.88	1.07	5.03	7.99
		10	1.47	0.82	0.55	1.81	1.15	5.03	7.72
		20	1.46	0.79	0.57	1.81	1.19	5.11	7.29
		30	1.52	0.76	0.65	1.78	1.33	5.14	6.75
		40	1.48	0.76	0.63	1.73	1.33	5.11	6.53
		50	1.50	0.75	0.66	1.69	1.40	5.02	6.19
		60	1.51	0.72	0.69	1.68	1.46	4.95	5.97
		70	1.55	0.74	0.72	1.66	1.53	4.94	5.87
		80	1.57	0.72	0.75	1.65	1.59	4.96	5.67
	POST	0	2.99	2.17	0.70	1.82	1.11	9.85	23.36
		10	2.85	2.03	0.70	1.82	1.12	10.69	21.41
		20	2.84	1.98	0.74	1.80	1.20	11.13	19.86
		30	2.85	1.93	0.81	1.79	1.31	11.55	18.70
		40	2.84	1.91	0.81	1.80	1.35	11.94	17.28
		50	2.82	1.86	0.85	1.78	1.42	12.05	16.37
		60	2.84	1.85	0.88	1.76	1.49	12.11	15.60
		70	2.85	1.82	0.91	1.75	1.55	12.24	15.16
		80	2.90	1.84	0.95	1.74	1.63	12.53	15.01
	COMB	0	1.48	0.83	0.57	1.47	1.05	5.56	5.89
		10	1.48	0.82	0.57	1.47	1.04	5.55	5.85
		20	1.48	0.83	0.57	1.46	1.04	5.54	5.86
		30	1.49	0.84	0.57	1.47	1.05	5.63	5.97
		40	1.52	0.84	0.60	1.47	1.08	5.61	5.85
		50	1.51	0.85	0.58	1.46	1.06	5.61	5.87
		60	1.50	0.83	0.59	1.48	1.07	5.69	5.81
		70	1.52	0.85	0.59	1.46	1.08	5.61	5.83
		80	1.58	0.88	0.61	1.48	1.10	5.74	5.81

Table 5.8: 8-Processor statistics with changing INSH scaling factor.

5.6.2 Effect of Replication

		R(%)	tt	mt	mtv	ttimp	mtimp	mtvimp
32-Processors	BLUNT	0	0.91	0.51	6.06	-	-	-
		50	0.89	0.49	5.95	2.49	4.25	1.68
		100	0.86	0.46	5.61	5.45	9.85	7.42
		150	0.86	0.46	5.47	5.97	10.53	9.59
		200	0.85	0.45	5.41	7.15	12.98	10.61
		250	0.83	0.43	5.25	9.46	17.14	13.28
		300	0.80	0.40	4.94	12.17	21.66	18.43
		350	0.81	0.40	4.92	11.95	21.90	18.77
	POST	0	1.99	1.57	16.60	-	-	-
		50	1.94	1.51	16.00	2.79	3.49	3.57
		100	1.89	1.47	15.42	5.13	6.50	7.09
		150	1.82	1.39	14.62	8.63	11.17	11.90
		200	1.78	1.36	14.09	10.47	13.29	15.11
		250	1.74	1.32	13.30	12.43	15.86	19.89
		300	1.71	1.28	12.73	14.12	18.27	23.32
		350	1.70	1.27	12.73	14.52	18.82	23.31
	COMB	0	0.76	0.49	5.67	-	-	-
		50	0.76	0.49	5.63	0.72	0.47	0.70
		100	0.74	0.46	5.45	3.66	5.12	4.04
		150	0.72	0.45	5.33	5.15	7.82	6.11
		200	0.71	0.44	5.13	6.63	10.81	9.54
		250	0.70	0.42	4.95	8.37	13.25	12.85
		300	0.69	0.41	4.89	9.17	15.19	13.76
		350	0.68	0.40	4.72	10.50	17.94	16.75

Table 5.9: 32-Processor statistics showing effects of replication.

		R(%)	tt	mt	mtv	ttimp	mtimp	mtvimp
16-Processors	BLUNT	0	1.19	0.79	8.50	-	-	-
		50	1.12	0.72	7.74	5.98	8.99	8.94
		100	1.09	0.67	7.43	8.62	15.75	12.53
		150	1.02	0.63	6.83	14.11	20.83	19.57
		200	0.97	0.57	6.35	18.47	28.09	25.23
		250	0.94	0.55	6.10	20.86	31.18	28.23
		300	0.91	0.52	5.82	23.40	34.98	31.46
		350	0.87	0.48	5.56	26.57	39.66	34.57
	POST	0	2.57	2.09	24.77	-	-	-
		50	2.47	1.99	23.65	3.81	4.87	4.52
		100	2.32	1.84	22.62	9.61	12.01	8.67
		150	2.27	1.79	21.47	11.67	14.68	13.31
		200	2.15	1.66	19.86	16.49	20.58	19.81
		250	2.01	1.52	18.16	21.83	27.33	26.66
		300	1.92	1.43	17.55	25.49	31.69	29.13
		350	1.92	1.42	16.97	25.49	31.98	31.49
	COMB	0	1.05	0.70	8.39	-	-	-
		50	1.03	0.69	8.27	1.12	1.90	1.46
		100	1.01	0.66	7.96	3.64	5.68	5.11
		150	0.98	0.63	7.66	6.43	9.91	8.73
		200	0.94	0.59	7.14	10.49	16.50	14.90
		250	0.92	0.57	6.95	12.20	18.92	17.20
		300	0.89	0.54	6.48	14.85	23.38	22.78
		350	0.86	0.51	6.10	17.31	27.30	27.30

Table 5.10: 16-Processor statistics showing effects of replication.

		R(%)	tt	mt	mtv	ttimp	mtimp	mtvimp
8-Processors	BLUNT	0	1.59	1.00	10.76	-	-	-
		50	1.55	0.88	9.93	2.80	11.85	7.73
		100	1.41	0.82	9.18	11.25	18.48	14.68
		150	1.32	0.72	8.29	17.09	27.70	23.00
		200	1.27	0.67	7.40	20.27	33.01	31.21
		250	1.19	0.59	6.58	25.28	40.76	38.88
		300	1.13	0.54	5.77	28.73	46.76	46.36
		350	1.03	0.43	4.92	35.09	56.56	54.30
	POST	0	3.20	2.43	26.19	-	-	-
		50	2.96	2.19	24.33	7.37	9.99	7.11
		100	2.78	2.00	22.25	13.15	17.71	15.05
		150	2.64	1.86	20.71	17.44	23.28	20.94
		200	2.46	1.68	18.77	23.09	30.92	28.35
		250	2.31	1.52	16.57	27.86	37.34	36.73
		300	2.15	1.36	14.87	32.86	44.03	43.22
		350	1.96	1.17	12.95	38.81	52.03	50.55
	COMB	0	1.61	0.99	12.18	-	-	-
		50	1.52	0.90	11.22	5.41	8.98	7.85
		100	1.43	0.81	10.21	11.20	18.33	16.14
		150	1.37	0.76	9.24	14.56	23.93	24.13
		200	1.30	0.68	8.37	19.22	31.91	31.25
		250	1.22	0.60	7.41	24.13	39.63	39.20
		300	1.16	0.54	6.53	27.61	45.47	46.43
		350	1.11	0.48	6.03	30.91	51.28	50.46

Table 5.11: 8-Processor statistics showing effects of replication.

5.6.3 Network Flow Algorithm Improvement

		R(%)	net	mt	msv	mtimp	msvimp
32-Processors	BLUNT	50	0	0.49	3.95	5.26	35.65
			1	0.47	2.54		
		100	0	0.46	3.61	7.74	46.39
			1	0.43	1.94		
		150	0	0.46	3.46	10.71	50.74
			1	0.41	1.70		
		200	0	0.45	3.21	9.02	47.64
			1	0.41	1.68		
		250	0	0.43	3.04	7.03	47.58
			1	0.40	1.60		
	POST	50	0	1.51	10.06	15.26	36.13
			1	1.28	9.32		
		100	0	1.47	9.57	23.57	50.10
			1	1.12	8.93		
		150	0	1.39	9.12	23.90	55.84
			1	1.06	8.48		
		200	0	1.36	8.93	25.05	55.59
			1	1.02	8.31		
		250	0	1.32	8.50	27.49	56.18
			1	0.96	8.08		
	COMB	50	0	0.49	3.01	9.40	20.87
			1	0.44	2.38		
		100	0	0.46	2.93	11.14	31.05
			1	0.41	2.02		
		150	0	0.45	2.87	13.28	32.67
			1	0.39	1.93		
		200	0	0.44	2.83	11.23	34.51
			1	0.39	1.85		
		250	0	0.42	2.76	12.49	36.10
			1	0.37	1.76		

Table 5.12: 32-Processor statistics showing improvements of network flow.

		R(%)	net	mt	msv	mtimp	msvimp
16-Processors	BLUNT	50	0	0.72	5.12	9.96	42.93
			1	0.65	2.92		
		100	0	0.67	4.50	5.39	48.06
			1	0.63	2.34		
		150	0	0.63	4.17	7.78	49.03
			1	0.58	2.13		
		200	0	0.57	3.41	2.31	43.63
			1	0.56	1.92		
		250	0	0.55	3.25	4.63	44.78
			1	0.52	1.79		
	POST	50	0	1.99	10.72	9.77	32.34
			1	1.80	7.25		
		100	0	1.84	9.78	5.46	42.15
			1	1.84	5.66		
		150	0	1.79	9.15	10.13	43.73
			1	1.79	5.15		
		200	0	1.66	8.34	8.33	44.80
			1	1.66	4.61		
		250	0	1.52	7.29	4.02	38.88
			1	1.52	4.46		
	COMB	50	0	0.69	4.32	7.98	23.64
			1	0.64	3.30		
		100	0	0.66	4.12	12.92	32.20
			1	0.58	2.80		
		150	0	0.63	4.00	14.69	35.03
			1	0.54	2.60		
		200	0	0.59	3.82	7.38	35.12
			1	0.54	2.48		
		250	0	0.57	3.65	13.84	37.41
			1	0.49	2.28		

Table 5.13: 16-Processor statistics showing improvements of network flow.

		R(%)	net	mt	msv	mtimp	msvimp
8-Processors	BLUNT	50	0	0.88	5.87	3.13	33.56
			1	0.86	3.90		
		100	0	0.82	4.94	5.05	31.96
			1	0.78	3.36		
		150	0	0.72	4.02	1.91	37.65
			1	0.71	2.51		
		200	0	0.67	3.46	0.55	28.23
			1	0.67	2.48		
		250	0	0.59	2.89	-1.69	27.09
			1	0.60	2.11		
	POST	50	0	2.19	12.02	-3.14	28.47
			1	2.26	8.60		
		100	0	2.00	10.23	-5.27	35.56
			1	2.10	6.59		
		150	0	1.86	8.76	2.93	29.14
			1	1.81	6.21		
		200	0	1.68	6.93	-1.79	27.90
			1	1.71	5.00		
		250	0	1.52	5.75	1.34	26.11
			1	1.50	4.25		
	COMB	50	0	0.90	6.09	6.71	21.53
			1	0.84	4.78		
		100	0	0.81	5.52	6.95	28.80
			1	0.75	3.93		
		150	0	0.76	5.13	12.70	33.41
			1	0.66	3.42		
		200	0	0.68	4.51	8.48	29.71
			1	0.62	3.17		
		250	0	0.60	3.97	0.85	31.19
			1	0.59	2.73		

Table 5.14: 8-Processor statistics showing improvements of network flow.

5.6.4 Total System Improvement

		R(%)	mod	tt	mt	ttimp	mtimp
32-Processors	BLUNT	50	0	1.14	0.82	33.15	43.25
			1	0.76	0.47		
		100	0	1.14	0.80	37.77	46.74
			1	0.71	0.43		
		150	0	1.11	0.80	38.52	48.34
			1	0.69	0.41		
		200	0	1.09	0.76	39.02	46.15
			1	0.66	0.41		
		250	0	1.07	0.75	38.28	47.44
			1	0.66	0.40		
	POST	50	0	2.51	2.18	36.04	41.15
			1	1.61	1.28		
		100	0	2.41	2.07	40.15	46.03
			1	1.44	1.12		
		150	0	2.35	2.02	41.16	47.51
			1	1.38	1.06		
		200	0	2.31	1.98	42.94	48.56
			1	1.32	1.02		
		250	0	2.23	1.89	43.38	49.45
			1	1.26	0.96		
	COMB	50	0	0.76	0.49	17.29	9.40
			1	0.63	0.44		
		100	0	0.74	0.46	19.38	11.14
			1	0.59	0.41		
		150	0	0.72	0.45	22.09	13.28
			1	0.56	0.39		
		200	0	0.71	0.44	22.89	11.23
			1	0.55	0.39		
		250	0	0.70	0.42	21.71	12.49
			1	0.55	0.37		

Table 5.15: 32-Processor statistics of total system improvement(net+INSH).

		R(%)	mod	tt	mt	ttimp	mtimp
16-Processors	BLUNT	50	0	1.29	0.94	19.94	17.93
			1	1.03	0.65		
		100	0	1.24	0.89	17.93	28.96
			1	1.02	0.63		
		150	0	1.20	0.85	20.04	31.56
			1	0.96	0.58		
		200	0	1.16	0.81	19.20	31.31
			1	0.94	0.56		
		250	0	1.12	0.77	19.84	32.60
			1	0.90	0.52		
	POST	50	0	2.93	2.49	22.50	27.90
			1	2.27	1.80		
		100	0	2.82	2.34	21.49	25.63
			1	2.21	1.74		
		150	0	2.66	2.22	21.94	27.76
			1	2.08	1.61		
		200	0	2.55	2.11	21.81	27.91
			1	1.99	1.52		
		250	0	2.43	1.99	20.38	26.60
			1	1.93	1.46		
	COMB	50	0	1.03	1.69	6.73	7.98
			1	0.96	0.64		
		100	0	1.01	0.66	10.28	12.92
			1	0.90	0.58		
		150	0	0.98	0.63	11.60	14.69
			1	0.86	0.54		
		200	0	0.94	0.59	6.78	7.38
			1	0.87	0.54		
		250	0	0.92	0.57	10.85	13.84
			1	0.82	0.49		

Table 5.16: 16-Processor statistics of total system improvement(net+INSH).

		R(%)	mod	tt	mt	ttimp	mtimp
8-Processors	BLUNT	50	0	1.53	0.96	5.04	10.56
			1	1.45	0.86		
		100	0	1.43	0.85	4.63	9.30
			1	1.36	0.78		
		150	0	1.35	0.77	3.21	8.30
			1	1.30	0.71		
		200	0	1.23	0.65	-2.44	-1.88
			1	1.26	0.67		
		250	0	1.17	0.60	-2.01	-0.28
			1	1.20	0.60		
	POST	50	0	3.07	2.31	1.12	2.54
			1	3.03	2.26		
		100	0	2.91	2.15	0.90	2.20
			1	2.88	2.10		
		150	0	2.73	1.97	4.97	8.30
			1	2.59	1.81		
		200	0	2.72	1.85	8.36	7.44
			1	2.49	1.71		
		250	0	2.55	1.64	10.44	8.30
			1	2.28	1.50		
	COMB	50	0	1.52	1.90	3.93	6.71
			1	1.46	0.84		
		100	0	1.43	0.81	4.15	6.95
			1	1.37	0.75		
		150	0	1.37	0.76	7.23	12.70
			1	1.27	0.66		
		200	0	1.30	0.68	4.94	8.48
			1	1.23	0.62		
		250	0	1.22	0.60	0.40	0.85
			1	1.21	0.59		

Table 5.17: 8-Processor statistics of total system improvement(net+INSH).

Chapter 6

Conclusion

Interactive direct volume rendering of large datasets still stands to be a hard problem especially with growing trend of dataset sizes. We believe for interactive direct volume rendering, use of state of the art algorithms, parallel processing and hardware assisted rendering is needed. Furthermore, unutilized system resources can be used to increase performance, leading lo lower execution times and interactivity. In this thesis, we tried to exploit the use of excess system memory to increase the performance for big scale volume rendering applications.

6.1 Work Carried Out

Hypergraph partitioning based remapping problem addresses the problem of flexibly assigning the pixel blocks to processors, without any predetermined restrictions, which helps to decrease the total volume of communication in the system. Furthermore, more balanced computation loads can be distributed via usage of this method. We tried to extend this flexibility to communication operations where processors are allowed to store replicated data.

Replication of the dataset in the system has two advantages. Firstly, replicated data effectively achieves avoidance of communication. If data-to-processor mapping dictates replicated data is the subject of a communication operation, no communication is needed. This is the natural effect of replication and enhanced by the extended replication-HP model where replicated data is taken into consideration to minimize the total communication volume. Therefore, reported 25% less communication volumes achieved by old HP-based remapping model is preserved for replication-HP model.

Secondly, replicated data presents the flexibility to assign a send operation to processors, which store the replicated data. This flexibility is used to balance send message volumes of processors or total message volumes of processors. Balancing send message volumes of processors corresponds to balancing maximum of send message volume and receive message volume, whereas balancing total message volumes of processors corresponds to balancing total of send message volume and receive message volume; both of which has suitable corresponding parallel architectures to be used. To balance send/total message volumes of processors, we have utilized a maximum flow network algorithm with parametric search, where optimal balance is found with nearly negligible processing cost with equal sized data primitives. Considerably better communication load balances were achieved and up to 50% savings for maximum send/total message volume handled by a processor is obtained. This savings transforms well into communication phase speedups with high processor numbers, which proves the validity of our algorithms to utilize excess memory for big scale volume rendering applications.

INSH is proposed to overcome the irregularity effects of the dataset, where vertex weights of the replication hypergraph are modified to encapsulate the effects of communication operations. More than 100% improvements on maximum receive volume handled by a processor is achieved for irregular datasets for high processor numbers with considerably less increase in rendering imbalance.

Maximum network flow based parametric search algorithm DELNET to delete replicas is proposed to round up three modes of network flow algorithms. DELNET increases the flexibility to assign send operations to processors via enforcing a lower bound on replication count of data primitives. So for each data primitive, there will be at least a number of processors to assign a send operation regarding the data primitive. Although it is not explicitly shown in this thesis, DELNET has the effect of decreasing the maximum total message volume handled by a processor considerably mainly because there is less flexibility to balance the total message volume handled by processors.

Effect of these algorithms is tested in image-space parallelization of a DVR algorithm and up to total 45% improvement in total time is achieved, excluding the communication avoidance effects of replication, especially for irregular datasets and high number of processors.

6.2 Future Work

Proposed INSH is an approximation of actually balancing receive volumes of processors and rendering loads of processors separately. We have proposed such algorithm because, to the best of our knowledge, state of the art HP tools does not support multi-constraint with fixed vertices formulation. Production of such tool could produce superior timing results. Finally, applying proposed methods to object-space parallelization could be an interesting future work since similar kinds of problems may arise in object-space parallelization of DVR algorithms.

Bibliography

[1] C. Aykanat, B.B. Cambazoglu, F. Findik, and T. Kurc, "Adaptive Decomposition and Remapping Algorithms for Object-Space-Parallel Direct Volume Rendering of Unstructured Grids", *J. Parallel and Distributed Computing*, vol. 67, no. 1, pp. 77–99, 2007.

[2] B.B. Cambazoğlu and C. Aykanat, "Hypergraph-Partitioning-Based Remapping Models for Image-Space-Parallel Direct Volume Rendering of Unstructured Grids", *IEEE Trans. Parallel and Distributed Systems*, vol. 18, no. 1, pp. 1–14, 2007.

[3] U.V. Catalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R.T. Heaphy, and L.A. Riesen, "Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations", in Proc. IPDPS'07, pp. 1–11, 2007.

[4] A. Pinar and B. Hendrickson, "Improving Load Balance with Flexibly Assignable Tasks", *IEEE Trans. Parallel and Distributed Systems*, vol. 16, no. 10, pp. 956–965, 2005.

[5] B. B. Cambazoğlu, A Hypergraph-Partitioning Based Remapping Model for Image-Space Parallel Volume Rendering, MSc Thesis, Bilkent University, Department of Computer Engineering, 2000.

[6] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser, "Smart hardware-accelerated volume rendering", in *VisSym'03: Proc. of the symposium on Data Visualization 2003*, pp. 231–238, 2003.

[7] F. Ino, T. Sasaki, A. Takeuchi, and K. Hagihara, “A Divided-Screenwise Hierarchical Compositing for Sort-Last Parallel Volume Rendering”, in Proc. IPDPS’03, pp. 14–1, 2003.

[8] R. Samanta, T. Funkhouser,and K. Li, “Parallel Rendering with K-way Replication”, in *Proc. IEEE Symposium on Parallel and Large Data Visualization and Graphics*, pp. 75–84, San Diego, CA.

[9] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. P. Singh, “Load Balancing for Multi-Projector Rendering Systems”, in *Proc. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp. 107–116, Los Angeles, CA.

[10] D. Weiskopf, M. Weiler, and T. Ertl, “Maintaining Constant Frame Rates in 3D Texture-Based Volume Rendering”, in *Computer Graphics International (CGI’04)*, pp. 604–607, 2004.

[11] M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, and T. Ertl, “Level-of-Detail Volume Rendering via 3D Textures”, in *Proc. Volume Visualization and Graphics Symposium*, pp. 7–13, 2000.

[12] E. Lamar, B. Hamann, and K. I. Joy, “Multiresolution Tecniques for Interactive Texture-Based Volume Visualization”, in *Proc. IEEE Visualization ’99*, pp. 355–361, 1999.

[13] J. Krüger, and R. Westermann, “Acceleration Tecniques for GPU-based Volume Rendering”, in *Proc. IEEE Visualization*, pp. 287–292, 2003.

[14] S. Guthe, M. Wand, J. Gonser, and W. Strasser, “Interactive Rendering of Large Volume Data Sets”, in *Proc. IEEE Visualization*, pp. 53–60, 2002.

[15] T. Cullip, and U. Neumann, “Accelerating Volume Reconstruction with 3D Texture Hardware”, in Technical Report TR93-027, 1993.

[16] B. Cabral, N. Cam, and J. Foran, “Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware”, in *Proc. of the 1994 symposium on Volume Visualization*, pp. 91–98, 1994.

[17] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A Sorting Classification of Parallel Rendering", in *IEEE Comput. Graph. Appl.*, vol. 14, no. 4, pp. 23–32, 1994.

[18] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh, "Hybrit Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs", in *Proc. Eurographics / ACM SIGGRAPH Workshop on Graphics Hardware*, pp. 97–108, Los Angles, CA, 2000.

[19] A. Garcia, and H.-W. Shen, "An Interleaved Parallel Volume Renderer with PC-Clusters", in *Proc. Eurographics Workshop on Parallel Graphics and Visualization (EGPGV)*, pp. 51–59, 2002.

[20] M. Magallon, M. Hopf, T. Ertl, "Parallel Volume Rendering Using PC graphics Hardware", in *Pacific Conference on Computer Graphics and Applications*, pp. 384–389, 2001.

[21] I. Ihm, and S. Park, "Wavelet-Based 3D Compression Scheme for Very Large Volume Data", in *Graphics Interface '98*, pp. 107–116, 1998.

[22] T.-Y. Kim, and Y. G. Shin, "An Efficient Wavelet-Based Compresion Method for Volume Rendering", in *Pacific Graphics '99 (1999)*, pp. 147–157.

[23] S. Muraki, "Approximation and Rendering of Volume Data Using Wavelet Transforms", in *IEEE Visualization '92*, pp. 21–28.

[24] A. Vilanova, and D. Ruijters, "Optimizing GPU Volume Rendering", in *WSCG - Winter School of Computer Graphics*, 2006.

[25] M. Hadwiger, C. Berger, and H. Hauser, "High-Quality Two-Level Volume Rendering of Segmented Data Sets on Consumer Graphics Hardware", in *Proc. IEEE Visualization 2003*, pp. 301–308, 2003.

[26] I. Viola, A. Kanitsar, and M. E. Gröller, "Importance-Driven Volume Rendering", in *Proc. IEEE Visualization 2004*, pp. 139–145, 2004.

[27] U. Bordoloi, and H.-W. Shen, "View selection for Volume Rendering", in *Proc. IEEE Visualization*, pp. 487–494, 2004.

[28] S. Guthe, and W. Straßer, “Advanced Techniques for High-Quality Multi-Resolution Volume Rendering”, in *Computers & Graphics, Volume 28, Issue 1*, pp. 51–58, 2004.

[29] L. Wang, Y. Zhao, K. Mueller, and A. Kaufman, “The magic Volume Lens: An interactive Focus+Context Technique for Volume Rendering”, in *Proc. IEEE Visualization*, pp. 367–374, 2005.

[30] C. Müller, M. Strengert, and T. Ertl, “Optimized Volume Raycasting for Graphics-Hardware-Based Cluster Systems”, in *Proc. EG Symp. Parallel Graphics Vis. (PGV)*, pp. 59–66, 2006.

[31] S. Bachthaler, M. Strengert, D. Weiskopf, T. Ertl, “Parallel Texture-based Vector Field Visualization on Curved Surfaces Using GPU Cluster Computers”, in *Eurographics Symposium on Parallel Graphics Visualization 2006*, pp. 75–82.

[32] S. Marchesin, C. Mongenet, and J.-M. Dischler, “Dynamic Load Balancing for Parallel Volume Rendering”, in *Eurographics Symposium on Parallel Graphics Visualization 2006*, pp. 43–50.

[33] J. Gao, C. Wang, L. Li, and H.-W. Shen, “A Parallel Multiresolution Volume Rendering Algorithm for Large Data Visualization”, in *Parallel Computing*, vol. 31, no. 2, pp. 185–204, 2005.

[34] C. Wang, J. Gao, Liya Li, and Han-Wei Shen, “A Multiresolution Volume Rendering Framework for Large-Scale Time-Varying Data Visualization”, in *Proc. Eurographics/IEEE VGTC Workshop Volume Graphics*, pp. 11–19, 2005.

[35] N. Fout, and K.-L. Ma, “Transform Coding for Hardware-accelerated Volume Rendering”, in *IEEE Transactions on Visualization and Computer Graphics* , vol. 13, no. 6, pp. 1600–1607, 2007.

[36] X. Cavin, C. Mion, and A. Filbois, “COTS Cluster-based Sort-last Rendering: Performance Evaluation and Pipelined Implementation”, in *Proc. IEEE Visualization 2005 Conference*, pp. 111–118.

[37] A.Ghosh, P. Prabhu, A. E. Kaufman, and K. Mueller, "Hardware Assisted Multichannel Volume Rendering", in *Compter Graphics International*, pp. 2–7,2003.

[38] NASA Data Set Archive, "http://www.nas.nasa.gov/Research/Datasets/datasets.html", 2004.

[39] M. P. Garrity, "Ray-Tracing Irregular Volume Data", in *ACM SIGGRAPH Computer Graphics*, vol. 24, no. 5, pp. 35–40, 1990.

[40] P. Shirley, and A. Tuchman, "A Polygonal Approximation to Direct Scalar Volume Rendering", in *ACM SIGGRAPH Computer Graphics*, vol. 24, no. 5, pp. 63–70, 1990.

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

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

[43] W. E. Lorensen, and H. E. Cline, Marching Cubes: A High Resolution 3D Surface Construction Algorithm, in *Computer Graphics (Proc. SIGGRAPH)*, pp. 163–169, 1987.

[44] T. T. Elvins, A Survey of Algorithms for Volume Visualization, in *Computer Graphics (ACM Siggraph Quarterly)*, vol. 26 no. 3 pp. 194–201, 1992.

[45] H. Kutluca, T. M. Kurç, and C. Aykanat, Image-Space Decomposition Algorithms for Sort-First Parallel Volume Rendering of Unstructured Grids, in *J. Supercomputing* , vol. 15, no. 1, pp. 51–93, 2000.

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

[47] Ü. V. Çatalyürek, and C. Aykanat, Hypergraph-Partitioning Based Decomposition for Parallel Sparse-Matrix Vector Multiplication, in *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, 1999.

[48] Paraview Data Formats,
"https://visualization.hpc.mil/wiki/Paraview_Data_Formats", 2008.

[49] A. Goldberg, and R. E. Tarjan, A New Approach to the Maximum Flow Problem, in *Journal of the ACM*, vol. 35, pp. 921–940, 1988.

[50] Network Optimization Library, "http://avglab.com/andrew/soft.html",2009.

[51] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Chichester, U.K.: Wiley-Teubner, 1990.

[52] C. Aykanat, A.Pinar, and Ü. V. Çatalyürek, Permuting Sparse Rectangular Matrices into Block-Diagonal Form, in *SIAM J. Scientific Computing*, vol. 25, no. 6, pp. 1860–1879, 2004.

[53] C. Aykanat, A.Pinar, and Ü. V. Çatalyürek, Permuting Sparse Rectangular Matrices into Block-Diagonal Form, in *SIAM J. Scientific Computing*, vol. 25, no. 6, pp. 1860–1879, 2004.

[54] B. Uçar, and C. Aykanat, Encapsulating Multible Communication-Cost Metrics in Partitioning Sparse Rectangular Matrices for Parallel Matrix-Vector Multiplies, in *SIAM J. Scientific Computing*, vol. 25, no. 6, pp. 1837–1859, 2004.