Universidade Federal Fluminense

BERNARDO BULGARELLI LABRONICI

Efficient High Resolution Volumetric Parallel Rendering of Unstructured Data on a Cluster of Multicores

NITERÓI

2010

BERNARDO BULGARELLI LABRONICI

Efficient High Resolution Volumetric Parallel Rendering of Unstructured Data on a Cluster of Multicores

Advisor:

Lúcia Maria de Assumpção Drummond

Co-advisor:

Cristiana Bentes

Universidade Federal Fluminense

NITERÓI

2010

Efficient High Resolution Volumetric Parallel Rendering of Unstructured Data on a Cluster of Multicores

Bernardo Bulgarelli Labronici

Master dissertation submitted to the Programa de Pós-graduação em Computaçào of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master in Science.

Approved by:

Niterói,5 of July of 2010.

Experience is not what happens to a man.

It is what a man does with what happens to him.

Aldous Huxle

Acknowledgments

I first thanks my Master advisor and Co-advisor, Professors Lúcia Drummond and Cristiana Bentes, for supporting and encouraging me before and during these past two years. Their patience, guidance and incentive make me want to give my best.

I want to thanks Professor Ricardo Farias, although he was not officially one of my advisors, he truly act as if he was. Thanks for all the help given. I have a great debit with him, and I intent to pay with some more Jambalaya with Turbo Dog one of this days.

Thanks to my family that have supported my decisions specially my mom, Vera, my step mom, Ana, and step father, Miguel. My grandma, Lúcia, for always care and remember me. My brother to always be "ready for the fun" when I need it, also wish good luck to him, since he is enduring the same journey as I had to earn his Master degree. My grandfather Domingos, although he is no longer alive, I am sure that some how he is still looking over me.

I thanks my dad, José Eduardo, for all my life he help me with what ever I needed to get until here. These few lines are not enough to show the gratitude that I have for him. I hope to make him very proud and honor every thing he has already sacrificed for me. He was and still is the model that I follow.

The most special thank to my wife, Eliana, without your support I would be no were near the end of this dissertation. She supported and stayed of my side each canceled trip and each weekend spent at home. I love you very much.

Thanks to CNPQ, CAPES, IEEE and UFF for all the financial support. Without this kind of support almost no one would be able to make good academic publication. I hope that those institutions keep these excellent work for years to come, and many other follow their example.

Thank for the RTM petrobrás group, especially to professor Cristina Boeres. Has been a major pleasure working with all you, even though for such a short time. Projects like this one put in perspective the importance of our academic work.

Last but not least I want to thanks each and every person who have encourage or support me during this time. I would be impossible to cite them all here, but I want to let them know that I haven't forgotten any of them.

Resumo

Visualização em tempo real de grandes massas de dados não estruturadas necessitam de grande poder computacional e banda de memória. Muitas soluções de algoritmos paralelos foram propostas para lidar com a complexidade computacional dos cálculos de intercessão celula-raio. Entretanto, a maioria não é capaz de prover taxas de renderização próprias para a interatividade, devido ao overhead gerado pela solução paralela. Este trabalho estuda a fundo os componentes do overhead de um algoritmo de renderização paralela, identificando os gargalos e sugerindo modificações no algoritmo a fim de se obter eficiência e escalabilidade, até mesmo quando imagens de grande resolução são utilizadas. Nosso algoritmo é baseado no algoritmo de raycast com paralelização dos dados. Utilizamos uma decomposição adaptativa da tela em porções chamadas tiles e estrategia de distribuição dos mesmos, um método paralelo para se encontrar o ponto de entrada dos raios na massa de dados e codificação da imagem para a gravação/envio das sub-imagens. O algoritmo alcançou ganhos significativos em termos de balanceamento de carga e significativa redução nos overheads da paralelização de imagens de grande resolução. Os resultados de speedup confirmam o potencial do algoritmo para renderizar eficientemente grandes massas de dado.

Abstract

Real-time visualization of large and unstructured volume datasets demands high computational power and memory bandwidth. Many parallel solutions have been proposed to deal with the computational complexity of the ray-cell intersection requirements. However, most of them are not capable of providing interactive frame rates for large datasets due to the overheads generated on the parallel solution. This work dissects the overhead components of a parallel rendering algorithm, identifying bottlenecks and suggesting modifications to the algorithm in order to achieve efficiency and scalability even when the images have high resolution. Our algorithm is built on the raycasting method with a data-parallel approach, it employs an adaptive decomposition of the screen into portions called tile and distribution strategy for those tiles, a parallel method for finding the rays entry points and an encoding method for subimages saving/transmitting. The resulting algorithm achieved significant gains in terms of load balancing, and significant reductions in the overheads of parallel rendering for big image resolution. The speedup results confirm the potential of the algorithm to efficiently rendering large-scale datasets.

Key-Words

1. Raycast
2. Parallel
3. Load Balance
4. Overheads
5. Irregular dataset
6. Irregular tile division
7. Volumetric rendering

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Scientific visualization is the process of graphically displaying real or simulated scientific data. It is useful to turn masses of numbers into pictures on the screen, and, therefore, is vital to many application areas such as biology, chemistry, computer science, geology, engineering, or medicine. For three dimensional data, there are various techniques, collectively known as volume rendering, for the direct visualization of the volumetric data. Volume Rendering comprises very powerful 3D visualization methods that convey the internal information of the 3D volume, providing semitransparent views of the spatial relationships of the structures. In contrast, other visualization techniques show only the surface of the volume, like raytracing, or a low definition composition of the isosurfaces from the data.

During the past decades, there has been remarkable advances in volumetric data acquisition. The evolution in scanners technology and numerical simulations enabled the production of large volumetric datasets in a broad range of domains. The visualization of such datasets is critical to analyze and comprehend the information contained inside the data and verify and validate the results of the simulations. Depending on the structure and type of data, different rendering methods can be applied to perform the visualizations.

The structure and type of data rely upon the source where volumetric data comes from. Equipment such as Computed Tomography scanners and Magnetic Resonance Imaging devices usually produce data with regularity in positions, generating a rectilinear or a regular grid. Numerical simulations, on the other hand, often produce data in arbitrary positions, generating a curvilinear or even a unstructured grid. Fully unstructured volume data is often converted into a grid of tetrahedra. Tetrahedral grids can model complex geometries and are powerful in defining arbitrarily-shaped elements. However, rendering this type of grid is particularly challenging for directing volume rendering algorithms since the irregular topology difficults the traversal of the data.

Direct volume rendering algorithms captures the overall data domain, considering the volume as medium in which light can be absorbed, scattered or emitted as it passes through the volume. It produces high quality images without losing the details inside the data, but the cost of all the computations needed to determine what happens to light as it passes through the volume turns direct volume rendering a computationally intensive problem. Furthermore, if the volume data is represented as an unstructured or tetrahedral grid, an additional difficulty is included: the computation of where the light intersects each tetrahedron.

The parallel processing approach has been used to speed up the rendering task for many years, in different ways. The first parallel volume rendering algorithms were proposed for expensive parallel machines like SGI Power Challenge, IBM SP2, or SGI Origin 2000 [25, 26]. More recently, parallel algorithms are being designed to run on highly parallel computing devices such as graphics processing units (GPUs) [11, 29, 31, 37] or on massively parallel architectures such as cluster of computers [32, 38, 43, 46, 47]. Regardless of the parallel architecture in use, one common way to exploit parallelism in volume rendering is to use a data-parallel approach, called image decomposition. In this approach, the screen space is divided into non overlapping regions, called tiles, which are assigned, in groups, to the processing elements. Since the tiles do not overlap, they can be computed in parallel. Tile-based rendering, however, is usually susceptible to high load imbalance during execution, due to the irregular nature of the datasets. Even if an equal number of tiles is assigned to each processing element, it is very likely that some tiles have different amount of work, and can take longer to be processed. When the tile distribution is static, it is very hard to achieve an optimal load distribution for any given frame. When the distribution is dynamic, it increases the algorithm complexity and may require communication among the processing elements. So, the load imbalance problem has great impact on the overall performance and is still a challenge to the implementation of a parallel rendering system.

The main goals of our work are analyze the overheads incurred by a specific parallel volumetric render algorithm with unstructured grid datasets mostly. Propose some techniques that would increase the performance of this algorithm in terms of speedup and load balance, even when high resolution images needs to be rendered.

The high computational requirements of direct volume rendering for unstructured grids has been tackled in different ways in the literature: (i) reducing the computational complexity of the rendering algorithm (e.g. [54, 10]); (ii) generating approximate results by statically simplifying the grid (e.g. [16, 7]); (iii) reducing the memory requirements of the rendering algorithm (e.g. [42, 41]); (iv) parallelizing the rendering algorithm (e.g. [39, 5]). We focus our attention on the latter approach.

Past research in parallelizing direct volume rendering algorithms has concentrated on algorithm redesign to better explore the highly parallel architecture such as graphics processing units (GPUs) [12, 29, 31, 37], or solving the specific inherent problem of the parallel solutions that is the partitioning problem, with the goals of maximizing load balance and minimizing communication [1, 34, 46, 47]. In this work, we contribute to parallel direct volume rendering by dissecting all the overhead components of the parallel algorithm, identifying bottlenecks in the rendering process and suggesting modifications to the algorithm in order to achieve efficiency and scalability. We perform a detailed evaluation of all the steps of the rendering process, including the partitioning/load balacing problem, the communication overhead, data locality and the use of memory hierarchy, and "face projection", and propose a novel parallel rendering algorithm based on this study. Our algorithm is built on the raycasting method, uses a data-parallel approach, called image decomposition, and employs a hybrid programming model that explores message passing and multithreading on a cluster of multicore processors.

Although the solutions we propose here to minimize the parallel overheads are specific to the algorithm and architecture, the lessons learned can possibly be extensively applied to other parallel direct volume rendering approaches, including others architectures.

The experimental results showed that the strategies incurred negligible overhead in the rendering computation and can provide significant performance gains when compared to a traditional data-parallel raycasting algorithm. It was achieved less than 11% of load imbalance and up to 45% of increase in the rendering performance. The remainder of the work is organized as the following. The Section 2 reviews the previous work in parallel rendering. Section 3 presents the raycasting paradigm and data structures used in this work. Section 4 describes the parallel rendering algorithm, the parallel structure and the parallel strategies used and proposed for each step of the rendering pipeline. Section 5 reports the experimental results and overheads analysis. Finally, Section 6 presents the conclusions and future research plans.

Chapter 2

Related Work

Several parallel direct volume rendering algorithms have been proposed throughout the years. They were classified by Molnar *et al.* [33], according to the division of the rendering task among the various rendering threads, as: sort-first, sort-middle and sort-last. In sort-first parallel renderering, the screen space is divided into tiles and each processor is assigned a set of tiles. This approach usually has smaller communication requirements, but they are very susceptible to load imbalance. Therefore, a number of works focused on the specific overhead caused by the load imbalance in sort-first approaches for unstructured grids.

Some works uses the work stealing paradigm for load balance. In work stealing paradigm a computational node without pixels to render, requests (or steals) pixels from its neighbors therefore this paradigm dynamically balance the load within a frame. Our work uses different paradigm for balancing the load. It uses a fixed tile distribution within a frame. Whitman [55] introduced work stealing in parallel rendering for shared-memory architecture, and Nieh and Levoy [39] for distributed-shared memory architectures. The work by Coelho *et al.* [8] and Farias *et al.* [13] proposed some work stealing algorithms for a distributed environment as a cluster of PCs. Balancing the load among the threads dynamically, however, requires either global information about the load of the rendering threads or incurs in communication overhead. Another path to increase load balancing, also used in this work, is to provide a good distribution of the rendering task before the actual computation begins. Muller [35, 36] describes different algorithms to recursively divide the screen according to estimated workloads. The work by Abraham *et al.* [1] resizes the tiles in order to promote the same amount of work for all tiles, Our work tries a similar approach of tile division. Kutluca *et al.* [20] presented a comparison of twelve adaptive IS decomposition algorithms. In a preliminary work [21] we proposed another screen partition algorithm that is adaptive and based on a quadtree division, this algorithm will be explained and deeply analyzed throughout this work.

In sort-last parallel rendering each rendering thread is responsible for rendering part of the scene. It has been widely used in different works. The works [27, 37, 56] focused on the load balancing overhead by dividing the volume into bricks, and reassigning bricks to less overloaded nodes. Aykanat *et al.* [4] proposed a graph partition scheme to the decomposition problem. Another important issue in sort-last algorithms is the final image compositing stage. This stage can potentially become a bottleneck, since it demands a large amount of message exchange. The works by Yu *et al.* [57] and Lee *et al.* [23] focused on reducing this overhead. Yu *et al.* introduced a new image compositing algorithm, called 2-3 swap. Lee *et al.* introduced a parallel pipeline method which avoids link contention. Childs *et al.* [6] focused on the scalability of the parallel solution and proposed a hybrid approach that parallelizes over both elements of the input data and over the pixels of the output image.

Another common way of speeding up volume rendering is by taking advantage of modern architectures, such as GPUs or Cell processor. In [52] Weiler *et al.* implemented a GPU-based raycasting algorithm that was further extended by Espinha and Celes [12]. Bernardon *et al.* [5] also proposed a GPU-based algorithm based on raycasting that renders non-convex irregular grids. Ruijters *et al.* [45] pointed out some of the bottlenecks of GPU-accelerated raycasting, but their work focuses on regular grids. Some attempts have been made to deal with the problem of the memory limitation of the GPU. Weiler *et al.* [53] and Fout and Ma [15] used data compression. Maximo *et al.* [31] implemented a new scheme for storing face data. Lately, there are some works on GPU clusters[18, 3, 28, 37]. The power of the Cell processor has been explored in [9] for raycasting of unstructered grids and in [19] for regular grids. The work by Smelyanskiy *et al.* [49] proposed a thread- and data-parallel implementation of ray-casting that explores the architectural trends of multi-core and GPUs, and an upcoming many-core processor. They tackled the communication overheads using compression and analyzed the cache behavior of their approches; They used, however, a sort-last approach for regular grids. The work by Marchesin *et al.* [28] also proposed a sort-last approach for regular grids, but that runs on multiple GPUs. They also analyzed the time breakup of their approach in order to identify the bottlenecks.The sort-middle scheme redistributes the middle result of the rendering pipeline. It is seldom implemented in software parallel renders, since its scalability is limited by the communication overhead generated. Our approach here is to focus on all overheads of a sort-first parallel raycasting algorithm for unstructured grids entirely implemented on software.

Chapter 3

Raycasting Algorithm Overview

Our parallel rendering algorithm is based on the raycasting paradigm proposed by Roth [44]. In the raycasting paradigm, a ray is cast from the viewpoint through each pixel of the image. As the ray moves forward in the data volume, it intersects a number of spatial structures called voxels in it. Every pair of intersections is used to compute the voxels contribution for the pixel color and opacity and this contribution is proportional to the path that a ray travels within a voxel. The ray stops when it reaches full opacity or when it leaves the volume. Figure 3.1 shows a 2D example of a single ray. As can be seen, in a) the ray enters the dataset through a visible external face. As the ray moves through the dataset, as seen in b), the color and opacity is calculated according to the path the ray travels inside the voxel. The next voxels are fetched and the process continues. For this example, the process ends as seen in c), when the rays leaves the dataset.

This work is based on the sequential raycasting algorithm ME-Ray proposed in [41]. The data structures and the rendering pipeline are presented next.

3.1 Data Struture

The volumetric data is composed of a cloud of points, each point with an scalar value α associated to it. The α value is result of simulation or measured by sensors and can represent any scalar field, for instance density inside a volume. This cloud of points are organized in structure called voxels. The algorithm assumes an unstructured volume dataset in the form of connected tetrahedral voxels. The faces of the tetrahedra are either shared between two adjacent neighbor tetrahedra (inner faces) or they belong to the boundary of the tetrahedral grid (external faces). For the algorithm to compute the ray traversal from one tetrahedron to the next, the following four data structure are employed:

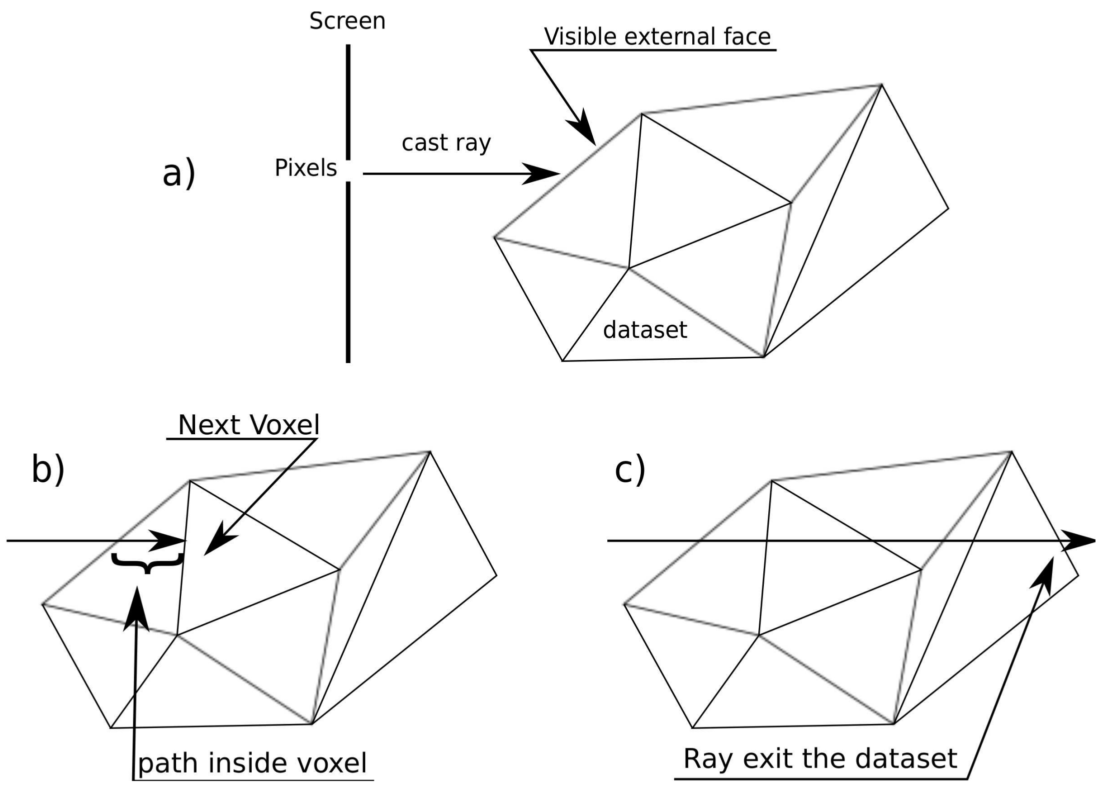

Figure 3.1: Example of the renderization process

array of points, *array of voxels*, *array of external faces* and *array of visible faces.* Each element in the *array of points* holds the X, Y and Z coordinates of the vertex, the scalar value α and an array with the index of the voxels that the vertex belongs to. The *array of voxels* stores the tetrahedra. The voxel structure contains the indices of the neighbor voxels and the indices of its vertices in the *array of points.* The *array of external faces* stores the indices of the voxels and its faces that are in boundary of the dataset. The *array of visible faces* is a sub group of the *array of external faces.* The *array of visible faces* has the indices of the voxels and its faces that are in the boundary of the dataset and are visible for a given point of view. With these structures, the algorithm can easily know the next face the ray will intersect, and consequently, what the voxels in the ray path are.

3.2 Preprocessing

In a preprocessing phase, the tetrahedral grid is read, and the *array of points* and the *array of voxels* are allocated in memory. After that, the array of external faces is also calculated. The preprocessing step is executed only once for each dataset, independently of the visualization angle chosen.

3.3 Pre-render

The pre-render phase is the first phase of the rendering pipeline, and has to be executed for each new visualization angle. It is divided into two distinct steps: *rotation* and *face projection*.

3.3.1 Rotation

The rendering from a certain point of view starts by executing the operations to rotate the data in the axis x, y, and z, by an angle degree of β, γ and δ respectively, according to the angle of visualization. Each vertex P is multiplied by the rotating matrix R_x, R_y and R_z given by (3.1), (3.2) and (3.3), respectively. The new coordinates of the point P_{new} is given by the equation (3.4). The amount of work performed in the *rotation* step is independent of the resolution of the rendered image.

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} \tag{3.1}$$

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix} \tag{3.2}$$

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.3}$$

$$P_{new} = P \cdot R_x(\beta) \cdot R_y(\gamma) \cdot R_z(\delta) \tag{3.4}$$

3.3.2 Face projection

After the data is rotated, the *array of external faces* is traversed to determine the faces that are visible in the chosen point of view. The visible faces are the external faces whose normals make angles greater than 90^o with the viewing direction. Figure 3.2 shows an example of a tetrahedron where one visible face and one invisible face are shown. The face with *Normal 1* is visible, because the normal makes an angle with the viewing direction

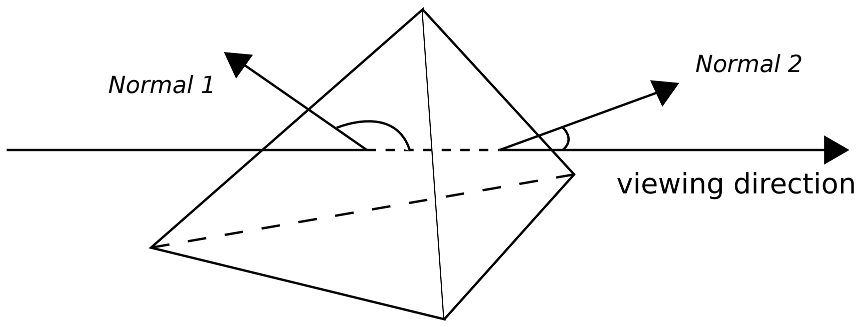

Figure 3.2: Example of the angles made between the viewing direction and the normal of a visible and invisible faces

greater than 90^o degrees. The Face with *Normal 2* is invisible, because the normal makes an angle with the viewing direction smaller than 90^o.

The visible faces are stored in the *array of visible faces.* Having computed all the visible faces, the algorithm projects them on the screen. To project the visible faces, the *array of visible faces* is ordered in such a way that the faces closer to the viewer are the first faces in the array, and the faces distant from the viewer are the last in the ordered *array of visible faces.* For each pixel, all the visible faces are checked for intersections, if the pixel falls into a certain visible face, this face is stored in a list of intersections belonging to this pixel. This list of intersections is used as the entry point for the rays.

3.4 Rendering

For each ray r that corresponds to a pixel s, the algorithm has to compute the next intersection of the ray by inspecting all other faces of the current voxel, or by inspecting the neighboring voxels. As the ray intersects the voxels, its entry point (e_{in}) and exit point (e_{next}) in each voxel are determined using a ray-plane intersection computation. Every time a new face is traversed, its coefficients are saved in a face buffer, and the lighting integral from e_{in} to e_{next} is computed, using an optical model. This computation calculates the contribution of the voxel in the color and opacity of pixel s.

When the exit face of a voxel is an external face, the ray leaves the dataset. If no more external faces are in the ray path, the color of the pixel has been computed. Otherwise, the ray re-enters the dataset in another voxel, and the process continues until the ray leaves the volume. Once the ray left the dataset it can re-enter the dataset if the dataset is not convex or has holes inside it. The process of traveling through the *array of voxels* calculating the color and opacity of the pixels is called *rendering process.*

It must be clear in the above explanation that the algorithm uses the orthographic projection view for the rendered image. The perspective projection can be achieved without significant changes in the algorithm. Instead of casting rays in a inclined direction, from the point of view through out each pixel of the image, the points can be transformed in such a way that the orthogonal projection will produce the same final image in perspective.

the physical illumination model we used is the one described in Max [30] where the semi-transparent substance in the volume absorbs and irradiates energy as the ray passes through.

Chapter 4

Parallel Algorithm

Parallelizing raycasting is relatively simple. Every cast ray can be traced through the volume independently from every other ray. Our parallel algorithm adopts the sort-first approach to divide the work among the processing elements. The algorithm divides the screen into tiles that are assigned to the parallel processing elements. A tile consists of a unique set of pixels that form a closed area with in the screen, each pixel of this tile will be traversed by a ray. The tile subdivision is not only useful to the assignment problem, but it is also important for improving cache performance, since nearby rays usually traverse a similar group of voxels of the volume.

Our parallel algorithm was designed to take advantage of recent heterogeneous architectures of multicore clusters, composed of shared memory computer nodes, connected by a messaging network. An example of the logical communication scheme of the algorithm for three computer nodes is illustrated in Figure 4.1. Each processor/core is called a rendering thread, and is responsible for rendering a set of tiles of the image an store those tiles in a local buffer. Each shared memory computer node is called a team of threads, and has a special rendering thread, called leader thread, that is responsible for creating the shared data structures for the team like the image buffer, dataset and list of visible faces. The whole system has one master rendering thread, that is responsible for work distribution and the final image construction. Remark that the leader thread is a rendering thread that also executes two additional tasks: structure allocation and message exchange. The master is a leader thread that executes also the tile distribution aiming at a good load balance among the rendering threads. For the sake of simplicity Figure 4.1 does not shown for the master thread a image buffer, but the master thread do have a image buffer, since it can also act as a rendering thread.

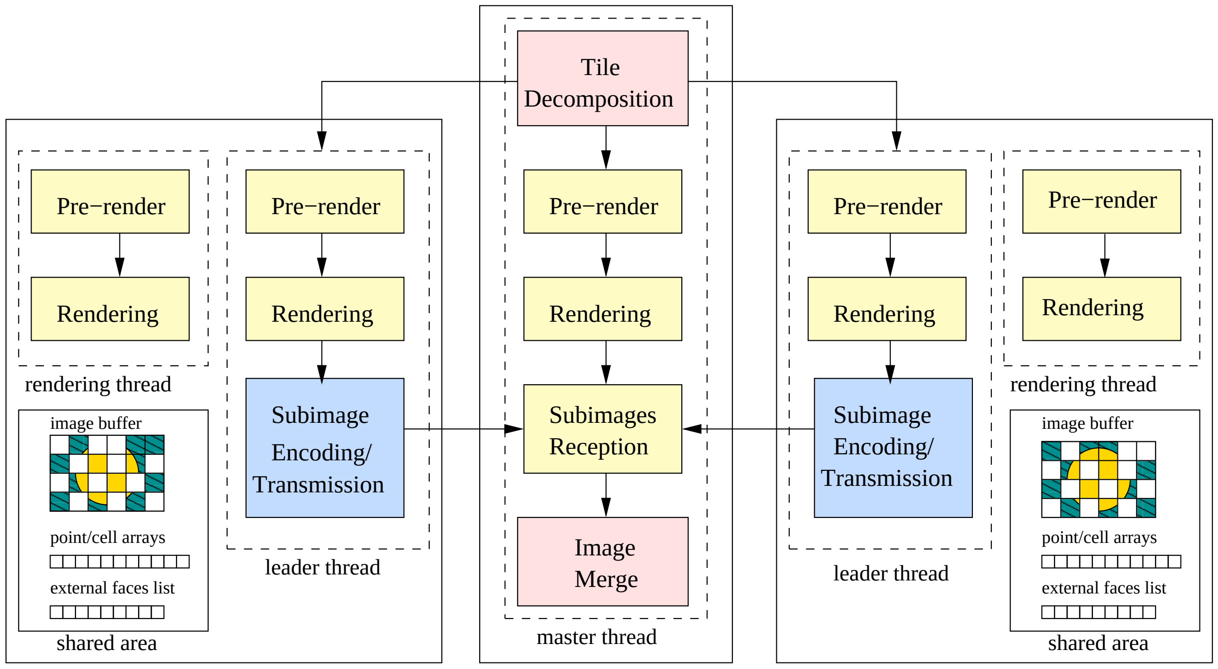

Figure 4.1: An example of our parallel algorithm.

The algorithm starts in a preprocessing step when all the leader threads read the entire dataset, and create the *array of points*, the *array of voxels* and the *array of external faces* in the shared memory.

The rendering of each point of view follows five steps: tile decomposition and distribution, pre-rendering, rendering, subimages sending and image merging. The master decomposes the screen into tiles and assigns the tiles to the rendering threads. After that, the master sends each leader thread the set of tiles to be computed by its team. In each team, each thread is responsible for the pre-rendering and rendering phases. After each team finishes the rendering of its tiles, the leader sends to the master the generated subimages. The master receives the subimages and merge them to form the final image.

4.1 Tile Decomposition and Distribution

The first step is to decompose the screen in tiles in order to divide the rendering work and, after that, the tiles have to be distributed among the rendering threads.

4.1.1 Decomposition

For a sort-first raycasting algorithm the tile division can be done by the master in two possible ways. The traditional one is a regular division where all tiles are squares with

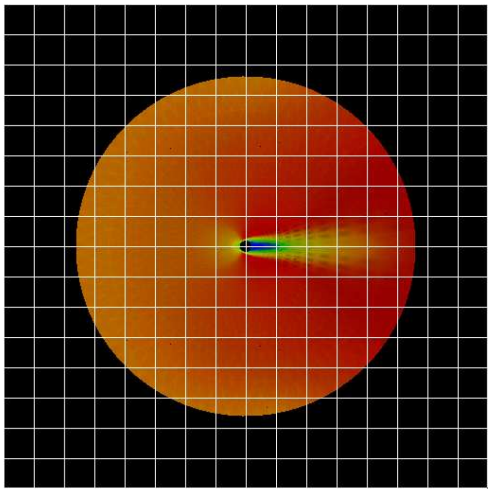

Figure 4.2: Example of a regular Division of tiles

the same size. This tile division is straightforward. The screen just has to be divided into the same amount of rows and columns, and the total number of tiles will be the product of the numbers of rows and columns. Figure 4.2 shows an example of a rendered dataset with this tile division. The image generated is divided into 16 columns and 16 rows with a total of 256 tiles.

The problem with this naive division is that it could generate tiles with very different computational costs. There are tiles that are more costly to render than others. As can be seen in Figure 4.2 the tiles at the corners of the image have lower computational cost than the ones at the center of the image.

Another option of division is an irregular tile division. An example of this division can be seen in Figure 4.3. The areas that require greater computing time to be rendered can be more divided, while areas that require less computing time can be less divided, generating tiles with nearly the same computing time requirements. By comparing Figures 4.3 and 4.2, it can be observed that the corners of Figure 4.3 has much less divisions, since there is no data to be rendered in those areas.

One problem is that, in traditional raycast algorithms, the computational cost of a pixel is not known until the pixels are actually rendered. Therefore the total cost of the tile is not known until the tiles are completely rendered.

In order to implement such technique, it is required to estimate the computational cost of the areas of the screen and to divide the tiles according to those estimated costs.

The tile decomposition scheme employed by the parallel algorithm is based on our work [21] that uses irregular tile division. The idea is to estimate the rendering cost of

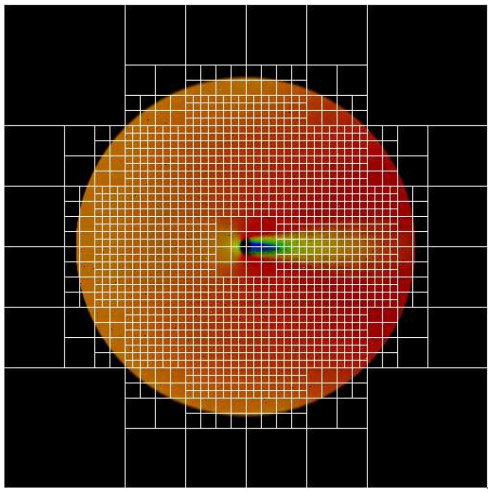

Figure 4.3: Example of a irregular division of tiles

each pixel and use this estimation to adaptively divide the screen into tiles. The pixel cost estimation exploits frame-to-frame coherence and use the total length of the path that the ray travels inside the volume (that reflects the final cost of the illumination integral), in the last frame generation, to estimate the cost of the pixel in the current frame. With the estimated cost of each pixel on the screen, the screen is adaptively divided until an even subdivision of tiles based on rendering loads is achieved. In other words, our goal is to have tiles with low standard deviation of costs among them.

This decomposition scheme uses a dynamic tile division that is called *Adaptive tile decomposition*. The main idea of the *adaptive tile decomposition* is to store all the tiles in a hierarchical structure of a quadtree that can be rearranged several times through the *quadtree rearrangement algorithm* until a critical value is found, guaranteeing that a good division has been achieved.

A quadtree is a structure that was initially proposed by Finkel [14] and has been broadly used in image processing, encoding and compression. Each leaf, that is a node without children nodes, in the quadtree corresponds to one tile in the screen. Each internal node has four children nodes. Each node stores the estimated computing cost of the tile it represents, in the case of a leaf node, or the total cost of its sub-quadtrees in the other cases. Figure 4.4 shows an example of the representation of a quadtree and the equivalent division of the screen. As shown in Figure 4.4, the tiles in the screen in a counterclockwise way, starting at the lower left corner correspond to the tiles stored in the quadtree from left to right.

For the first quadtree no cost can be estimated, since no frame has been rendered yet. Thus, the first quadtree is constructed initially as a full quadtree of a certain length. The

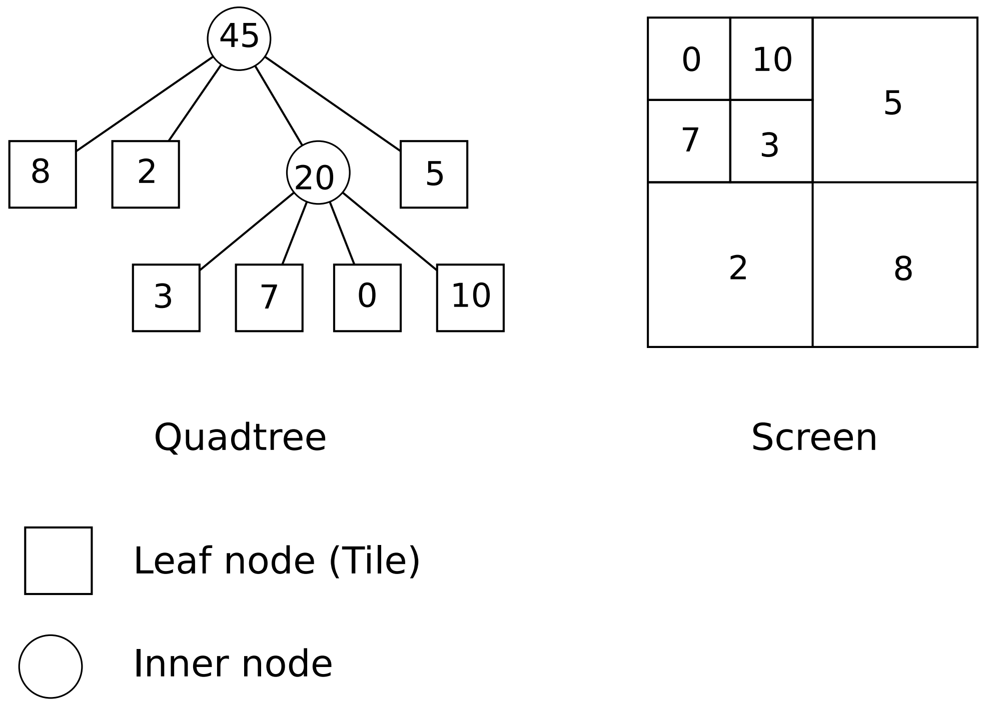

Figure 4.4: Example of a quadtree.

quadtree is guaranteed to always have at least one tile per rendering processor. This is done by evaluating if the number of leaf nodes in the quadtree is at least equal to the number of rendering processors. If there is not at least one tile per rendering thread, all leaves of the quadtree are split in four new nodes until there is at least a single tile for each processor. This first quadtree has a tile division similar to a traditional one. The tiles, for this first quadtree, are also distributed in a traditional way, as it is not possible to estimate the cost of the tiles. After that, for the rendering of the next frames, the quadtree of the past rendered frames is used and a rearrangement algorithm is employed to provide an adaptive division.

The *quadtree rearrangement algorithm* is a recursive algorithm that only ends when the Rule 1, defined next, is satisfied.

Rule 1 *Given a limit cost W, no internal node of the quadtree has a total cost lower than W and no leaf node has cost greater than W.*

The *quadtree rearrangement algorithm* performs two operations named *split* and *join* in the quadtree in order to guarantee that Rule 1 is satisfied. For a given limit cost W, for every node n_i of the quadtree with associated cost C_i the following two actions may be performed:

1. If n_i is a **leaf node** and $C_i > W$, then a *split* operation is applied. In the *split* operation the leaf node is split in four new tiles each with cost equal to $\frac{C_i}{4}$.

2. If n_i is an **internal node** and $C_i < W$, then a *join* operation is applied to this node. In the *join* operation, all the sub-quadtrees of this node are deallocated and the internal node becomes a leaf node.

If C_i is equal to W, no operation is executed in this node. The split and join operations are recursively applied to the quadtree until the Rule 1 is satisfied. At this point one execution of *quadtree rearrangement algorithm* finishes.

The *quadtree rearrangement algorithm* is executed several times, for different values of W. For each execution j, there is a limit cost W_j associated with it. Such cost W_j is decremented at each execution until a value $W_{critical}$ is found, meaning that a good tile division was achieved. This limit cost adjustment is performed by the *critical value adjustment* algorithm proposed by Aguilar [2], and it is also responsible for calculating the first W parameter (W_0).

The *critical value adjustment* uses the concept of information entropy or Shannon entropy [48] that was first used in data mining field by [51] and generalized by [2] to divide areas of equal load or weight in such a way that the standard deviation between all the areas were as small as possible and keeping the compromise of having as few areas as possible. This algorithm mathematically ensures that the division obtained is the best possible in terms of load balance and, at the same time, generates a small number of tiles, which will result in a low overhead due to the tile management.

Some parameters need to be acquired from the quadtree in order to calculate the *critical value adjustment*. From Rule 1, it is clear that the given limit cost W at the j^{th} execution, called W_j, represents a measure of work. Considering that the initial quadtree has L leaves, each one with a cost associated to it C_n, by Aguilar [2] a value suitable for W_0 (Initial value) is the average cost of all leaf nodes given by equation (4.1).For the first quadtree of the algorithm, the full quadtree at the first frame, all the C_n can have the same random value. The value used for the first frame in this work is $\forall n, C_n = 1$

$$W_0 = \frac{\sum_{n=1}^{L} C_n}{L} \tag{4.1}$$

Once the first W parameter (W_0) has been calculated, the *quadtree rearrangement*

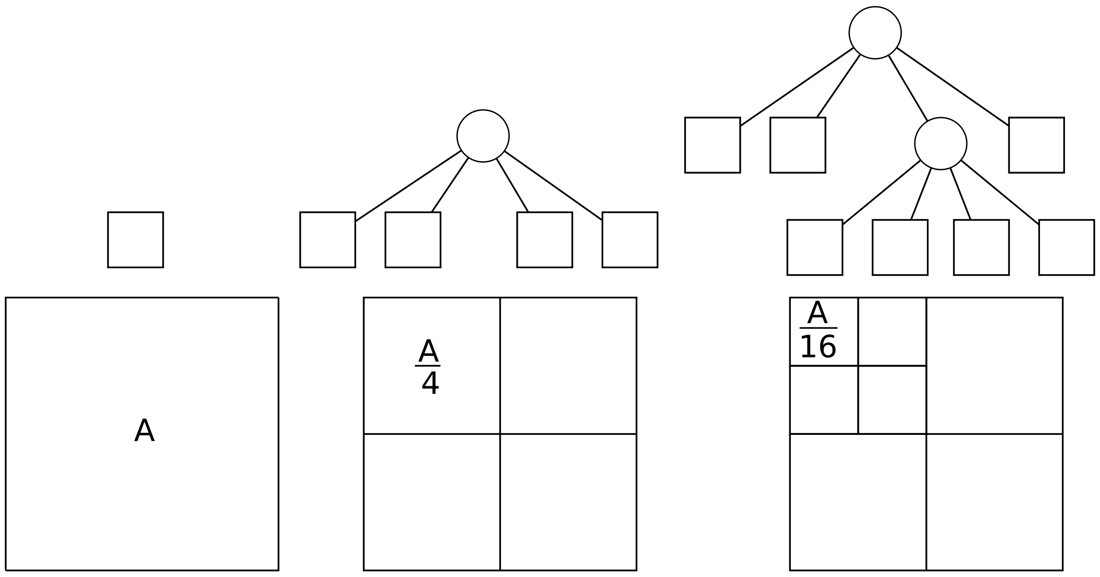

Figure 4.5: Every new level of tiles divide the minimum area of the tile by a factor of four.

algorithm can be applied, joining and splitting nodes when it is necessary. By Rule 1, no tile will have estimated cost greater than W_j, at the end of the j^{th} iteration. Here it is necessary to define the maximum density of work D_{max}, which represents the maximum work cost per area of tile that can exist in the given quadtree. The maximum density of work D_{max} can be given by equation (4.2), where A_{min} is the minimum area among all the tiles.

$$D_{max} = \frac{W_j}{A_{min}} \tag{4.2}$$

Knowing that the total area of the image is given by A_{image}, every time that one level is added to the quadtree the smallest possible tile area is divided by a factor of four as seen in Figure 4.5.

Thus, the minimum possible area of a tile is given by equation (4.3) where h is the height of the quadtree. Combining equations (4.2) and (4.3), the equation (4.4) is obtained as the maximum density of work.

$$A_{min} = \frac{A_{image}}{4^h} \tag{4.3}$$

$$D_{max} = \frac{W_j 4^h}{A_{image}} \tag{4.4}$$

The closer to the root a tile is in a quadtree, the greater is its area. On the other hand,

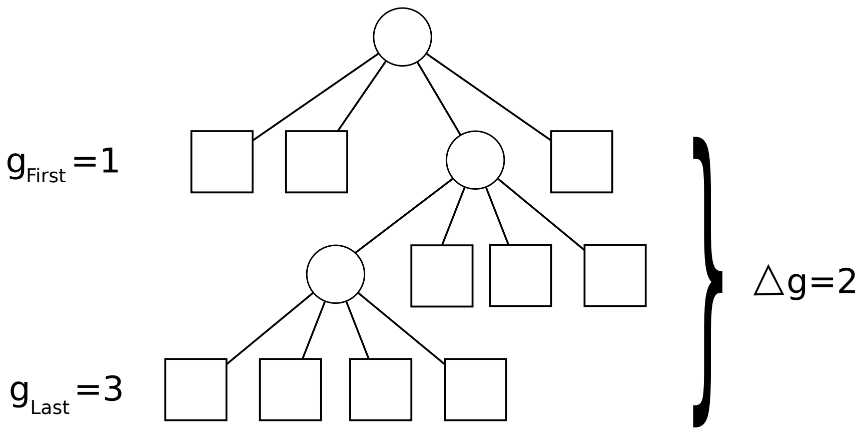

Figure 4.6: Example of the Δg calculation.

the deeper this tile is, the smaller is its area. Knowing that the leaf with smaller depth in a quadtree given by g_{First} and the leaf with greater depth is given by g_{Last}, it is assured that all other leaf nodes have their depths between g_{First} and g_{Last}. Because the depth of the leaves are discrete by definition, all possible depths are given by equation (4.5) and this also represents all possible values for area of the tiles, since the area of a given tile is directly proportional to its depth within the quadtree. In Aguilar [2], the Δg value is called generational difference, since it represents the difference of the first generation, or less divided regions of the dataset, and the last generation or most divided regions of the dataset. One example of the calculation of Δg for a quadtree can be seen in Figure 4.6. For this example the leaves with smaller depth are at level 1 and leaves with greater depth are at level 3. Consequently, the Δg value is 2.

$$\Delta g = g_{Last} - g_{First}$$
$$Number\ of\ possible\ areas = \Delta g + 1 \tag{4.5}$$

Each tile i within a quadtree can have cost W that varies from 0 (empty tile) to W_j which is the greatest possible value for the j^{th} iteration of the *quadtree rearrangement algorithm*. The amount of possible costs values for the quadtree is given by (4.6).

$$\Delta W = W_{max} - W_{min}$$
$$\Delta W = W_j - 0$$
$$Number\ of\ possible\ cost\ values = [0, W_j] = W_j + 1 \tag{4.6}$$

Considering that each tile can have $\Delta g + 1$ possible sizes for its area and each tile can have one of the $\Delta W + 1$ possible costs associated with it, the interval given by $(\Delta g + 1)(W_j + 1)$ represents the amount of possibilities for tile density. In an iteration j the density of work of a tile is inside the interval given by $[0, D_{max}]$. The width of those intervals are called *discrete density of work displacement* and is given by ΔD. Thus, to obtain ΔD, one must divide the maximum possible value of work density D_{max} by all possible intervals of density given by $(\Delta g + 1)(W_j + 1)$. The result is given by equation (4.7).

$$\begin{aligned} \Delta D &= \frac{D_{max}}{(\Delta g + 1)(W_j + 1)} \\ \Delta D &= \frac{W_j 4^h}{(\Delta g + 1)(W_j + 1)A_{image}} \end{aligned} \tag{4.7}$$

Must be point out that ΔD is an statistical entity defined by [2] and has no direct relation to the width of the cartesian product of the Δg and W_j real values. The follow example will illustrate this difference. Lets have a Δg equals 1 and W_j equals 2. Lets name G the set of all possible areas for the tile and C the set of all possible costs for the tile. Equation (4.8) shows those two sets, and the cartesian product between them. As we can see, we end up with only 4 possible values for the tile density with different widths between them.

$$\begin{aligned} G &= \{1, 2\} \\ C &= \{0, 1, 2\} \\ C \times G &= \{\{0,1\}, \{0,2\}, \{1,1\}, \{1,2\}, \{2,1\}, \{2,2\}\} \\ C \times G &= \{0, 1, 2, 4\} = 4 \end{aligned} \tag{4.8}$$

This value is not the one we are looking for. The value defined by [2] would be simply $(\Delta g + 1)(W_j + 1) = 6$, and the distance between each of those values is considered to be the same.

Figure 4.7 graphically shows the displacement values within the interval of density of work cost for a given quadtree. In this example, the total area of the image is given by A_{image}. This quadtree has 2 levels of tiles which leads to a $\Delta g = 1$. The cost W_j is equal to 6, so ΔW is also equal to 6 by equation (4.6). This situation will lead to 14 possibilities for tile density that are represented by the line below the quadtree. The displacement between those possibilities of tile density is the *discrete density of work displacement* that

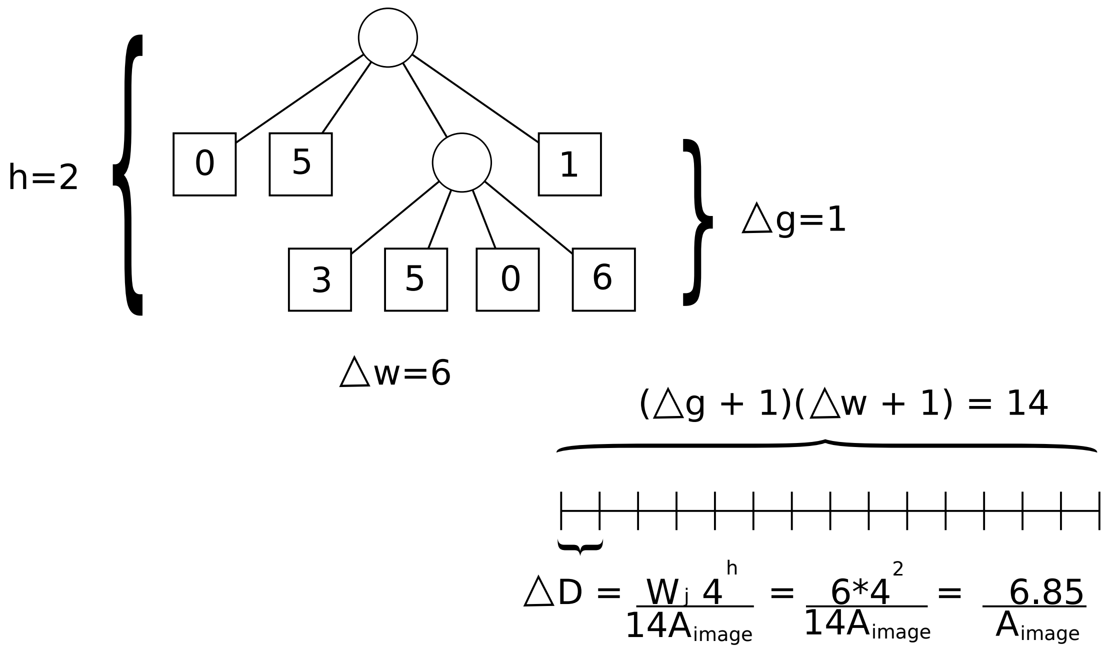
h=2
0
5
1
3
5
0
6
△g=1
△w=6
(△g + 1)(△w + 1) = 14
△D = Wj 4^h / 14A_image = 6*4^2 / 14A_image = 6.85 / A_image

Figure 4.7: Example of the critical value adjustment.

is pointed out as been $\Delta D = \frac{6.85}{A_{image}}$.

Aguilar has mathematically shown in [2] that when the value of *discrete density of work displacement* changes its order of magnitude from one iteration $j-1$ to the next j the *critical value* W_j is achieved and the algorithms terminate.

In this work, the change in the order of magnitude of the *discrete density of work displacement* ΔD_j is found when the value of ΔD_{j-1} in the $j-1$ iteration, is p times greater than ΔD_j in the current iteration j. The critical value is found if the inequality given by (4.9) holds true in a given iteration of the *critical value adjustment* algorithm. As can be seen in the inequality 4.9 the comparison between the ΔD_{j-1} and ΔD_j is independent of total area of the image A_{image} since this term is canceled in both sides of the inequality. It is not necessary to measure, by any means, the value of the total area of the image in order to apply the *critical value adjustment algorithm.*

$$\begin{gathered}
\Delta D_j \geq p \Delta D_{j-1} \\
\frac{W_j 4^{h_j}}{(\Delta g_j + 1)(W_j + 1) A_{image}} \geq p \frac{W_{j-1} 4^{h_{j-1}}}{(\Delta g_{j-1} + 1)(W_{j-1} + 1) A_{image}} \\
\frac{W_j 4^{h_j}}{(\Delta g_j + 1)(W_j + 1)} \geq p \frac{W_{j-1} 4^{h_{j-1}}}{(\Delta g_{j-1} + 1)(W_{j-1} + 1)}
\end{gathered} \tag{4.9}$$

The value p is chosen in a preprocessing step. A study of the determination of p will be discussed in Chapter 5 Section 5.3.

In summary, the idea of the *critical value adjustment* algorithm to find the best tile division is:

```
$j = 0$ //First iteration
$W_j = Avg(leaves)$ //Equation (4.1)
**Repeat until** *critical value* found //equation (4.9)
    *quadtree rearrangement algorithm*$(W_j)$; //rearrange quadtree
    **if** *critical value* for $W_j$ is found **break**
    **else**
        $W_j = W_j - 1$;
        $j = j + 1$;
**end repeat**
```

The list below summarizes the parameters that need to be acquired from the quadtree in order to apply the *critical value adjustment* algorithm.

- D_{max} - Maximum density work.
- Δg - Range for all possible tile depths in the quadtree.
- ΔW - Range for all possible costs in the quadtree.
- ΔD - Width of the discrete density of work displacement.

4.1.2 Distribution

After the screen has been adaptively divided into tiles, the tile assignment is done by using a 2-optimal algorithm, called Makespan Reduction heuristic, proposed by R.L. Graham in [17] for processor scheduling. In this heuristic, at first a list of tiles L is created in decreasing order of their costs, and one tile in the list is assigned to each rendering thread within a list R. After that, the list R of rendering threads are ordered by their loads in an increasing order. Then, each tile in L is assigned to the rendering thread with the lowest load, the rendering threads are continually reordered by their increasing loads, and the assignments go on, until there are no more tiles in the list L. The lower the standard deviation of the tile costs is, the closer the heuristic gets to the optimum solution.

The algorithm for this rearrangement is:

```
Create $L$ and $R$
Sort decreasing $L$ //$L$ sorted in a deceasing order of load
For each rendering thread in $R$ do
    Take out the fist tile in $L$ and assign to a thread in $R$
Repeat until $L$ is empty
    Sort increasing $R$ //$R$ sorted in an increasing order of total load.
    Give the first tile in $L$ to the first rendering thread in the list $R$
end repeat
```

Despite the fact that a 2-optimal heuristic can give good results, the overall result may still be poor if the tile division is not a good one. The granularity of the tiles will have great influence in the overall efficiency of the algorithm. Course grain tiles will produce high load unbalance; On the other hand finer grain division will lead to high overhead to manage and distribute all those tiles. Figure 4.8 shows an example of Makespan reduction heuristics working upon a divided image. The numbers in the tiles represent the estimated cost of each tile. In step 1, the list L is created. The tiles in the list L are ordered in decreasing order of its cost. In step 2, the list R is created. There are two elements in the list R named $R1$ and $R2$. For each phase of step 2, the list R is ordered according to its total tile load and one tile from L is assigned to the first element in R. In step 3 all the tiles in L have been assigned to a rendering thread in R. As it is seen, Makespan reduction heuristic gives the distributions of total costs 17 for $R1$ and 13 for $R2$.

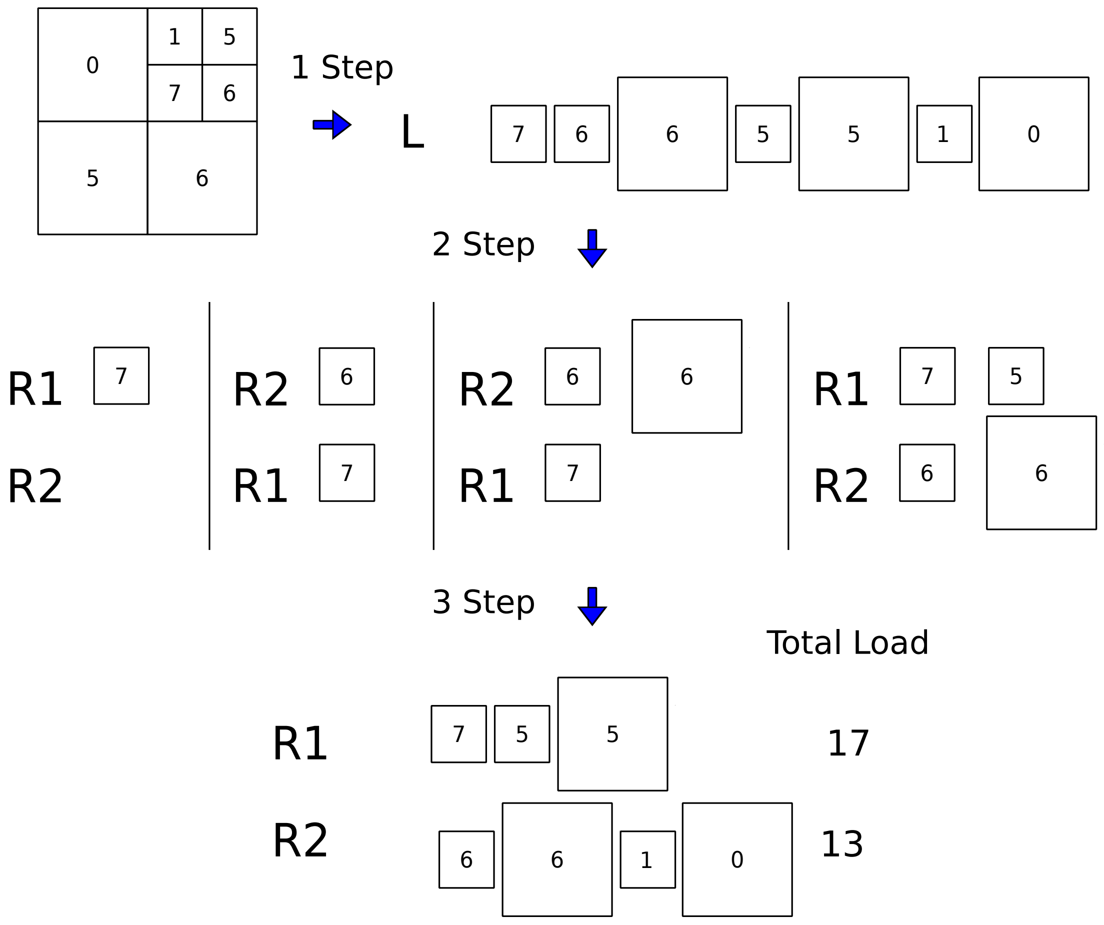

Figure 4.8: Example of Makespan heuristic with 7 tiles and two elements in the R list.

4.2 Prerendering

Once the rendering work is distributed to all threads of the system, the prerendering process takes place. At first, the data is rotated according to the point of view. This is accomplished by performing the rotation of the dataset, as explained in Section 3.3, in each shared memory computer node. In this case, the vertices to be rotated are evenly divided among the rendering threads, so that each thread applies the rotation matrix in its subset of vertices.

After that, the threads compute the list of visible faces, and start the visible face projection process. Parallel raycasting algorithms do not parallelize this process, since the amount of time spent in face projecting is usually much smaller than the amount of time spent in the rendering process. However, as the image resolution increases, the face projection phase cannot be underestimate, otherwise it will limit the speedup increase. So, here we propose two schemes for parallelizing this phase called *Faces-per-tile* and *Faces-per-quadtree*.

4.2.1 Faces-per-tile

In this technique it is assumed that the master thread sends each rendering thread a list of tiles computed in the *tile distribution* step. In this parallel face projection scheme, each rendering thread projects only the visible faces that are within the tiles that were assigned to it. The thread traverses the list of visible faces, and checks for each face if it belongs to one of its assigned tiles. The faces that do not belong to the tiles are ignored. For the faces that belong to a tile, the thread projects only the pixels inside the tile. This is done to avoid double projection of pixels when the face is within more than one tile.

This scheme can generate load imbalance, since the tile division takes the cost of rendering the pixels into account, but not the number of pixels inside each tile. So, some threads can be assigned with more pixels than others, generating a high face projection cost. Another important issue in using this parallel scheme is that, for images with small resolutions, and consequently a small number of pixels, the overhead of checking if the visible faces are within the tiles can outpace the parallelization gains.

Consider that the number of tiles is n, and the number of external visible faces is m. For each face to be projected, all the tiles in the list of tiles have to be tested for intersections, this will lead to a complexity for this search of $O(mn)$ asymptotically. This complexity will be used as a comparison parameter with the next technique.

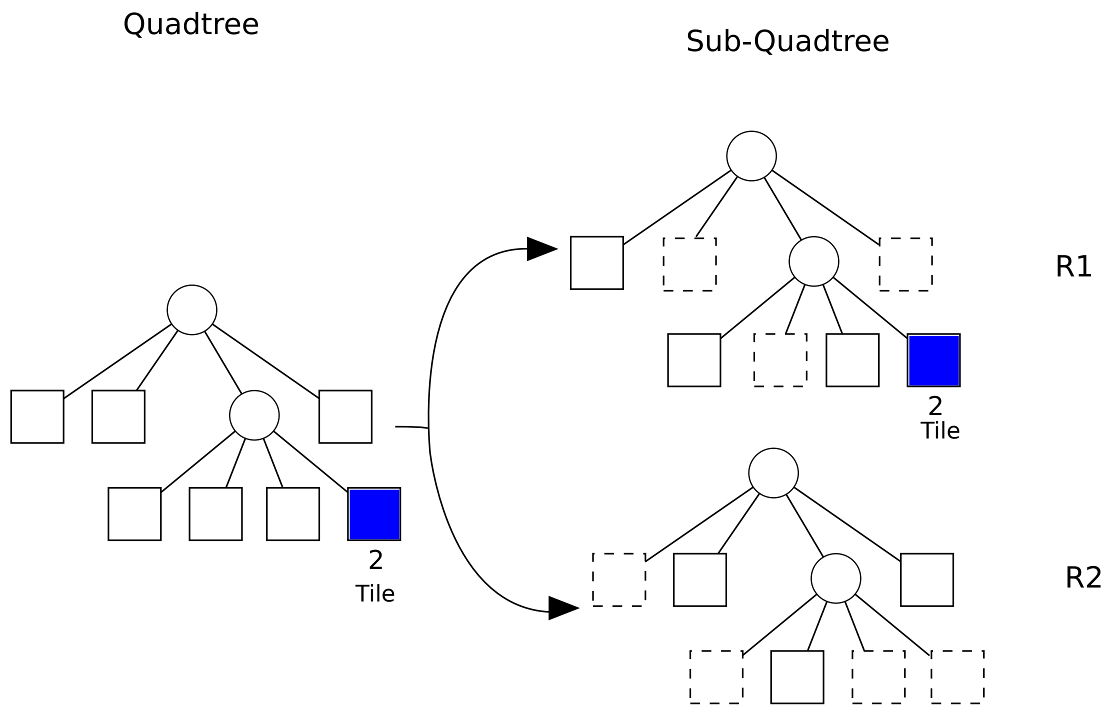

Figure 4.9: Example of the distribution of sub-quadtree

4.2.2 Faces-per-quadtree

In this technique, it is assumed that the master thread sends each rendering thread a sub-quadtree with the tiles calculated in the *tile distribution* step. The *Faces-per-quadtree* projection schemes, addresses the overhead of checking which visible faces fall under the assigned tiles. The idea is to reduce the number of checks by taking advantage of the quadtree structure created by the tile decomposition phase. Each internal node in the quadtree represents a quadrant of the screen. When we check if a certain visible face falls under an internal node of the quadtree, we are checking whether its pixels fall under the quadrant represented by that nodes.

To accomplish this scheme, the master thread no longer sends the list of tiles to the rendering threads, but the sub-quadtree that holds all the rendering threads tiles. Since the quadtree is usually small, this transfer can be negligible in the overall face projection overhead. Figure 4.9 shows the master sending the sub-quadtree to two different rendering threads. The dashed leaves of the sub-quadtree are tiles that were assigned to other rendering threads. Remark that all the leaves of original quadtree have to be assigned to one, and only one rendering thread. In the example the tile 2 of the master thread quadtree was assigned to the rendering thread $R1$.

So, each rendering thread stores locally only the portion of the quadtree that represented the tiles that were assigned to it. For each visible face, the rendering thread checks whether the face falls under the root node of its local quadtree. If it does, the same test is done with its children nodes, and so on, until a leaf node is reached; In this case the face is projected in the same way as *face-per-tile*, in which only the pixels inside the leaf node is projected. If the face does not fall under a quadtree node, no further checks are made to the descendants of that node. In this way, not all the visible faces are compared to all tiles assigned to this rendering thread. A great number of visible faces are compared only with the local quadtree root node, and soon discarded.

Considering that the number of tiles is n, and the number of external visible faces is m. This hierarchical structure increases performance because one element of the quadtree can be find on $O(log_4(n))$ as shown by Finkel [14]. Therefore the search for all external visible faces can be done in $O(mlog_4(n))$ asymptotically, which is lower than the $O(mn)$ complexity of the *face-per-quadtree* scheme. It is expected that *face-per-quadtree* might lead to a better result in cases of higher resolution than *face-per-tile*.

This technique still has the same load imbalance problems due to face distribution that *face-per-tile* has. This work does not address any technique to avoid or reduce this load imbalance, since this is not the most expensive computation in the whole visualization process.

4.3 Parallel Rendering

After each rendering thread has the list of pixels to be computed and the entry point for each pixel computed in the face projection phase, the raycasting algorithm starts. The algorithm used is exactly the same as the algorithm described in Section 3.4, where for each ray, the intersections are found and the illumination integral computed.

4.4 Subimages Transmission

After rendered, each pixel has its final color stored in a three byte structure called C_{RGB}. The three bytes of C_{RGB} stores the Red, Green and Blue values of the pixels, also known as the RGB color of the pixels. The C_{RGB} of all rendered pixels are stored in an one dimensional array called *raster*. The *raster* is a structure that can be easily mapped onto a two dimensional screen. Initially all the RGB values of all C_{RGB} are set to zero indicating a non rendered pixel.

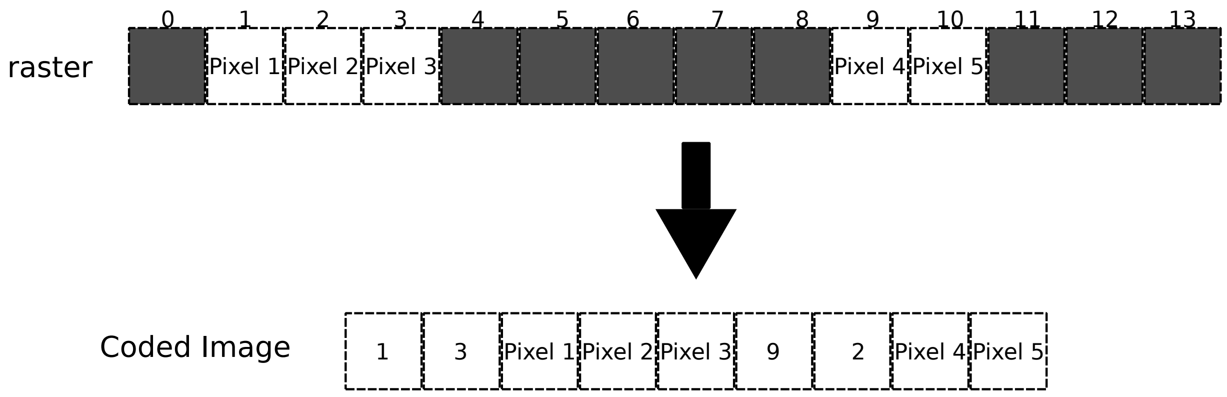

Figure 4.10: Example of image encondig

Each rendering thread has its own *raster* array, that has the same size as the final image. At the end of the rendering step each rendering thread has a unique set of rendered pixels in its own *raster* array, those rendered pixels are within the tile areas that were originally assigned to this rendering thread at the *tile distribution* phase. The rendering thread has to send its subimage back to master thread, but sending the whole *raster* array would be costly.

So, in order to reduce the size of the sending structure, we proposed an image encoding scheme. All the subimages generated by a rendering thread are sent in a single message, that is composed by an array of pixels structure. This array contains a number of continuous pixels structure, called *Cp*. Each *Cp* is a variable structure that has two control integers followed by a number of C_{RGB}. The first control integer points to a position in *raster* array where there is a continuous number of pixels that are not black. The second control integer contains the number of C_{RGB}, and each C_{RGB} represents a RGB color of a rendered pixel. When the subimage has one or more black pixels, a new *Cp* is added to the array. Since it is more likely to have groups of colored pixels together than many interleaved colored and black pixels, it is expected that the array of *Cp* is almost the same size as an array of pixels. Figure 4.10 shows an example of a *raster* array mapped to a message with two *Cp*s. Remark that the encoded image is smaller than the original *raster* array, and there are only four control integers of overhead more than the a single list of C_{RGB} would have.

Although the subimages send phase comprises only the transmission of the*raster* array from each rendering thread to the master. This phase timing is highly susceptible to the load imbalance intrinsic to the work division. Here we use a formula proposed by [25] as the measure for load imbalance.

$$LoadImbalance = 1 - \frac{Avg}{Max} \tag{4.10}$$

Where Avg is the average timing among all rendering threads and Max is the maximum timing of all rendering threads.

4.5 Image Merging

As soon as the Master receives a message from a team of thread it decodes the subimages received and save them in the final *Raster*. The assembly of the final image is a straightforward phase, since it involves only the transfer of each Cp included in the message to the correct position in the image.

After that, if there are more frames to be rendered, all the steps of the rendering pipeline, with exception of the preprossesing phase, are repeated, otherwise the algorithm finishes.

Chapter 5

Experimental Results

5.1 Methodology

Our experimental environment consists of a cluster with SMP nodes called *Netuno*. The cluster is located on the Rio de Janeiro Federal university. This cluster was top 138 in June of 2008 in the top 500 [50]. Each node of this cluster is composed of IBM blade boards. where each node consists of 2 Intel Xeon 2.66 GHz quad-core processors that share a 16GB RAM. The nodes are connected via Gigabit Ethernet network. All the 2048 cores run Linux CentOS 4.2.3. The parallel algorithm was developed in C, using MPI for the communication between the leaders and the master, and pthreads for the parallelization and communication of the rendering nodes inside a team. Table 5.1 summarizes some of the characteristics of this cluster.

Operational System	CentOS
Kernel Version	2.6.18-53-el5
CPU	Intel Xeon E5430 (Duo-Quad core) 2.66GHz
Cache Size	6144KB
Memory RAM	16GB
MPI Version	Intel 3.2.0.011
C++ Compiler	Intel 11.0.074

Table 5.1: Cluster *Netuno*

We used five well-known representative tetrahedral datasets: in Figure 5.1 we can see SPX from Lawrence Livermore National Lab, Liquid Oxygen Post shown in Figure 5.2, in Figure 5.4 and Figure 5.3 we see Delta Wing and Fighter from NASA respectively, and finally in Figure 5.5 we see Torso from University of Utah. SPX is a very irregular dataset that contains a hole in the grid, bringing extra difficulties to the renderer. Fighter is based on an aircraft plane that has thin and thick regions. Delta Wing, Liquid Oxygen Post, and Torso are tetrahedralized versions of regular datasets. Liquid Oxygen Post, in particular, is a thin cylinder that presents different rendering complexity according to viewing direction. Table 5.2 shows the number of vertices, tetrahedra voxels and external faces for each dataset.

Dataset	**# Vertices**	**# Tetrahedra voxels**	**# Number of External Faces**
SPX	149 K	827 K	44160
Post	109 K	513 K	27676
Delta	211 K	1.0 M	41468
Fighter	256 K	1.4 M	83504
Torso	168 K	1.0 M	6118

Table 5.2: Dataset sizes.

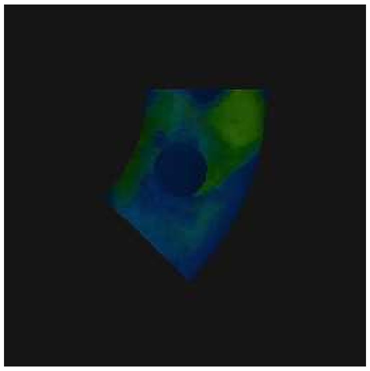
Figure 5.1: SPX dataset.

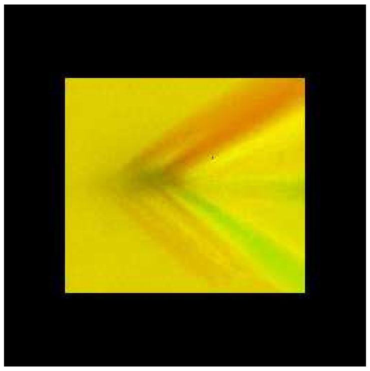
Figure 5.3: Fighter dataset.

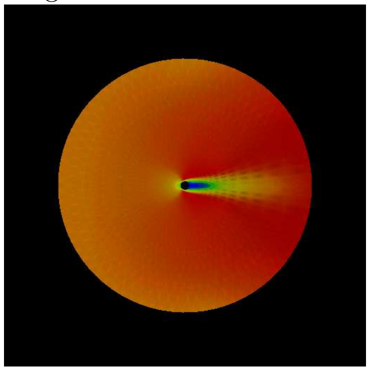
Figure 5.2: Post dataset.

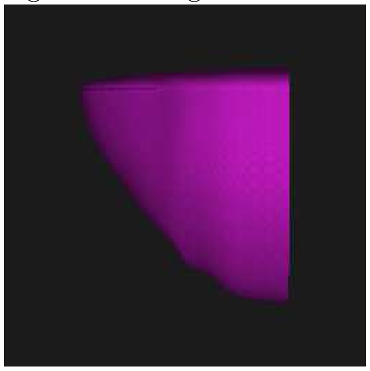
Figure 5.4: Delta dataset.

The resolution chosen to render the data set is also an important factor to be taken into consideration. The resolution measured in this work is always in terms of pixels that the image has. We can imagine the image as a matrix of pixels. Even though the number of columns and rows in this matrix can be different in size, in this work they are always the same size. For the sake of simplicity an image that has 1024 columns and 1024 rows in the pixel matrix is called an image with a 1G pixel resolution. An image with 2048

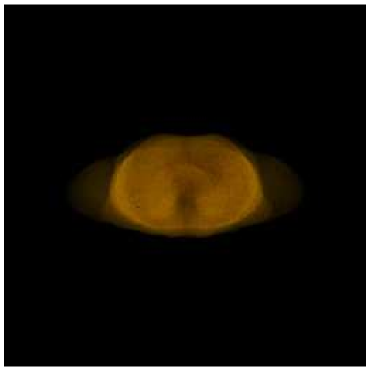

Figure 5.5: Torso dataset.

columns and 2048 rows is an image with 2G pixel resolution and so on. We used a 4G pixel resolution image and each dataset was rendered from different points of view. For all these datasets, an animation path was defined. We considered that the point of view starts at 0^o, and was constantly rotated using a fixed stride of 2 degree angle.

The parallel raycasting algorithm proposed is called PRay. This algorithm uses the techniques *tile decomposition and distribution* for load balancing and the *face-per-quadtree* for face projection. This version is compared with the traditional parallel raycast. The traditional parallel raycast divides the screen equally into 32×32 tiles and distributes the tiles randomly across the rendering threads. This tile division was determined in a previous experiment where different tile divisions were evaluated to find the best one for all datasets.

5.2 Parallel Rendering Overheads

Before start our analyzes of the overhead, we present in Table 5.3 the total application time of the sequential algorithm and the PRay algorithm. Those data were acquired with 4G pixels images and 64 cores for the parallel algorithm PRay, the total time is presents in milliseconds.

Datasets	Sequential(ms)	PRay(ms)
SPX	176235.03	4211.99
Delta	196235.08	4524.09
Fighter	153251.76	4412.00
Post	328853.07	6381.46
Torso	188054.98	4489.75

Table 5.3: Total application time for 4G pixel resolution.

As we can see, great improvement has been achieved. Next we will present the and analyzes the the reason for such improvement starting with the overhear.

In Figure 5.6 we show the timing breakdown of our parallel algorithm, PRay, for each dataset, when running on 64 rendering threads. The breakdown was divided into four phases: Tile Decomposition and Distribution, Pre-render, Subimages transmission, and Image Merging. This overhead was measured exclusively in the master thread were all the load balance policies are applied and the final image is merged. Time measuring functions were started before the execution of each part of the overhead breakdown and stopped at the end of those parts instrumenting this way the whole execution of the master thread.

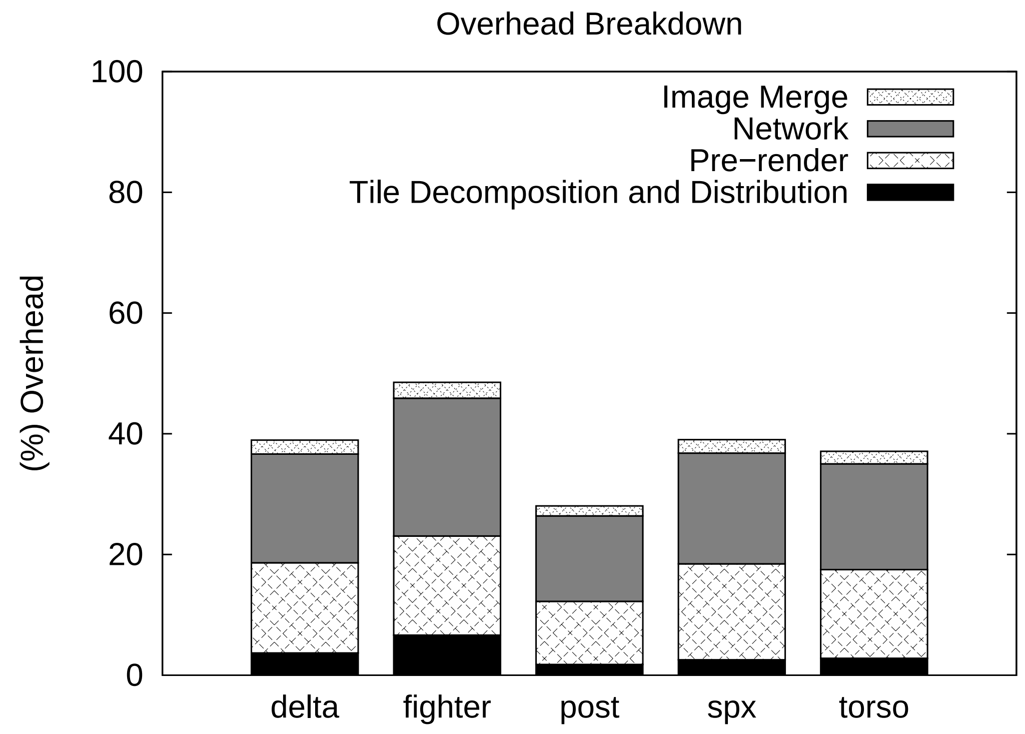

Figure 5.6: Timing breakdown of our parallel algorithm.

As we can observe in Figure 5.6, the **Pre-render** and **Subimages Send** components dominate the total overhead. The total overhead remains around 40% of the total execution time. The smallest dataset, Post, was the one that generated the smallest overhead, and as the dataset size increases, the total overhead also increases, mainly due to the increase in the **Pre-render** component of the overhead. This component is related to the number of external faces of a dataset, and this component is higher in the fighter dataset, which is the one with most external faces. For experiments with larger image resolutions, we obtained increasing in the total overhead. Following, we will analyze each of the overhead components and the rendering behavior in detail.

5.3 Tile Decomposition and Distribution

Our study, at this point, has to investigate how well our tile decomposition strategy is in dividing the screen into tiles. Before we analyze the tile decomposition results, we have done some experiments in order to define the best value for the p parameter of the *adaptive tile decomposition*. This value defines when ΔD changes by an order of magnitude. We varied p from 1.1 to 4.0 and obtained similar results for all datasets. Figure 5.7 shows an example of these results for SPX. Increasing the value of p, shown in the x axis, can produce a significant reduction in the total execution time presented in the y axis, until p reaches a value near 3.0. After that, further increases in p would have great impact in the execution time. So, we decided to set p = 2.9 for the next experiments. This experiment uses 64 rendering threads, but the usage of more or less threads should make no difference in the final result (finding the best value of p), since this results depends only in the dataset geometry.

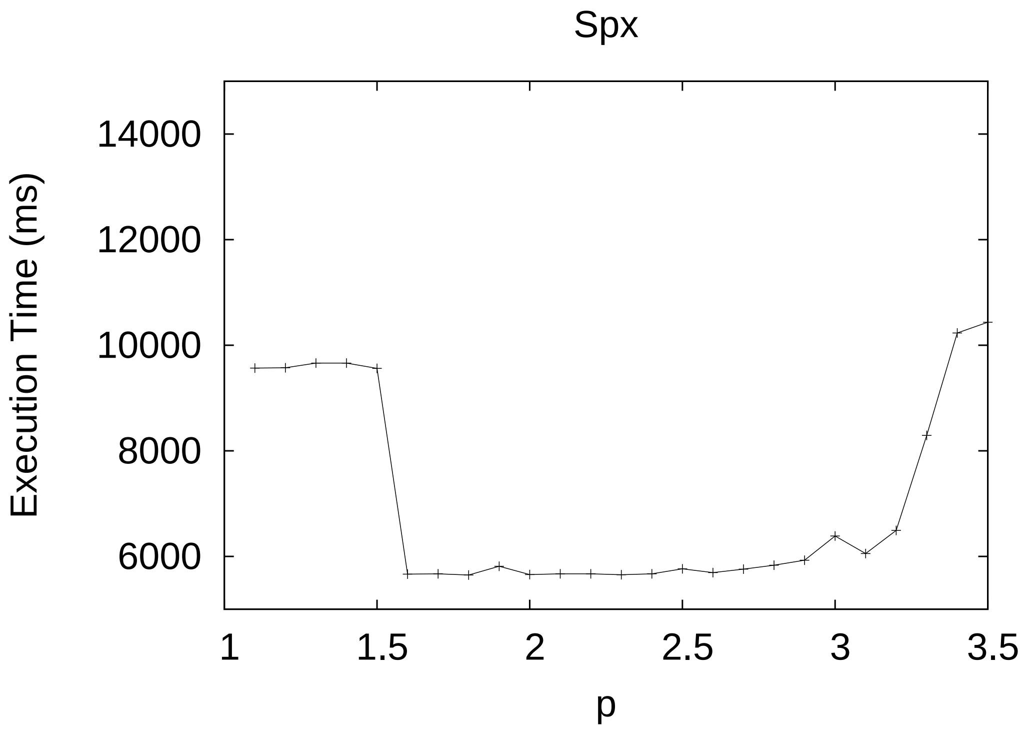

Figure 5.7: Variation of p parameter for SPX dataset.

Having calibrated the p parameter, our second study investigates the accuracy of our load estimation strategy. After the rendering of each point of view, we compute the difference between the estimated cost of each pixel and its actual cost. We used SPX as an example for this analysis, since SPX is more irregular than the other datasets. Figures 5.8

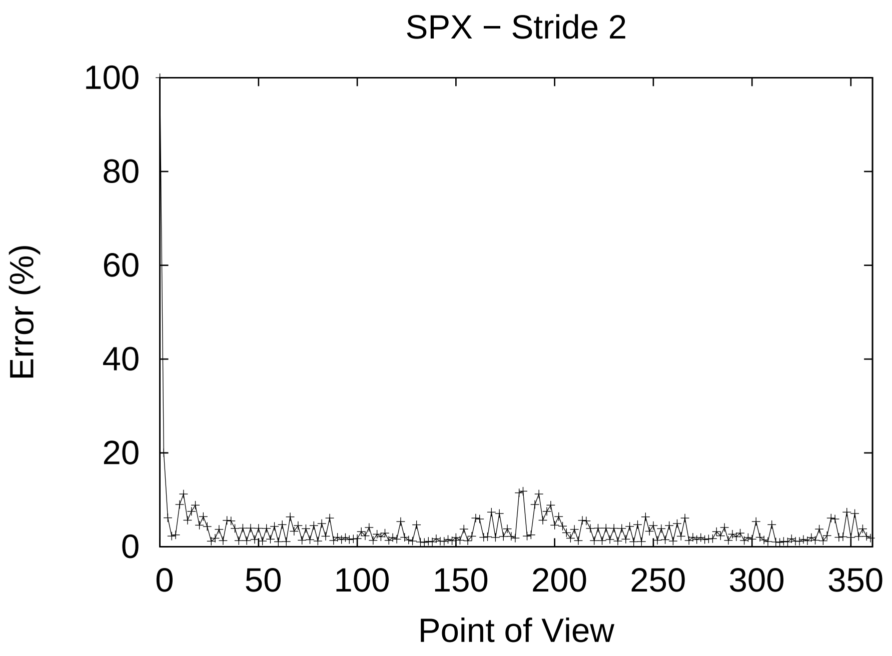

Figure 5.8: Accuracy of load estimation for SPX with a stride of 2^o.

to 5.10 show the average percentage difference between estimated and actual cost for all pixels in the screen for SPX, when the point of view varies from 0^o to 360^o. Figure 5.8 shows this difference when the point of view varies by 2^o. Figure 5.9 shows for a 10^o variation and Figure 5.10 for a 40^o variation. As we can observe in these figures, for a small stride, our estimated cost is very close to the actual cost, the difference between the estimated and actual cost stays near 5% and remains almost constant for all points of view. For a 10^o of stride, the difference stays near 10%. For a 40^o of stride, the estimate is in the average 50% different than the actual cost. Big angle strides, however, are not the usual requests of the users. Usually, the user does not desire an abrupt change in the angle of view of the data. Based on these results, on the other experiments, we rotated the data for 20 different point of views, using the stride of 2^o.

In order to evaluate the screen division generated by our tile decomposition strategy, we measured the standard deviation of cost of the tiles, and compared to the standard deviation generated by a fixed 32×32 tile subdivision. We normalized the standard deviation of the two strategies in order to compare them. Table 5.4 shows the comparison of the standard deviation of our strategy with the fixed division. The numbers show the percentage of the standard deviation generated by our strategy, considering that the standard deviation of the fixed division is 100%. As we can observe in this table, our decomposition provides a standard deviation always smaller than the fixed division. For smaller strides, our decomposition generates tiles with a standard deviation that is less

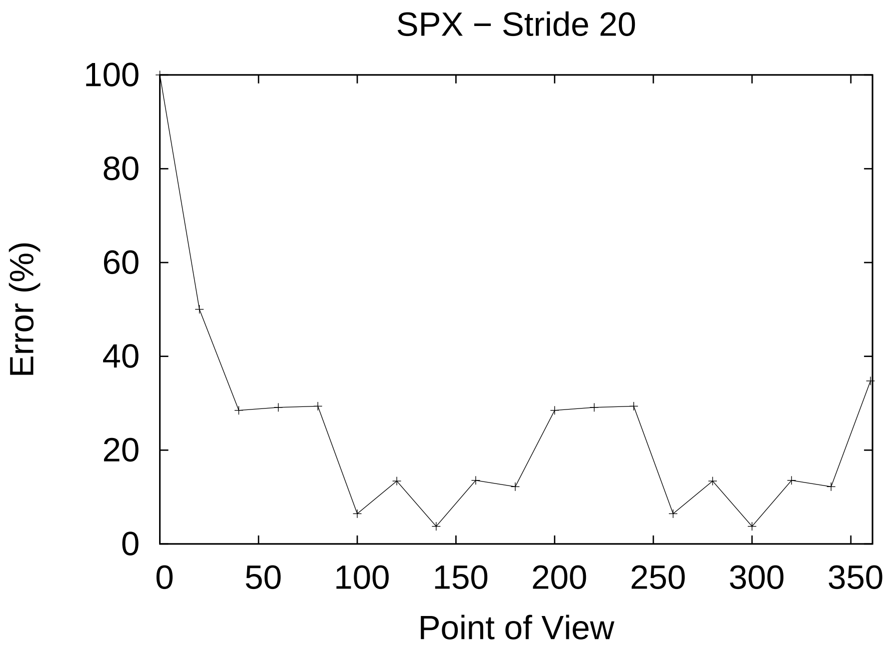

Figure 5.9: Accuracy of load estimation for SPX with a stride of 10^o.

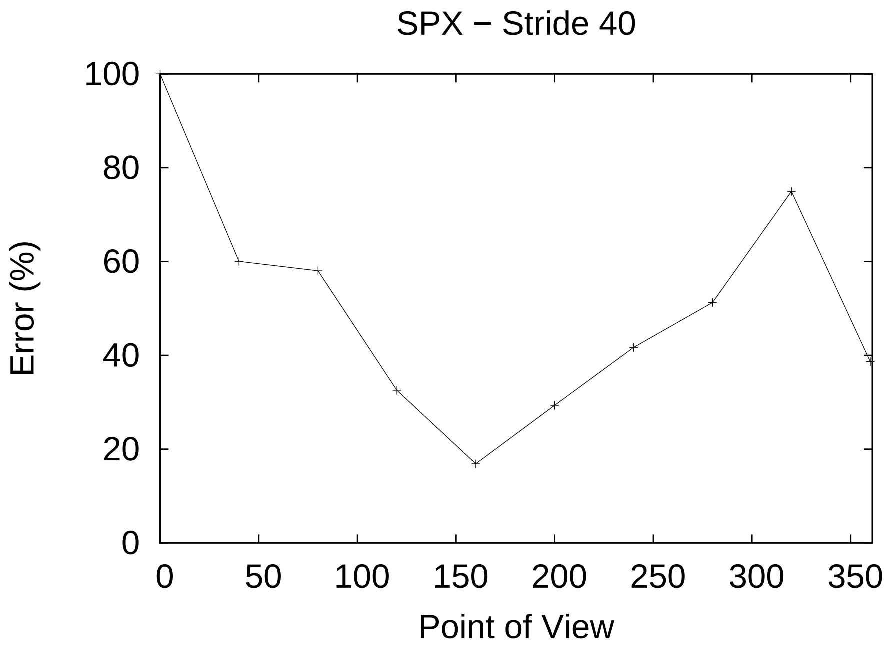

Figure 5.10: Accuracy of load estimation for SPX with a stride of 40^o.

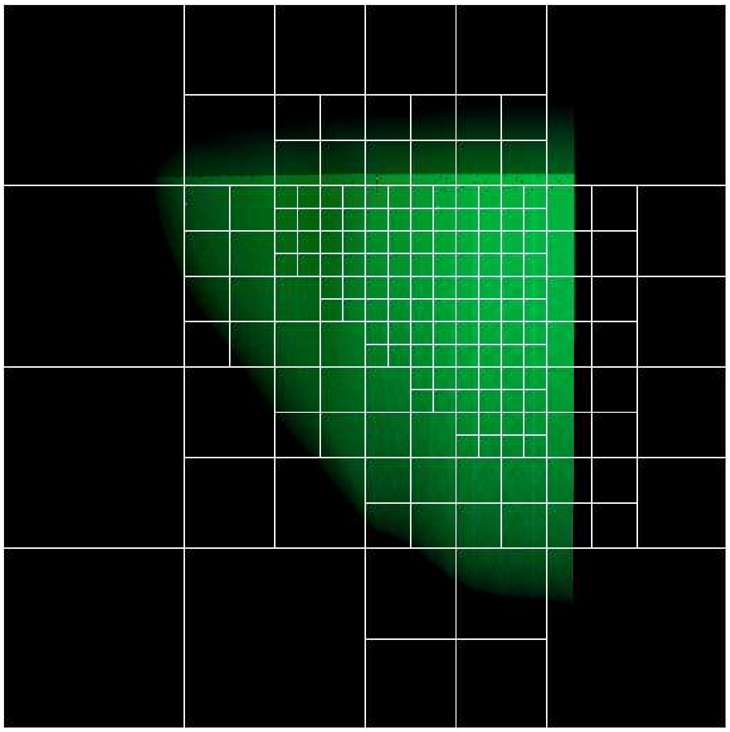

Figure 5.11: Example of tile decomposition for the Delta dataset.

than 21% of the standard deviation of the fixed division. The increasing in the percentage shown for 18^o is due to the increasing in the error of the tile load estimation measure. An increasing in such measure degrade the decomposition effectiveness.

Datasets	Angle stride				
	2	6	10	14	18
SPX	16.7%	18.3%	20.3%	24.4%	41.2%
Post	12.6%	18.0%	24.3%	38.0%	65.8%
Delta	16.5%	17.4%	18.6%	20.6%	33.1%
Fighter	20.1%	21.1%	25.1%	27.2%	40.0%
Torso	18.1%	19.9%	21.5%	25.3%	53.7%

Table 5.4: Standard deviation comparison with the fixed division.

Figure 5.11 illustrates an example of the tile decomposition for the Delta dataset. It can be seen that our adaptive strategy performs more subdivision in the middle of the screen where there is more data to be computed.

Table 5.5 depicts, for each dataset, the percentage of time that the parallel algorithm spent in the computation of tile decomposition and distribution step. This time measurement is done in the same way as the overhead breakdown. As we can observe, the overhead for this step is very small, under 3.6% of the total rendering time, for most of the datasets, except for Fighter, that obtained 6.6% of tile decomposition and distribution overhead. Fighter is the dataset that has the smallest number of empty tiles, and the biggest number of the rendered tiles, as its image uses up all the space in the screen. In addition, for Fighter the tiles costs are more homegeneous than for the other datasets, and, in this case, the tree rearrangement algorithm takes more time to find the critical value. For the other datasets, the overhead slightly increases with the increase in the dataset. It is important to notice that, in terms of memory consumption, the quadtree used only around 25 Kbytes for the biggest dataset, which is less than 0.1% of the memory needed for the rendering.

Dataset	Overhead
Post	1.7%
SPX	2.5%
Delta	3.6%
Torso	2.8%
Fighter	6.6%

Table 5.5: Overheads incurred by tile decomposition and distribution strategies.

To analyze the tile distribution, we compared the load imbalance generated by the use of Makespan compared to the load imbalance generated when the tiles are randomly assigned to the rendering threads in a round-robin fashion. The load imbalance was computed using equation (4.10) and the Makespan heuristic provided improvements from 35% to 75% in the load imbalance, when compared to the Random tile distribution scheme.

5.4 Pre-render

As can be observed in Figure 5.6, the overhead of the pre-render step is significant, about 15% of the total rendering time. Following, we analyze each of the components of this overhead, data rotation and face projection, in detail.

5.4.1 Data Rotation

The data rotation phase represents only a small part of the pre-render computation. It is independent of the image resolution and the geometry of the data, it depends only on the number of vertices of the data. In Table 5.6, we show the time spent in data rotation and the percentage that this time represents on the overall execution time. As we can observe, this phase imposes only a negligible overhead in the parallel rendering.

5.4.2 Face Projection

The face projection phase, on the other hand, accounts for most of the overhead of the pre-render step. This is a necessary phase in any raycasting algorithm, whose performance implications are often neglected by parallel rendering systems, mainly when the image generated has high resolution. Tables 5.8, 5.7 and 5.9 show the time taken for projecting the visible faces in the screen, when the image generated has the resolution of 8G pixels, 4G Pixels and 1G pixels, respectively. The tables show the face projection time and the percentage of this time in the overall execution time for the parallel schemes proposed here, faces-per-tile and faces-per-quadtree, and for a traditional scheme, where all the visible faces are projected. This traditional scheme is simpler and avoids the need of searching for the visible faces that project inside the tiles assigned to each thread. However, the results show that, when the image has high resolution, the unnecessary projections turn the projection from 20 to 70 times slower for a 4G pixel resolution, and from 50 to 800 times slower for a 8G pixel resolution. The gains get smaller for low resolution images as seem in Table 5.9. That is why many of the parallel renders which handles low resolution images never parallelize this step of the pipeline. The face-per-tile technique has even had worse projection time when compared to the traditional parallel algorithm in a 1G pixel resolution, but the faces-per-quadtree has always had the lower projection time when compared to the other two approaches even for such small image resolution.

When we compare the two parallel approaches, faces-per-tile and faces-per-quadtree, we observe that the faces-per-quadtree is faster than the faces-per-tile scheme for all datasets, except Torso. For a 4G pixel resolution, the face-per-quadtree is about 26% faster for Fighter dataset. However, when the number of pixels quadruplicates, we observe huge gains for the faces-per-quadtree scheme. It is about 46% faster for Fighter dataset than faces-per-tile. Figther is the dataset where face-per-quadtree presents the greatest gains, since this dataset is the one that has the least number of non-rendered pixels in the screen. Torso, on the other hand, has the greatest number of non-rendered pixels and

Datasets	Rotation time(ms)	% Total Time
Post	5.79	0.09%
SPX	9.15	0.22%
Delta	13.33	0.29%
Torso	10.71	0.24%
Fighter	16.11	0.37%

Table 5.6: Rotation timing.

the least number of visible faces to be projected.

Datasets	Faces-per-quadtree		Faces-per-tile		Traditional	
	% Total Time	time(ms)	% Total Time	time(ms)	% Total Time	time(ms)
SPX	15.67%	660.02	18.49%	807.98	55.59%	13287.95
Delta	14.65%	662.78	14.98%	682.59	53.86%	24492.57
Fighter	16.07%	709.01	33.63%	1881.24	48.24%	49629.69
Post	10.36%	661.12	11.17%	720.18	66.5%	40311.76
Torso	14.44%	648.32	14.26%	640.5	50.98%	19804.27

Table 5.7: Face projection timing for 4G pixel resolution.

Datasets	Faces-per-quadtree		Faces-per-tile		Traditional	
	% Total Time	time(ms)	% Total Time	time(ms)	% Total Time	time(ms)
SPX	7.63%	1439.93	13.06%	2619.23	46.84%	84782.84
Delta	13.24%	2551.59	13.68%	2639.26	47.4%	1676786.92
Fighter	16.73%	3066.68	41.72%	10863.52	49.14%	2460465.88
Post	5.14%	1333.73	9.7%	2643.3	43.84%	82034.43
Torso	14.32%	2945.72	12.78%	2583.58	38.23%	167853.85

Table 5.8: Face projection timing for 8G pixel resolution.

Datasets	Faces-per-quadtree		Faces-per-tile		Traditional	
	% Total Time	time(ms)	% Total Time	time(ms)	% Total Time	time(ms)
SPX	4.23%	57.53	35.12%	302	16.58%	98.75
Delta	13.26%	60.64	14.6%	68.71	18.15%	112.85
Fighter	13.86%	98.44	42.17%	396.41	30.91%	259.71
Post	7.79%	50.39	11.09%	92.01	15.87%	134.29
Torso	6.89%	46.15	7.38%	49.47	11.66%	88.56

Table 5.9: Face projection timing for 1G pixel resolution.

5.5 Rendering

The rendering process in each rendering thread follows the same computation as proposed in [41]. In the rendering algorithm, as a ray traverses the dataset, three computations take place: Find Next Face, Compute α, and Update Light. Find Next Face, discovers which the next face to be intersected is, and the intersection point of the ray in this next face. It generates the z coordinate of this point. Compute α calculates the scalar value of the intersection point. The scalar value, α, is computed by the interpolation of the scalar values of the next face vertices. Update Light computes the illumination integral from the current position to the next intersection, using the values of z and α computed previously. Table 5.10 shows the proportion of these three computations in the overall rendering time. Find Next Face is the most expensive part of the rendering process, accounting for 66.4% of the rendering time. This is due to the update coefficient computation that calculates the coefficient of the plane defined by the three vertices of a triangular face. Update Light accounts for 29% of the rendering, due to the illumination integral computation and Compute α accounts for only the other 4.6%.

Computation	% of Rendering Time
Find Next Face	66.4%
Compute α	4.6%
Update Light	29%

Table 5.10: The contribution of each part of the rendering in the rendering time.

In the parallel algorithm, however, the tile division can affect the cache behavior of the rendering process. So, we compared the cache utilization of the parallel algorithm with the cache utilization of the sequential algorithm. The cache utilization was measured using PAPI [40] and Perfctr library to access the hardware counters.

Table 5.11 shows L1 cache miss rate of each of the three computations performed during the rendering step, for the parallel and the sequential algorithm. Each core of the cluster has 32K of L1 cache. As we can observe in this table, the adaptive tile decomposition strategy enhances the cache utilization. This occurs because our tile decomposition strategy divides the computation evenly among the tiles and nearby rays that lie inside the tiles tend to intersect the same faces, so that the faces data can be reused in the cache. When we compare the three main functions of the rendering, we observe that Update Light is the one that generated the highest cache miss rate. This is due to the need of reading a color table, in order to insert in the integral the range of the color values for the α computed. The Find Next Face and Compute α function, on the other hand, had benefited from the reuse of face data.

Computation	Cache Miss Rate	
	Parallel Algorithm	Sequential Algorithm
Find Next Face	0.24%	1.15%
Compute α	0.14%	1.33%
Update Light	1.16%	1.21%

Table 5.11: Cache miss rate of rendering for the parallel and sequential algorithm.

5.6 Subimages Transmission

The subimages send overhead is the time the master spent in waiting for the subimages to arrive before merging them into the final image. The master has to wait for the rendering threads to finish their work, and for the messages to arrive. So, this overhead is due not only to the network message exchanging, but also to the load imbalancing effect. Following we discuss both overheads in detail.

5.6.1 Message Exchanging

Table 5.12 shows the time spent with message exchanging and the percentage of time that the message exchanging represents in the subimages send overhead. As we can observe in this table, the time spent in message exchanging is negligible in the total subimages transmission overhead. For Torso, the message exchange overhead is the smallest, even being bigger than Post and SPX. This is primarily due to the fact that Torso is a regular dataset. As our algorithm can also deal with hexahedral voxels, we considered each cubic voxel of Torso as an hexahedral voxel. Due to the regular nature of Torso, some voxels have no contribution for the final color of the pixels. Nevertheless, the image generated by Torso uses up the smallest screen space, compared to the other datasets. Fighter has the greater network overhead, this fact is due to the nature of Fighter dataset. Fighter has the greater number of external face, and the faces occupy almost all the image.

Dataset	% of Message exchange Overhead
SPX	9.75%
Post	13.03%
Delta	8.07%
Fighter	7.73%
Torso	6.62%

Table 5.12: Message exchanging percentual time.

We also compared the message exchanging measured time with the theoretical propagation time, when there is no delay in the network. The theoretical propagation time is calculated using the message size and the network bandwidth. In this experiment, we observed that the difference between the measured and the theoretical is very small, under 4%. This result confirms that the network is not imposing delays in the message exchange, and the communication with the master does not generate bottlenecks.

5.6.2 Load Imbalancing

Table 5.13 shows the percentage of load imbalance generated by our algorithm using equation (4.10). As we can observe in this table, for most of the datasets, our algorithm generated less than 10% of load imbalance. For Fighter, the load imbalance is 7.81%. All these results, for such a big image resolution (4G pixels), are very good, since according to Mueller[35], a load imbalance is reasonable if it is below 33%.

Dataset	Load Imbalance
SPX	8.00%
Post	3.09%
Delta	5.28%
Fighter	4.21%
Torso	7.90%

Table 5.13: Load imbalance of our algorithm, according to equation (4.10).

In order to put our load imbalance results in perspective, we compared our parallel algorithm results with a plain parallel version. This plain parallel version divides the screen equally into 32×32 tiles and distributes the tiles randomly across the rendering threads. Table 5.14 shows the percentage of gains in the load imbalance obtained by our algorithm. As we can observe in these tables, our algorithm obtained significant reductions in the load imbalance, when compared to the plain tile division and distribution These results confirm that the load imbalance problem has great impact on the overall performance of a parallel rendering algorithm.

Dataset	% of Gains
SPX	83.37%
Post	85.92%
Delta	84.97%
Fighter	81.86%
Torso	85.61%

Table 5.14: Gains in load imbalance of our algorithm, when compared to a plain parallel rendering algorithm.

Another important result observed in our parallel algorithm is that the load imbalance does not increase significantly with the increase in the number of cores, while the opposite occurs for the plain parallel version. In our algorithm, the adaptive tile decomposition ensures a good number of tiles to be distributed to the cores. For the plain algorithm, on the other hand, the tile decomposition is fixed. So, as the number of cores increases, less tiles are given to each core, increasing the chances of having unbalanced load. Figures 5.12 to 5.16 show the comparison of the loadimbalance for all the datasets for PRay and traditional parallel raycast. Even though the Fighter dataset, shown in Figure 5.14, had the best load balance performance for traditional parallel raycast, this result is not even near the worst load balance performance of PRay algorithm shown in Figure 5.12 for SPX

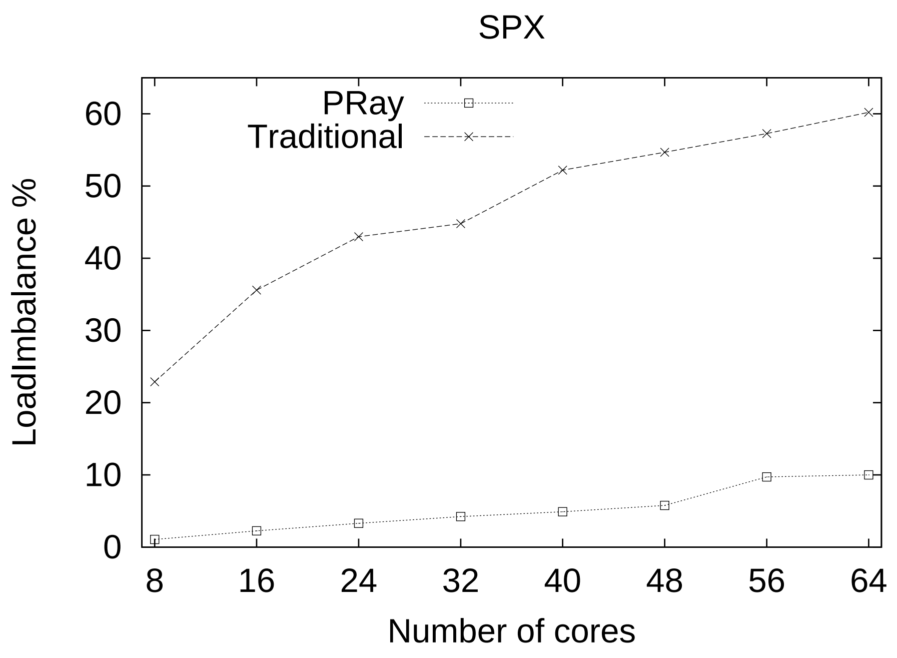

Figure 5.12: Overhead of PRay and traditional raycast for SPX dataset.

dataset.

5.7 Image Merge

The image merge overhead includes the time spent for the rendering threads to encode their subimages and for the Master to decode the subimages received and save them in the final *Raster*. Table 5.15 depicts, for each dataset, the percentage of time that the parallel algorithm spent in the computation of encoding and decoding the subimages. As we can observe in this table, the time spent due to image merging is always lower than 1.15% of the total execution time for all datasets. This is the smallest overhead of the parallel computation.

Datasets	Overhead
Post	0.83%
SPX	1.12%
Delta	1.15%
Torso	1.04%
Fighter	1.3%

Table 5.15: Overhead of image merging

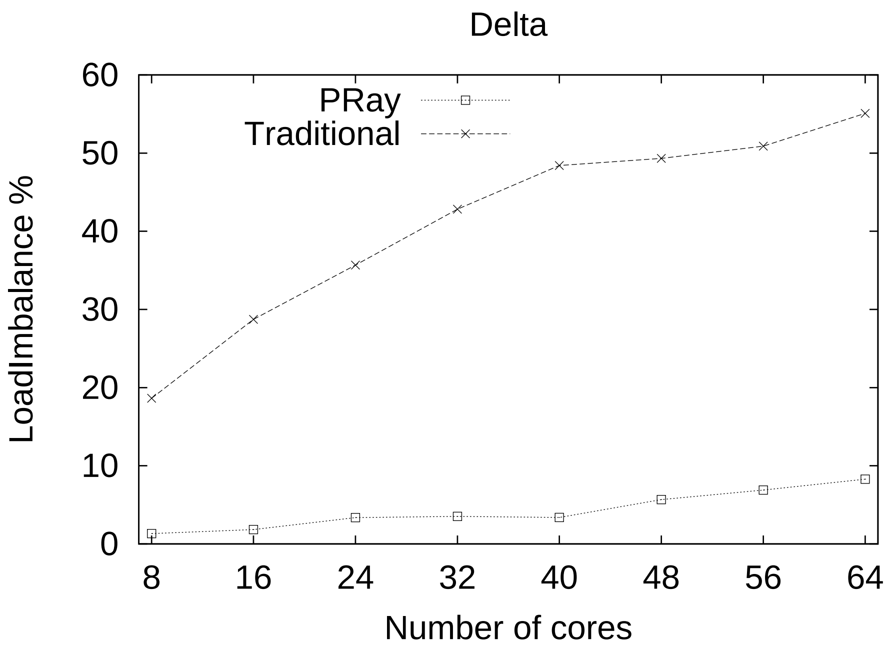

Figure 5.13: Overhead of PRay and traditional raycast for Delta dataset.

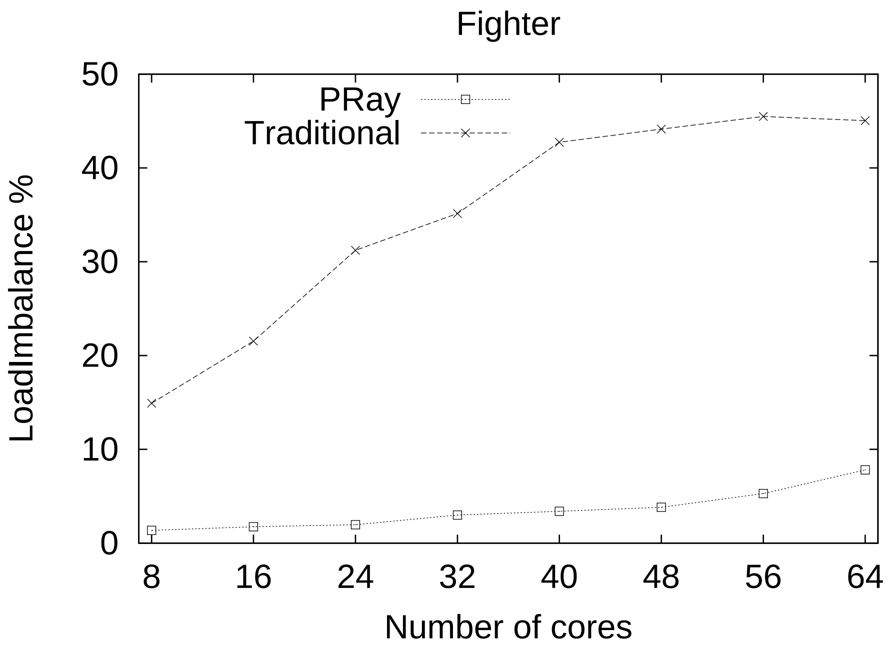

Figure 5.14: Overhead of PRay and traditional raycast for Fighter dataset.

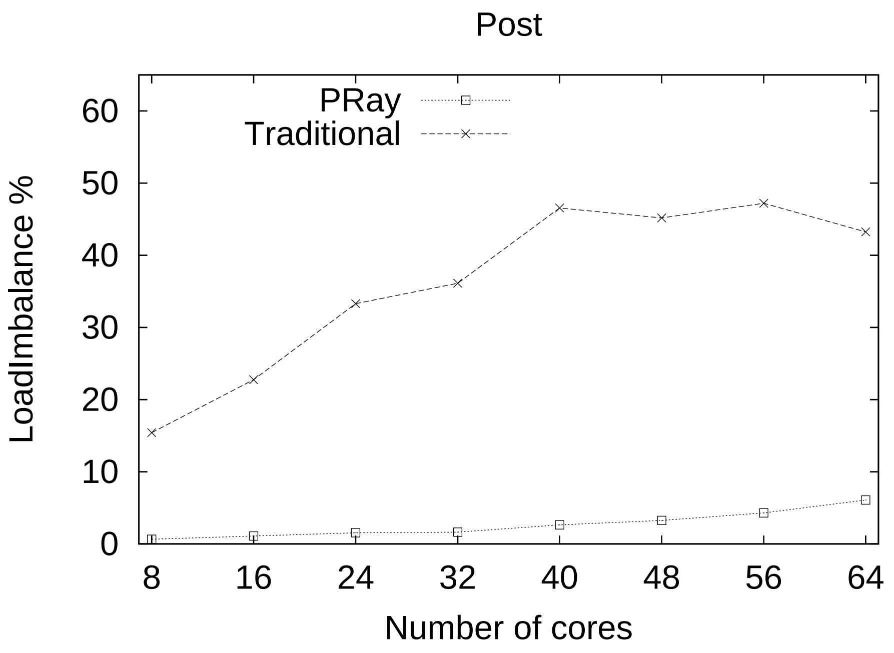

Figure 5.15: Overhead of PRay and traditional raycast for Post dataset.

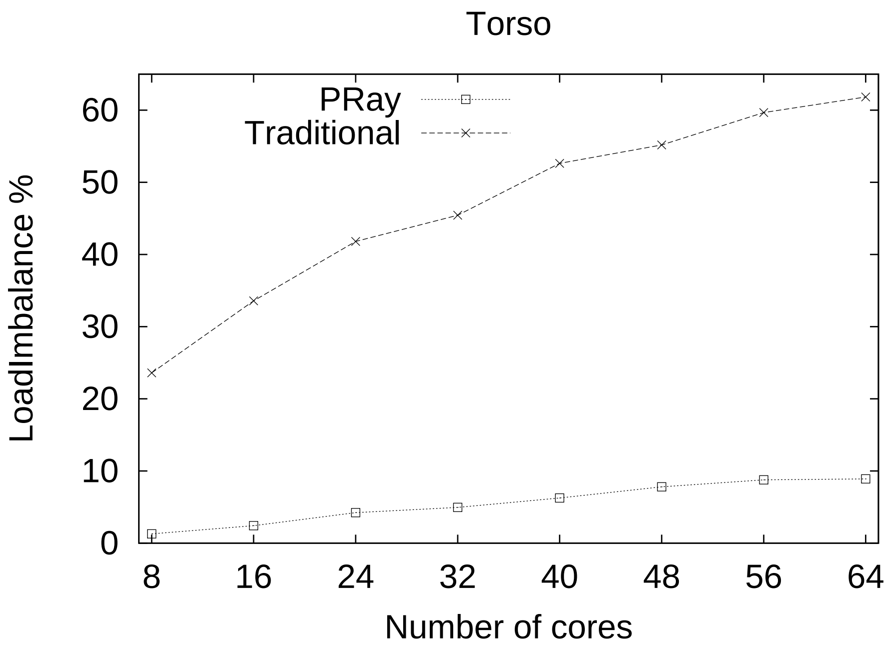

Figure 5.16: Overhead of PRay and traditional raycast for Torso dataset.

5.8 Speedup Results

The speedup is a metric used to measure the algorithm is defined by Equation (5.1) where S is the speedup, $T_{Parallel}$ is the time of the parallel algorithm, T_1 is the time spent by the sequential algorithm. The time T_1 was calculated using the sequential algorithm ME-Ray. The PRay was compared with the traditional parallel raycast in the same way that was done in section 5.6.2.

$$S = 1 - \frac{T_1}{T_{Parallel}} \tag{5.1}$$

Equation (5.2) shown the metric used to measure the efficiency of the parallel algorithm. F is the efficiency. S is the speedup and C is the number of cores used to obtain the speedup S.

$$F = \frac{100 * S}{C} \tag{5.2}$$

Figs. 5.17 to 5.21 show the speedup obtained by PRay and traditional parallel raycast for each dataset as the number of rendering threads increases. We can observe in these figures that the speedups obtained by PRay are quite high, and always greater than the speedups obtained by PRay. The dataset which achieved the best result was the Post dataset with a speed up of almost 51, which results in a efficiency of 79% for the parallel algorithm. For the same dataset, traditional parallel raycast only achieved an efficiency of 42%. Fighter dataset achieved the worse speedup among all the datasets, a speedup of 34.5 for PRay and 24.5 for the traditional parallel raycast. This performance can be easily explained by the *pre-render* and *subimage sending* overheads already discussed.

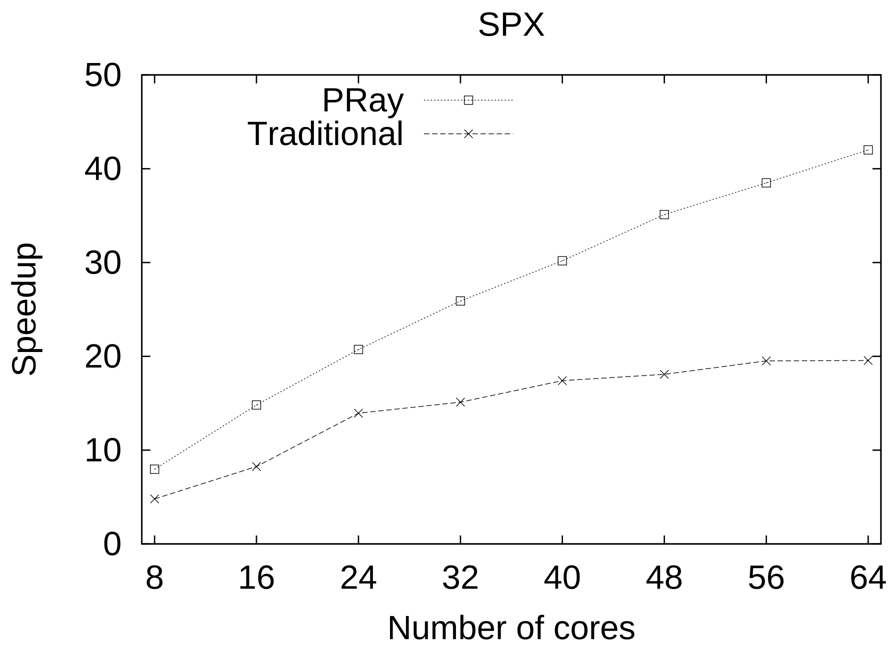

Figure 5.17: Speedup of PRay and traditional raycast for SPX dataset.

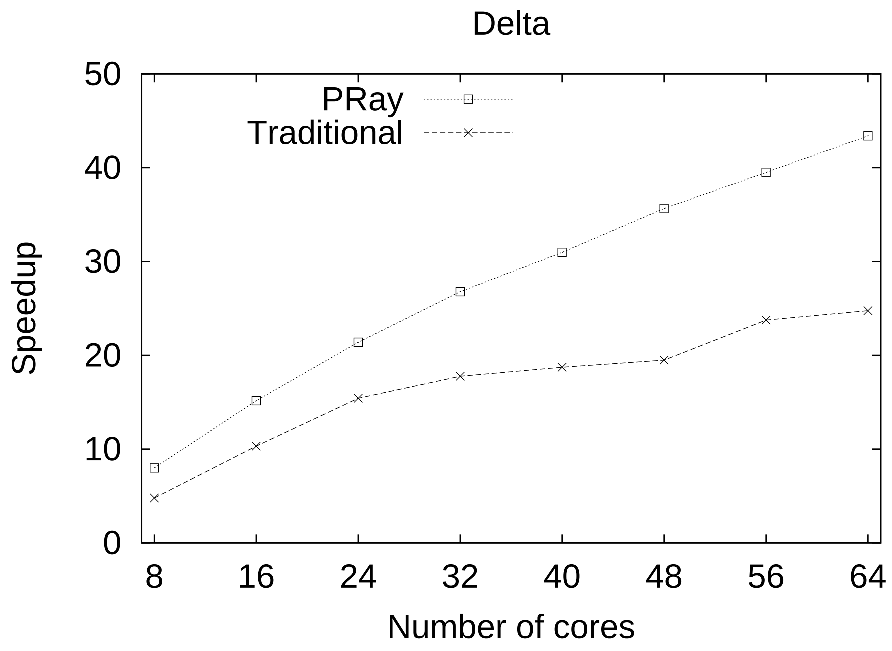

Figure 5.18: Speedup of PRay and traditional raycast for Delta dataset.

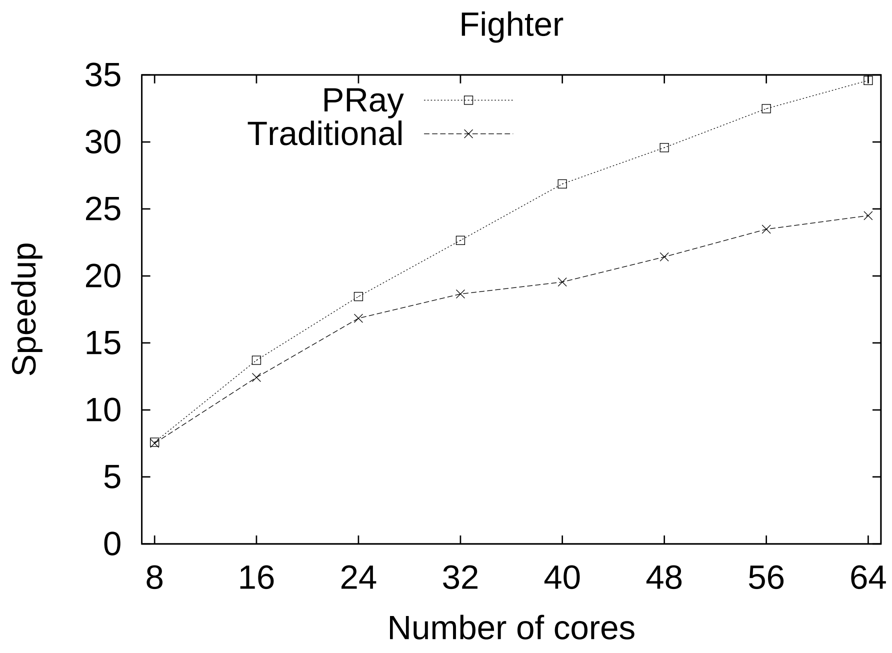

Figure 5.19: Speedup of PRay and traditional raycast for Fighter dataset.

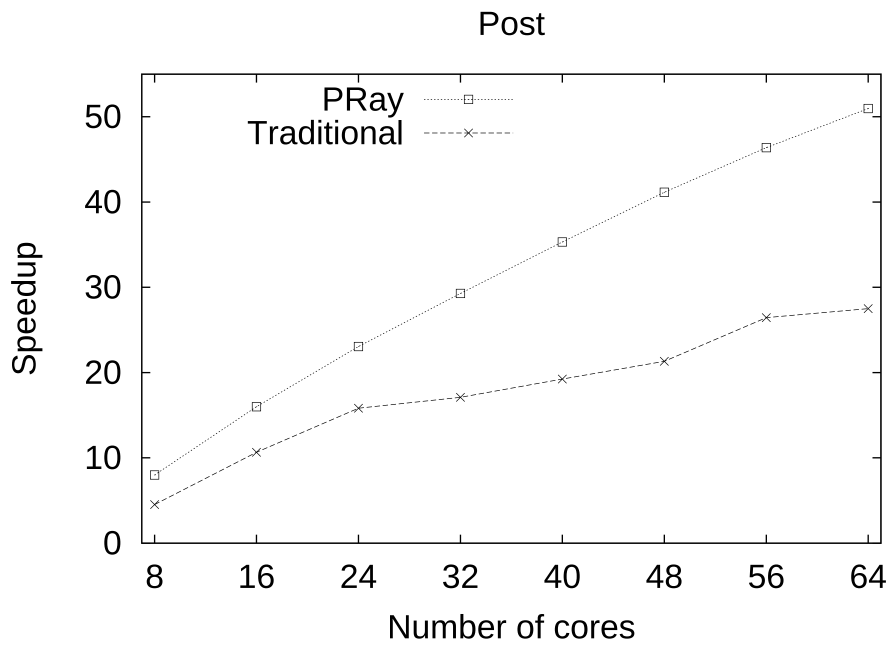

Figure 5.20: Speedup of PRay and traditional raycast for Post dataset.

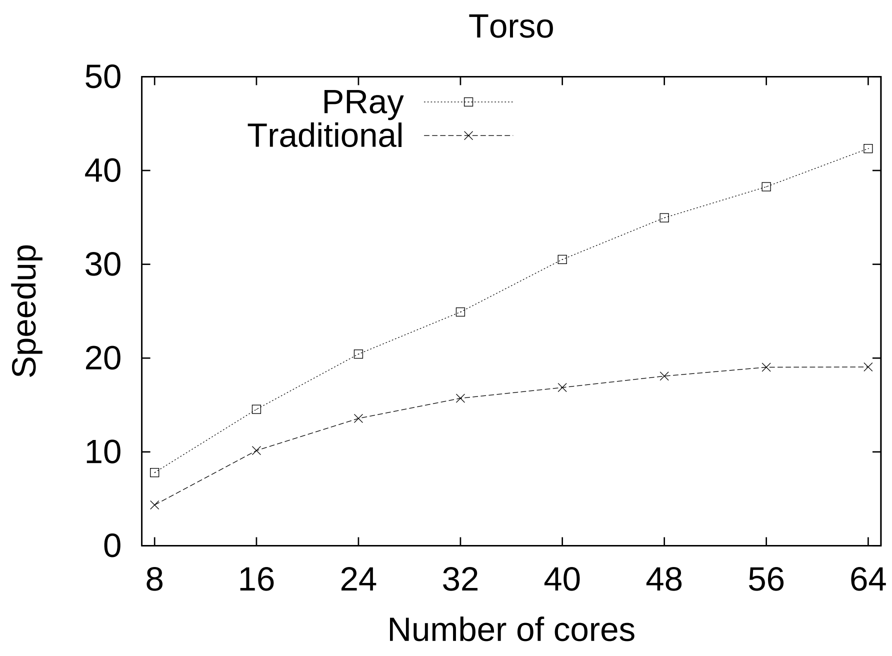

Figure 5.21: Speedup of PRay and traditional raycast for Torso dataset.

Chapter 6

Concluding Remarks

In this work, we propose a new parallel raycasting for unstructured grids that is based on sort-first division of the rendering task and was designed to explore the recent architecture of clusters of multicores. We dissected all the overhead components of the parallel rendering algorithm, identifying bottlenecks and suggesting modifications in order to handle images with high-resolutions. Our evaluation included all the parallel aspects of the rendering process, including decomposition/load balancing, face projection, data locality during the dataset traversal, and communication overhead.

We divided the parallel computation into five steps: tile decomposition and distribution, pre-render, rendering, subimages send, and image merge. For each of these steps, we applied different techniques to improve performance. In the tile decomposition and distribution step, we applied an adaptive tile decomposition strategy that hierarchically subdivides the screen using a quadtree structure and uses the the concept of entropy as the stopping criteria. The tile distribution is done using the Makespan heuristic. For the pre-render step, where the data is rotated and the entry point of the rays are found by visible faces projections, we applied a parallel face projection scheme that took advantage of the quadtree structure built in the tile decomposition step. For the rendering step, we used a memory-efficient sequential algorithm that benefited from an even tile division and provided better cache behavior. For the subimages transmission and image merge steps, we applied an image encoding scheme in order to reduce message sizes.

Our parallel rendering algorithm obtained significant performance gains when compared to a plain parallel raycasting algorithm for a 4G pixel resolution image. Tile decomposition and distribution schemes included negligible overhead and improved the load balancing in at most 86%. The parallel face projection scheme proved to be indispensable for high-resolution image rendering, since it provides gains of up to 26% and up to 46% for 8G pixel resolution. The sequential rendering in each thread improved the cache hit in about 89%, due to the tile division. Considering the whole rendering process, we obtained speedup gains of up to 51%.

One important aspect of our study is that, although the solutions proposed to reduce parallel overheads are employed in our specific algorithm, the lessons learned can be possibly extensively applied to other parallel direct volume rendering approaches, such as GPGPU versions of raycast.

References

[1] Abraham, F.Celes, W.Cerqueira, R.Campos. J. A load-balancing strategy for sort-first distributed rendering. In *17th Brazilian Symposiumon Computer Graphics and Image Processing* (2004), p. 292–299.

[2] Aguilar, E. *Automatos celulares generalizados como modelo de influência para agrupamento de dados e interações sociáis*. PhD thesis, COPPE Universidade federal do Rio de Janeiro, Brazil, Rio de Janeiro, 2008.

[3] Allard, J.Raffin. B. A shader-based parallel rendering framework. In *Proceedings of IEEE Visualization conference* (2005), p. 127–134.

[4] Aykanat, C. Cambazoglu, B.B. Findik, F. Kurc, T. Adaptive decomposition and remapping algorithms for object-space-parallel direct volume rendering unstructured grids. *Journal of Parallel Distributed Computing 67*, 1 (2007), 77–99.

[5] Bernardon, F.F. Pagot, C.A. Luiz Dihl Comba, J. Silva, C.T. Gpu-based tiled ray casting using depth peeling. *Journal of Graphic tools 11*, 3 (2006), 23–29.

[6] Childs, H.Duchaineau, M. Ma, K.L. A scalable, hybrid scheme for volume rendering massive data sets. In *Proceeding of Eurographics Symposium on Parallel Graphics and Visualization* (2006), p. 153–162.

[7] Chopra, P.Meyer, J. Tetfusion: an algorithm for rapid tetrahedral mesh simplification. In *VIS '02: Proceedings of the conference on Visualization '02* (2002), p. 133–140.

[8] Coelho, A. Lopes, A. Bentes, C. Castro, M. Farias, R. Distributed load balancing algorithms for parallel volume rendering on cluster of pcs. In *XXXII Conferencia Latinoamericana de Informtica CLEI 2006* (2006), p. 51–58.

[9] Cox, G. Maximo, A. Bentes, C. Farias, R. Irregular grid ray-casting implementation on cell broadband engine. In *Proceedings of the 2009 21st International Symposium on Computer Architecture and high performance Computing* (2009), p. 93–100.

[10] Danskin, J. Fast algorithm for volume ray tracing. In *VVS '92: Proceedings of the 1992 workshop on Volume visualization* (1992), p. 91–98.

[11] Espinha, R.Celes, W. High-quality hardware-based ray-casting volume rendering using partial pre-integration. In *SIBGRAPI '05: Proceedings of the XVIII Brazilian Symposium on Computer Graphics and Image Processing* (2005), p. 273–281.

[12] Espinha, R.Celes, W. High-quality hardware-based ray-casting volume rendering using partial pre-integration. In *SIBGRAPI '05: Proceedings of the XVIII Brazilian Symposium on Computer Graphics and Image Processing* (2005), p. 273–281.

[13] Farias, R.Bentes, C.Coelho, A.Guedes, S.Goncalves, L. Work distribution for parallel zsweep algorithm. In *XI Brazilian Symp. on Computer Graphics and Image Processing* (October 2003), p. 107–114.

[14] Finkel, R.A.Bentley, J.L. Quad trees a data structure for retrieval on composite keys. *Acta Informatica 4*, 1 (march 1974), 1–9.

[15] Fout, N. Ma, K.L. Transform coding for hardware-accelerated volume rendering. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (November 2007).

[16] Garland, M. Zhou, Y. Quadric-based simplification in any dimension. *ACM Trans. Graph. 24*, 2 (2005), 209–239.

[17] Graham, R. L. Bounds on multiprocessing anomalies and related packing algorithms. In *AFIPS '72 (Spring): Proceedings of the May 16-18, 1972, spring joint computer conference* (New York, NY, USA, 1972), ACM, p. 205–217.

[18] Humphreys, G. Houston, M. Ng, R. Frank, R. Ahern, S. Kirchner, P. Klosowski, J.T. Chromium: A stream processing framework for interactive rendering cluster. In *SIGGRAPH 2002: Computer Graphics Proceedings* (2002).

[19] Kim, J. Jaja, J. Streaming model based volume ray casting implementation for cell broadband engine. *Scientific Programming 17*, 1–2 (2009), 173–184.

[20] Kutluca, H. Kurç, T. Aykanat, C. Image-space decomposition algorithms for short-first parallel volume rendering of unstructured grids. *Journal of Supercomputing 15*, 1 (2000), 51–93.

[21] Labronici, B. B. Bentes, C. Drummond, L. Farias, R. . Dynamic screen division for load balancing the raycasting of irregular data. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on* (New Orleans, LA, August 2009), no. 978-1-4244-5011-4 in 1552-5244, IEEE, p. 1–10.

[22] Labronici, B. B. Bentes C. Farias R. . Paralelização do algortimo de raycast, university of rio de janeiro state - uerj. Monograph required to obtain the tile of Electrical Engineer, December 2006.

[23] Lee J.K. Newman, T.S. Acceleration of opacity correction mechanisms for over-sample volume ray-casting. In *EGPGV '08: Symposium on Parallel Graphics and Visualization* (2008), p. 22–30.

[24] Ma, K. Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. In *IEEE Parallel Rendering Symposium* (1995), p. 23–30.

[25] Ma, K. L. Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. In *IEEE Parallel Rendering Symposium* (October 1995), p. 23–30.

[26] Ma, K.L. Crockett, T. . A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data. In *IEEE Parallel Rendering Symposium* (November 1997), p. 95–104.

[27] Marchesin, S.Mongenet, C.Dischler, J. M. Dynamic load balancing for parallel volume rendering. In *Eurographics Symposium on Parallel Graphics and Visualization* (Braga, Portugal, 2006).

[28] Marchesin, S.Mongenet, C.Dischler, J. M. Multi-gpu short-last volume visualization. In *EGPGV '08: Symposium on Parallel Graphics and Visualization* (2008), p. 1–8.

[29] Marroquim, R. Maximo, A. Farias, R. Esperança, C. . Volume and isosurface rendering with gpu-accelerated cell projection. In *Computer Graphics Forum* (2008), vol. 27, p. 24–35.

[30] Max, N. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics 1*, 2 (1995), 99–108.

[31] Maximo, A. Ribeiro, S. Bentes, C. Oliveira, A. Farias, R. . Memory efficient gpu-based ray casting for unstructured volume rendering. In *IEEE/EG Int. Symp. Volume and Point-Based Graph* (2008), p. 55–62.

[32] Meissner, M. Hüttner, T. Blochinger, W. Weber, A. . Parallel direct volume rendering on pc networks. In *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications* (July 1998).

[33] Molnar, S. *Image-composition architectures for real-time image generation.* PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1992.

[34] Moloney, B.Weiskopf, D.Moller, T.Strengert, M. Scalable sort-first parallel direct volume rendering with dynamic load balancing. In *Eurographics Symposium on Parallel Graphics and Visualization* (Lugano, Switzerland, 2007), p. 45–52.

[35] Mueller, C. he sort-first rendering architecture for high-performance graphics. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics* (1995), p. 75–84.

[36] Mueller, C. Hierarchical graphics databases in sort-first. In *PRS '97: Proceedings of the IEEE symposium on Parallel rendering* (1997), p. 49–57.

[37] Muller, C.Strengert, M.Erl, T. Optimized volume raycasting for graphics-hardware-based cluster systems. In *Eurographics Symposium on Parallel Graphics and Visualization* (Braga, Portugal, 2006), p. 59–66.

[38] Muraki, S. Lum, E. Ma, K.L. Ogata, M. Liu, X. . A pc cluster system for simultaneous interactive volumetric modeling and visualization. In *Proc. of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (October 2003).

[39] Nieh, J. Levoy, M. Volume rendering on scalable shared-memory mimd architectures. In *Proc. of the 1992 Workshop on Volume Visualization* (October 1992), p. 17–24.

[40] Papi. http://icl.cs.utk.edu/papi/, 2009.

[41] Pina, A. Bentes, C. Farias, R. Memory efficient and robust software implementation of the raycast algorithm. In *WSCG'07: The 15th Int.Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision* (2007).

[42] Ribeiro, S. Maximo, A. Bentes, C. Oliveira, A. Farias, R. . Memory-aware and efficient ray-casting algorithm. In *SIGGRAPH '07: Proceedings of the XX Brazilian Symposium on Computer Graphics and Image Processing* (2007), p. 147–154.

[43] Roth, M. Rieb, P. Reiners, D. Load balancing on cluster-based multiprojector display systems. In *14th International Conference in Central Europe on Computer Graphics Visualization and Computer Vision* (2006), p. 55–62.

[44] Roth, S.D. . Ray casting for modeling solids. *Computer Graphics and Image Processing 18* (February 1982), 109–144.

[45] Ruijters, D. Vilanova, A. . Optimizing gpu volume rendering. In *WSCG '06: The 15th International conference in Central Europe on Computer Graphics, Visualization and Computer Vision* (2006).

[46] Samanta, R. Funkhouser, T. Li, K. Singh, J.P. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *Proc. of the SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2000).

[47] Samanta, R. Zheng, J. Funkhouser, T. Li, K. Singh, J.P. . Load balancing for multi-projector rendering systems. In *Proc. of the SIGGRAPH/Eurographics Workshop on Graphics Hardware* (August 1999), p. 107–116.

[48] Shannon, C. A mathematical theory of communication. *Bell System Technical Journal 27* (October 1948), 379–423.

[49] Smelyanskiy, M. Holmes, D. Chhugani, J. Larson, A. Carmean, D.M. Hanson, D. Dubey, P. Augustine, K. Kim, D. Kyker, A. Lee, V.W. Nguyen, A.D. Seiler, L. Robb, R. . Mapping high-fidelity volume rendering for medical imaging to cpu, gpu and many-core architectures. *IEEE transaction on Visualization and Computer Graphics 15*, 6 (2009), 1563–1570.

[50] Top-500. http://top500.org, June 2008.

[51] Wang, W. Yang, J. Muntz, R. Sting: A statistical information grid approach to spatial data mining. In *Proceedings of 23rd International Conference on Very Large Data Bases* (1997), p. 186–195.

[52] Weiler, M. Kraus, M. Merz, M. Ertl, T. Hardware-based ray casting of tetrahedral meshs. In *Proceedings of the 14th IEEE conference on Visualization* (2003), p. 333–340.

[53] Weiler, M. Mallon, P.N. Kraus, M. Ertl, T. Texture-encoded tetrahedral strips. In *VV '04: Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics* (2004), p. 71–78.

[54] Westover, L. Footprint evaluationfor volume rendering. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques* (1990), p. 367–376.

[55] Whitman, S. Dynamic load balancing for parallel polygon rendering. *IEEE Computer Graph. and Appl. 14*, 4 (July 1994), 41–48.

[56] Wylie, B.Pavlakos, C.Lewis, V.Moreland, K. Scalable rendering on pc clusters,. *IEEE Computing 21*, 4 (2001), 62–70.

[57] Yu, H. Wang, C. Ma, K.L. Parallel volume rendering using 2-3 swap image compositing for an arbitrary number of processors. In *Proceedings of IEEE/ACM Supercomputing conference 2008.*