Parallel Algorithms for Rendering Large 3D Models on a Graphics Cluster

Alexandre Beaudoin

A Thesis

in

The Department

of

Computer Science and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montréal, Québec, Canada

April 2012

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared:

By: Alexandre Beaudoin

Entitled: Parallel Algorithms for Rendering Large 3D Models on a Graphics Cluster

and submitted in partial fulfillment for the degree of:

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. J. Paquet

_____ Examiner
Dr. G. Butler

_____ Examiner
Dr. T. Radhakrishnan

_____ Supervisor
Dr. D. Goswami

_____ Supervisor
Dr. S. Mudur

Approved by: _____
Chair of Department or Graduate Program Director

_____ 20____ _____
Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Parallel Algorithms for Rendering Large 3D Models on a Graphics Cluster

Alexandre Beaudoin

We address the problem of distributing rendering computations for real-time display of very complex three dimensional (3D) scenes using a graphics cluster. The rendering of 3D scenes is increasingly being carried out using at least two different programs on the graphics processing unit (GPU): a vertex shader program for vertex (geometry) processing, and a fragment shader program for pixel (colour) processing. With fragment shader programs becoming more and more time consuming, distributing load solely based on geometry – as is done in most contemporary systems – can cause significant load imbalance and redundant work.

In this thesis we propose a number of parallel rendering algorithms which divide the traditional cluster rendering pipeline into two different phases: one which primarily concerns itself over vertex operations to generate depth information, and a second which primarily concerns itself over fragment operations. By performing communication between these two phases, each node can perform fewer fragment operations with little overhead over traditional cluster rendering algorithms. We also propose a number of load-balancing algorithms which utilize the information gained earlier in the pipeline to improve the management of GPU resources with the aim of improving performance. The techniques are implemented on a graphics cluster and experimental results demonstrate significant improvements in rendering performance.

Acknowledgements

I would like to thank my two supervisors, Dr. Goswami and Dr. Mudur: their time, their patience, and their understanding.

Many thanks for everyone at work, for your understanding and for giving me the flexibility to spend time on my thesis.

Many, many, thanks to my friends and family. For all their encouragements, all the time they spent revising my thesis and all their suggestions. It may take the rest of my life to repay them for all their efforts.

Most importantly, most sincere thanks go to my girlfriend, Katherine, who put up with me (my grumpy old self) for all these years.

Without all their help, this thesis would never have been written.

Contents

List of Tables

List of Figures

List of Code Fragments

Chapter 1

Introduction

The last thirty years have seen a revolution in the game, television and motion picture industries. Computers, now commonplace in the home, produce much of what we view. No longer are animations commonly drawn by hand, instead they are generated through the use of computers. Three-dimensional (3D) graphic technology has progressed at breakneck speed, its performance is growing at a much higher pace than that of Central Processing Units (CPUs).

Nevertheless, computer-generated effects can take days to render a single frame of a movie. Games always push the boundaries of what graphic cards can render while producing enough frames per second to be playable. Sometimes, such as in scientific rendering, the situation is mixed. We can have a model necessitating a great deal of resources to render a single frame, but also require interactivity. A common approach to this mixed situation is to use a graphics cluster, where a number of computers are connected to a high-speed network, each renders a portion of a scene and the outputs are merged to a single display.

Rather than in giving each pixel its own colour (i.e., in computing vertex and geometry and not fragment [pixel] colour), much effort has been put into the performance of systems for scenes with highly complex 3D models, most of the time being spent transforming the geometry of these models into pixels. However, the current trend in computer graphics is to keep the geometry as simple as possible and spend the time in rendering each pixel. This results in the failure of some of the assumptions, described further in the thesis, made in current generation cluster-based graphics renderers.

In general, for parallel processing to be efficient, communication, load-balancing costs and redundant computations must be minimized. Current generation cluster-based graphics rendering techniques perform well in minimizing communication costs. However, they ignore load imbalances resulting from pixel colour and/or geometric operations. The problem of redundant operations must also be addressed.

What classes of algorithms can balance overheads and reduce redundant computations? and in what instances should each algorithms be used?

This thesis presents a novel technique, *Two-phase Rendering*, to help overcome these problems and then extends it in a number of ways which is the main contribution of this thesis. This technique minimizes the redundant pixel colour computation, as well as load imbalance, while keeping the communication costs low [4]. We implement each of our algorithms and perform extensive testing to compare and contrast their relative performance to a well known cluster-based rendering algorithm *Sort-last* [13].

We discuss some of the, previous, related works in Chapter 2, including (1) the 3D rendering pipeline, (2) current cluster-based rendering algorithms, (3) common communication algorithms employed when rendering in a cluster environment, finally,

(4) various algorithms for reducing fragment shader load in single Graphical Processing Unit (GPU) systems. *Two-phase Rendering* algorithms and load-balancing techniques developed as part of this research are presented in Chapter 3. Implementation and of these algorithms and the testbed are covered in Chapter 4. Analysis of the test results are to be found in Chapter 5. Chapter 6 consists of conclusions and recommendations for further work.

Chapter 2

Related Works

2.1 3D rendering pipeline

Images processed on the GPU follow a well defined workflow – or pipeline – to produce a final 3D image on the screen, commonly referred as the *3D rendering pipeline*. Note that, on most desktops, for every frame to be displayed, a single GPU runs through the whole pipeline to completion. Please refer to Figure 2.1 and the following for a basic understanding of the steps which make up the 3D rendering pipeline.

(1) The user interacts with the system (user input).

(2) The program supplies the data required to render the scene to the GPU, such as model, colour, and lighting information (preprocessing).

(3) The GPU transforms the vertices into screen coordinates (vertex shader).

(4) The GPU transforms the geometry (geometry shading).

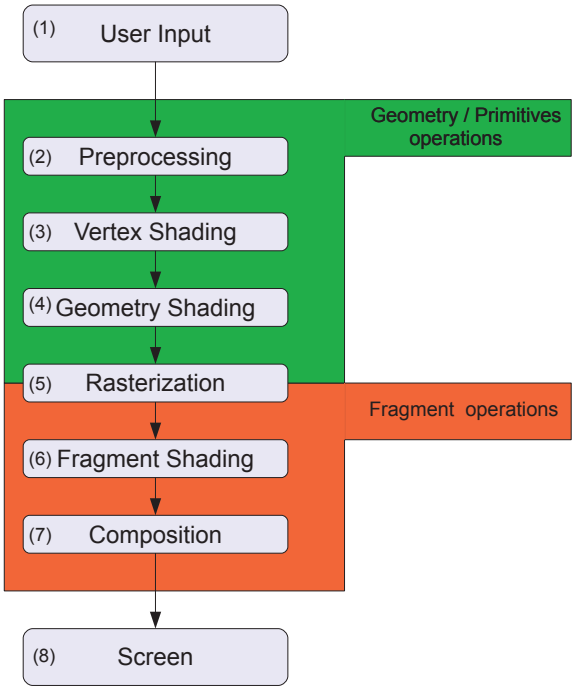

Figure 2.1: Rendering pipeline

(5) The GPU then converts the geometry into fragments (rasterization). Note that in essence, a fragment is the precursor of a pixels in that it has a colour and X-Y coordinate, while having a depth value as well. As they may fall behind an opaque fragment, not all fragments can be expected to modify the final image.

(6) The GPU selects a colour for each fragment (fragment shader).

(7) The GPU merges the fragments into pixels (composition).

(8) The screen displays the image to the user (screen).

The inputs of the rendering pipeline is a series of primitives (geometry) often grouped into models and other data, such as eye coordinates, specialized vertex shaders and fragment shaders, as well as lighting information. While a more detailed

explanation is beyond the scope of this thesis, interested readers may look into any up-to-date book on the subject, for example the OpenGL Programming Guide [24].

2.2 Cluster-based rendering algorithms

All cluster-based rendering algorithms follow the same basic structure as the 3D rendering pipeline shared by all current mass-market GPUs. The difference is in where the internode communication takes place [13].

2.2.1 Sort-first

In the *Sort-first* cluster-based rendering algorithm, the communication takes place at the very beginning of the rendering pipeline. The screen is partitioned into tiles, and only a single node renders the primitives (3D elements, usually triangles) that fall within its boundaries. This necessitates an extra computation-wise and/or communication-wise expensive step to identify which node must render each primitive. This depends on whether all nodes individually calculate where each primitive falls (computation), or is calculated in a distributed fashion and shares this information with the other nodes (communication).

In the naïve case, in which each node renders a single tile of equal size, there is no assurance that each node will have similar workloads. In the worst case, the model might fall in a single tile causing a single node of the system to do all the rendering work, resulting in a great workload imbalance.

In the case when a primitive falls within multiple tiles, it must be rendered by multiple nodes. More sophisticated tile partitioning algorithms have been proposed [22],

but they produce tiles with larger perimeters and produce more primitives that fall on multiple tiles. This will also occur if there are many nodes – as the number of nodes increases, so do the number of tiles and overlapping primitives.

Because of these drawbacks, it is generally understood that the *Sort-first* approach has inherent scalability and load-balancing issues that can only be partially mitigated through heuristic techniques.

2.2.2 Sort-middle

In *Sort-middle*, the communication occurs in the middle of the pipeline. The algorithm is very close to that of *Sort-first*, except that instead of adding a step to discover which tile each primitive falls into, it employs the geometry shading component of the GPU. This was relatively easy when the graphics pipeline was done in software. However, current generation GPUs are not optimized for this operation [21]. For this reason, this approach appears to have fallen out of favour.

2.2.3 Sort-last

In *Sort-last*, the communication takes place after the 3D rendering pipeline. After each node renders its own share of the primitives, the node reads the colour and depth information for each pixels and communicates it to the system. The final image is composed of the colour of the closest fragment at each pixel location. Although this is a very expensive operation, the Binary swap, Direct send or other similar algorithms (see Section 2.3), make it somewhat manageable.

Sort-last is inherently communication-heavy as the composing of the final image

requires comparatively more bandwidth. In addition, since each node works independently during the rendering phase, each node will produce a colour for each visible pixel for its share of the primitives. As the number of nodes increases, so does the chance that this pixel will be overwritten in the final composing phase by another node's fragment. If the cost of producing this colour is expensive, as it is assumed in our thesis, a great deal of wasted computation will have occurred.

We have used *Sort-last* as the basis for the algorithms in our study because it scales better, with respect to nodes [21], than the previous two approaches. Since the behaviour of this algorithm is well understood, we employed it in our analysis.

2.2.4 Hybrid

A *Hybrid* cluster-based rendering algorithm communicates in more than one step of the rendering pipeline. An example of this approach is a specialized parallel framework [9] which, among its algorithms, includes a hybrid *Sort-first–Sort-last* algorithm. In order to keep the number of tiles manageable in large systems, multiple nodes may be used to render a single tile. The tiles are produced in a *Sort-first* fashion, but the rendering of each tile is performed in a *Sort-last* fashion. Although the scalability is better than *Sort-first*, there are still problems with load-balancing. Even if the number of nodes used to render each tile is dynamically allocated with respect to the complexity of rendering each tile, this is a very coarse-grained approach. The number of nodes used to render each tile cannot be very large, otherwise the whole system will behave similarly to *Sort-last*. It is also understood that the amount of communication in the *Sort-last* step is still quite large, even if there is a very small

number of nodes rendering each tile. In the best case, where there are two render nodes for each tile and the total number of nodes approaches infinity, the total amount of communication in the composing step will be half that of *Sort-last* (see Section 2.3).

2.2.5 Vortex

Santilli, in his doctoral thesis, Vortex: Deferred Sort Last Parallel Graphics Architecture [23], proposes a hardware solution to the cluster-based rendering problem. Each node is given a GPU with an extra shared high-bandwidth line to a hierarchical datastructure that contains the *global depth* at each pixel location. Whenever a fragment is generated, the GPUs read this global depth. If the fragment is closer to the viewer than the global depth, the depth is updated using the depth of this fragment, otherwise the fragment is discarded. Santilli proves that the quantities of communication and synchronization are manageable. Unfortunately, this was performed by emulating old hardware and additional, specialized, communication hardware. This technology does not exist in any known current hardware, and thus we cannot determine how this technology would be implementable.

2.3 Cluster-based rendering communication algorithms

There are a few common algorithms that can keep the amount of communication manageable in the composing of the final image in *Sort-last*. We shall cover two of

the more common approaches, Binary swap and Direct send. In the naïve case, where each render node sends the whole view to a single display node for processing, the communication time is given by Equation 2.1. The display node must read $n*d$ bytes of data, where n is the number of rendering nodes and d is the amount of information used to describe the whole image (*number of pixels * number of bytes to describe the colour of each pixels * number of bytes to describe the depth of each pixels*). In the equation b is the network bandwidth, and l is the network latency.

$$naive\ communication\ time(n, d, b, l) = \ n(\frac{n * d}{b} + l) \tag{2.1}$$

2.3.1 Binary swap

When dealing with more than a few nodes, the Binary swap algorithm [11] is a great improvement over the naïve communication strategy. This success is achieved by employing a divide and conquer approach with multiple rounds of communication. In the first round, each node sends half the frame's information to another node which then merges the pixels by depth in the same way as the naïve method. At this point, each node has 2 nodes' information over 1/2 the frame's data. In the next round, as long as the nodes were chosen correctly, each node sends half of their merged data to another node in the same manner as the previous round. At this point, each node has 4 nodes' merged data over 1/4 of the frame's data. This compacting of data continues until each node has full knowledge over one portion of the frame.

Figure 2.2 illustrates that the data communicated during each round is half that of the previous round. This is a geometric series:

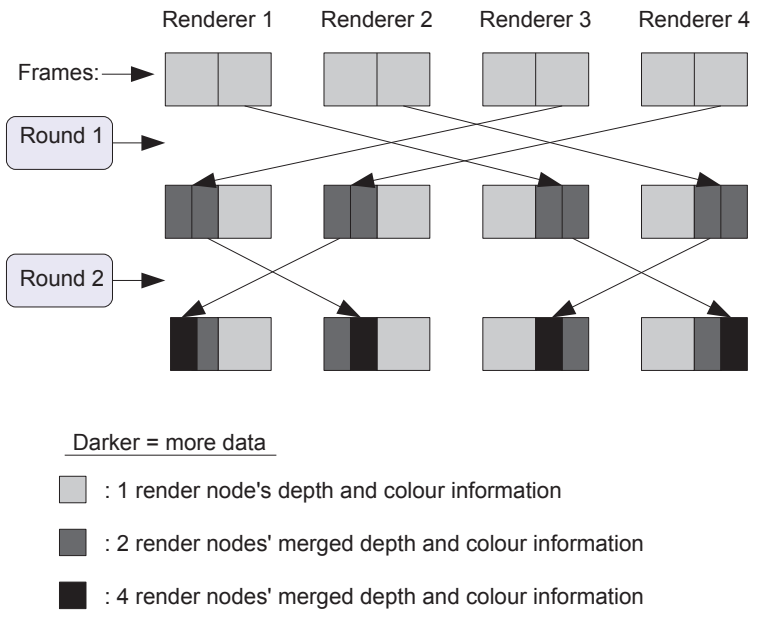

Figure 2.2: Communication Model for Binary swap with four render nodes

$$geometric\ series(n) = \sum_{i=1}^{n} \frac{1}{2^i} \tag{2.2}$$

It is well known that as, n (the number of nodes in this instance) tends towards infinity, the sum nears 1. As shown in Equation 2.2, each additional n adds less and less to the sum. This means that when dealing with more than a few nodes, the total number of nodes does not greatly affect the amount of data being sent to merge the whole frame.

When the number of nodes is a power of two, the actual communication time will be:

$$binary\ swap\ time(n, d, b, l) = (\frac{d}{b})(\frac{n-1}{n}) + l(log_2(n)) \tag{2.3}$$

where n is the number of nodes, d is the amount of information used to describe

the whole image (*number of pixels * number of bytes to describe the colour of each pixels * number of bytes to describe the depth of each pixels*), b is the bandwidth and l is the network latency.

Note that when the number of nodes is not a power of two, the number of rounds will increase, and there will be a communication round where not all nodes will need to communicate causing the network to be under-utilized. If this is the case, Direct send (described next in Section 2.3.2) might be a better choice.

Returning to the case of the number of nodes being a power of two, each node will now have $1/n$ pixels with the final correct depth and colour information. This information must then be sent to a single node for displaying the frame:

$$\textit{send to display node time}(n, c, b, l) = (\frac{c}{b}) + (n * l) \tag{2.4}$$

where c is the colour information of the frame (*number of pixels * number of bytes to describe the colour of each pixels*).

The sending of the final image to the display node also scales well with the number of nodes. The only variable influenced by the number of nodes is network latency and, in most cases, the time spent in sending the frame information will be much greater than the time lost because of network latency. This is even more the case since cluster networking hardware and networking stack have very low latency [1].

2.3.2 Direct send

In Direct send [6], the image is split into n equal sized tiles, each "owned" by a single node. When a node has finished rendering, it sends the information for each tile to

the "owner" of the tile. When a node has received the tile information from all the sending nodes, the final image is composed (merged) by depth.

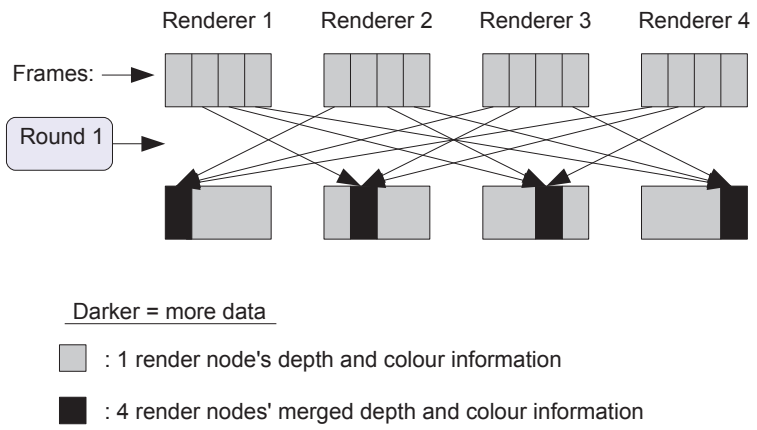

Figure 2.3: Communication Model for Direct send with four render nodes

Figure 2.3 illustrates how the data is sent. Note that all the data is sent in a single round of communication instead of $log_2(n)$ rounds of communication as in Binary swap.

Assuming bi-directional and point-to-point communication, the communication time is as follows:

$$direct\ send\ time(n, d, b, l) = (\frac{d}{b * n} + l)(n - 1) \tag{2.5}$$

where n is the number of nodes, d is the amount of information used to describe the whole image (*number of pixels * number of bytes to describe the colour of each pixels * number of bytes to describe the depth of each pixels*), b is the bandwidth and l is the network latency.

Once this communication has occurred in every node, each will have a tile that

contains its contribution to the final image. When each node sends this tile to a single display node, the image can be displayed to the user. The communication cost is the same as the one for Binary swap (Equation 2.4).

Direct send will be preferable over Binary swap when the number of nodes is not a power of 2, and when there are fewer synchronization costs (only a single round of communication). Eilemann and Pajarola introduced the concept of Direct send for *Sort-last* applications, the implication being that this algorithm has "marginally better performance than Binary swap ... [d]epending on the model size and screen-space distribution" [6]. Since the Binary swap algorithm was already coded for this work, and the benefits of using Direct send were at best minor, we employed Binary swap when sharing information between nodes.

2.4 Single GPU fragment shading optimizations

A number of algorithms can be used to minimize the redundant fragment operations when working on a single GPU. We will describe some of the more popular algorithms here. Note that some of the algorithms discussed cannot be implemented in a *Sort-last* rendering pipeline since the bandwidth between nodes is much less than the bandwidth within the GPU and the extra communication time will overly impact performance. As part of this research we have implemented some of those that can. We describe load-balancing techniques that can be used to improve performance.

2.4.1 Early-z culling

In Early-z culling, fragment shading is only performed after a depth test. Fragments which fall behind the GPU's own depth buffer (z-buffer) are discarded. To get the most out of this method, geometry should be rendered front-to-back to maximize the chance that each fragment falling in front of the current z-buffer will be in the final image. This operation is quite expensive as it may not be feasible to fully sort all the geometry. In addition extra computation may occur, due to the parallel nature of GPUs and the cost of ordering all the primitives, not all non-visible fragments may be discarded.

Another negative aspect of this algorithm is that if the fragment is discarded before it is run, then the fragment shader cannot modify the depth of its fragment, thus precluding the use of some fragment shading effects such as deflection.

2.4.2 Deferred shading

Deferred shading is similar to Early-z culling, except that the depth test is done even later in the pipeline, after the visibility of all pixels have been resolved. In the fist pass, the pipeline dumps the inputs of the fragment shader to a pixels buffer instead of running the fragment shader. Only the fragments closest to the eye are stored. In the second pass, a fragment is generated for each pixels and executed with the stored information. The amount of data stored in the pixels buffer tends to be much greater than the final pixels colour information – for example, the deferred shading configuration used in the game Starcraft II requires 32 bytes of information per pixels [7]. For a resolution of 1680x1050, the colour buffer would only require 7

megabytes (MB) of memory, while the deferred pixels buffer would require 56 MB.

Since only one fragment per pixels now remains, transparency and similar effects cannot be generated using this technique.

2.4.3 k-buffer

The k-buffer algorithm allows multiple fragments to be carried over from the first pass. This is performed through the use of k-buffers [3], which can not be implemented without artifacts in the final image due to read-modify-write hazards which can not be handled in current generation hardware. The ability to have multiple fragments stored allows for graphical effects such as transparency [3, 17]. However, this requires much more graphical memory. In the StarCraft example above, having 8 fragments stored in the k-buffer would require almost half a gigabyte of memory (452 MB).

The excessive memory requirement is not the only concern with this method. There is always the possibility of having more than k fragments at any one pixels location. It may be very difficult to determine the number of fragments at each screen position when rendering a geometrically complex scene. If a fragment is lost and would have been used for the final image, then this image will not be correct.

2.4.4 Index rendering

To reduce the amount of information required in using the k-buffer and deferred shading techniques, Index rendering may be useful. This means that, instead of storing all the inputs to the fragment shader, the index of the geometry which produced the fragment is stored [10, 17]. In the second pass, the polygons referred to in the pixels

buffer/k-buffer are rendered to produce the final image.

2.5 Cluster load-balancing techniques

To reduce load imbalance in cluster-based rendering pipelines, Nguyen and Zahorjan qualify the various scheduling policy categories which fall within two basic categories [14] as discussed below.

2.5.1 Static reassignment

In Static reassignment, work is reassigned between nodes at predetermined times. "At each reassignment time any unfinished tasks are redistributed as evenly as possible over all participating processors" [14]. Reassignment may occur either at regular intervals, or be dependent on how many tasks were to be computed during the previous reassignment, or when the network is free.

2.5.2 Dynamic reassignment

Dynamic reassignment, as opposed to Static reassignment, depends on the current state of the system, for example, when a node (or a specified number of nodes) becomes idle [14].

2.6 Coherence in graphics

One of the rendering techniques we propose involves the concept of *frame coherence* [8] as a heuristic. Frame coherence is a form of *temporal coherence* and denotes that

successive frames tend to be similar as long as the time between frames is small. If this coherence is not in place, the image may seem "jerky" or otherwise hard to follow with the eye. Using such a technique may allow for optimizations on the amount of work performed, as long as imperfect results can be tolerated.

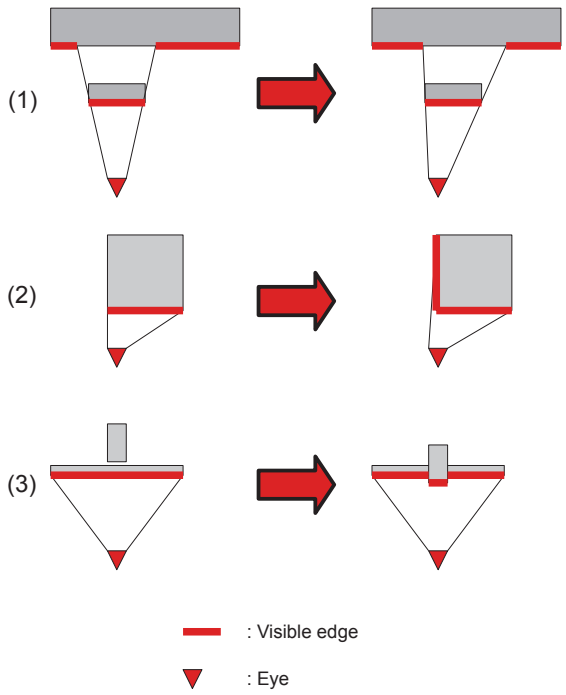

Figure 2.4: Examples of poor frame coherence

There are a few instances where frame coherence does not hold very well (see Figure 2.4). When two objects are in front of each other, even a small change in position of the viewer (parallax), or the objects, may greatly change what is visible and what is occulded (1). Object rotation, is another instance where frame coherence may not hold. Take for example, a cube where only the front face is visible, a small rotation will result in at least a whole side becoming visible (2). Another instance is

when an object moves through another (3). If the geometry of newly occluded area or the geometry of the newly visible area is complex, then the number of polygons now visible may be very different from those in the previous frame. However, even if this is the case, frame coherence can still be a useful heuristic, as long as it is assumed that there can be some change between frames, and the algorithm takes these changes into account.

Chapter 3

Parallel Rendering Algorithms

In this chapter we shall present the various algorithms that we have developed to render a scene that has both complex geometry and a complex fragment shader.

In scenes which are *geometry-bound*, i.e., where the amount of processing time spent on handling geometry is much more than that spent on handling fragment colour, *Sort-last* (Section 2.2.3) approach is perfectly suitable. The benefit of using *Sort-last* is that it can distribute the geometry equally between all the nodes. This will produce a system that is well balanced because the cost of rendering is mostly geometry based. However, this approach performs poorly when the fragment shader is expensive.

In *fragment-bound* scenes, where the amount of processing time spent on handling fragment colour is significantly greater than that spent on handling geometry, *Sort-first* (Section 2.2.1) approach would appear to be a better fit. This is because the geometry is distributed by tile in *Sort-first*. Each node has ownership of a single subsection of the screen (the tile) and renders the geometry that falls within it. At

first glance, this would appear to produce a well balanced system. However, in cases where the model does not uniformly cover the screen or when there are different fragment shaders with different costs, this algorithm may not perform well.

Let us assume that *Sort-first* performs well in fragment-bound scenes, and that *Sort-last* performs well in geometry-bound sceness. Which should we change when rendering the scene is affected strongly by both fragment shader and geometry related costs? Conversely, what if the scene is fragment-bound but performs poorly for the reasons discussed previously? In this chapter we will take a novel approach which performs well in this instance.

Since the suggested algorithms are based on the *Sort-last* paradigm, we shall begin with a more detailed explanation of this algorithm (Subsection 3.1.1). We will then discuss a simple version of the *Two-phase Rendering* algorithm in subsection (3.1.2), to be followed by an investigation into further variations that perform better in specific circumstances.

3.1 Rendering algorithms

3.1.1 Sort-last

As we have previously seen in Section 2.2, cluster rendering algorithms may be categorized as *Sort-first*, *Sort-middle*, *Sort-last* or as a combination. Since our new algorithms are based on the *Sort-last* paradigm, we shall begin with a more detailed explanation of this algorithm.

The basic *Sort-last* algorithm is performed through a number of sequential steps:

Step (1): Set view information.

Step (2): Rendering.

Step (3): Share depth and final image.

Step (4): Display final image.

In the Set view information step (1), all required rendering information (view data) generated dynamically at the view node is sent to each render node. View data includes everything required to render a scene and comes in two flavours, static and dynamic. Static view data does not change (or changes very rarely) over the execution of the program; a common example of this is model data. In contrast, dynamic view data changes often during the execution of the program; a common example of this is the location and direction of the camera, but may also include any data that must be shared by every node before rendering the frame.

In some implementations, to reduce load imbalance, dynamic view data may also include information for the redistribution of geometrical data. In the most naïve implementations, geometrical load is static, however if pruning of non-visible geometry is required or if there are multiple levels of detail, the geometrical load becomes dynamic and will introduce a load imbalance. To reduce this imbalance, the dynamic view data may include a means of redistributing geometry. To keep the redistribution data from becoming prohibitively large, instead of using all the details of the primitives, geometry is grouped into *buckets* of close members through the use of a hierarchical octree [12] and only the index of the octree node is used (please refer to Figure 3.1 for an example of octree partitioning). Note that in our implementation

we load the whole model into the memory of each node, Xiaohong Jiang et al. [9] covers a few out-of-core data management techniques but they are beyond the scope of our research.

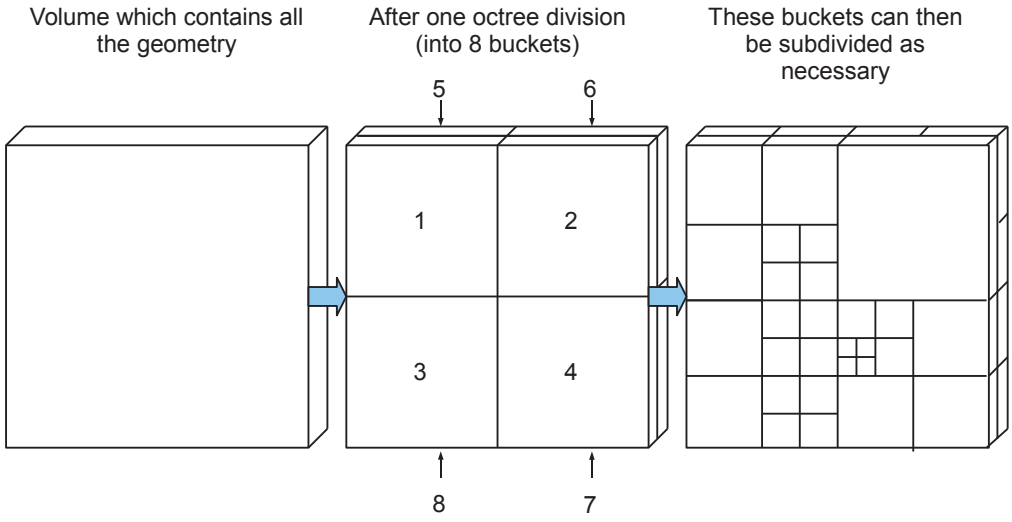

Figure 3.1: Example of buckets generated through octree partitioning

Because it can be modified by user input, the dynamic data is generated from the display node. Generated locally, it must then be broadcasted to each of the render nodes. Although the information sent is usually quite small, rendering cannot begin until the node receives this data. This will introduce delays when no productive work is being done. To keep the render nodes from lying idle during this (and during the Display final image phase), the render nodes delay the utilization of this data for a single frame. This introduces, a single frame latency, which is imperceivable if the number of frames per second displayed by the system is large enough. We have observed that, depending on the scene and amount of user interaction, a single frame delay is often unnoticeable in a 15 frames per second (fps) system.

In the Rendering step (2), each render node renders its allocated geometry using the whole 3D rendering pipeline (Section 2.1).

In step 3, Share depth and final image (3), the pixels from each node must now be composed into a final image to be viewed at the display node. Note that, since each render node generated its image independently, it cannot determine which of its pixels fall behind a pixel generated on another node. During this phase, these occluded pixels are discarded and the final image is composed using the depth information for each pixel.

We have chosen to employ the Binary swap algorithm, which distributes the communication and composing steps between nodes to improve performance. The data being communicated is fragment colour (at 24 bits per pixel in our implementation), and depth (at 16 bits per pixel). It is necessary to include the fragment depth as only the closest fragment to the eye should be visible. We have discussed the Binary swap algorithm in detail in Section 2.3.1. There exists other algorithms which include Direct send (Section 2.3.2) and have similar overheads.

The user sees the image which has been sent to the display node once the display step (4) has been completed.

Each node completely renders its view of its subset of the data allowing the system to render from a single distributed copy of the geometric data. Note that a pixel may be rendered multiple times. In the worst case, the system will render a pixel colour for each pixel in the image for every rendering node. As described in Binary swap, section 2.3.1, the number of nodes do not greatly influence the time spent in communication.

3.1.2 Two-phase Rendering (2pR)

Two-phase Rendering (*2pR*) is our first attempt to improve the performance of the *Sort-last* algorithm's poor performance when rendering scenes with expensive fragment shading. Our focus is on minimizing the amount of expensive fragment operations while minimizing network overhead. We use various algorithms to reduce redundant fragment operations on a local GPU (see section 2.4), however some of these algorithms require the use of large datastructures which would be required to be shared between all the nodes. This inter-node communication is cost prohibitive in a cluster environment.

Colour and depth information is shared between nodes in *Sort-last*. Similar to Deferred shading (Section 2.4.2), rendering is done in two passes or *phases*, Phase-1 (depth-rendering) and Phase-2 (pixel-rendering). Deferred-rendering often has a large amount of data stored at each pixel. To reduce this data, geometry is rendered twice (once in each phase).

The basic algorithm of our *2pR* technique follows:

Step (1): Set view information.

Step (2): Phase-1 rendering.

Step (3): Share depth.

Step (4): Phase-2 rendering.

Step (5): Share final image.

Step (6): Display final image.

Note that the main difference between this algorithm and *Sort-last* is that the Rendering phase (2) and Sharing phase (3) are split in this algorithm. This allows the system to have more information earlier in the pipeline, thus allowing the algorithm to reduce the number of fragment operations. We will present our reasoning after describing the *TpR* technique in more detail.

The Set view information step (1) is similar to that of the *Sort-last* step of the same name (Section 3.1.1).

In Phase-1 rendering (2), only the depth information is generated. This is achieved using a simple fragment shader that omits allocating any colour to the pixel, the pass thus takes significantly less time than the Render Image phase of *Sort-last*. The system need only read depth information from the GPU (shared in step (3)).

Since this step is geometry-bound, each node only renders the depth for the geometry allocated to it. Allocating the same amount of geometrical data to each node will often produce a well balanced system [27].

The depth generated in the Phase-1 rendering step is shared globally and merged with respect to depth (only the depth of the closest fragment is retained) in the Share depth step (3) using an efficient algorithm such as Binary swap (Section 2.3.1) or Direct send (Section 2.3.2). The depth information (usually 16 or 24 bits) and a node identifier (which is related to the number of nodes, for example 4 bits for 16 nodes) comprise the data required to be sent. After the merge, the final shared information contains the depth of the closest fragment and the node identifier that produced it for each pixel location.

The Phase-2 rendering (4) step generates the colour only for the fragments that share the same depth as the globalized depth. All other fragments are discarded.

Since this discarding is done on the GPU, the globalized depth must be written there as well. To generate the fragments, the geometry is rendered once again. For those fragments that do not fall behind the globalized depth, the full, expensive, fragment shader operations are performed.

During the Share final image step (5), each rendering node sends pixel colour information to the display node. Since the Share depth step (3) caused each node (including the display node) to know the node identifier for the node which produced the closest fragment for each pixel location, each node only needs to send those pixel colours.

By the time the system has reached the Display final image step (6), the node identifier for each pixel location produced in the Share depth step (3) and the colour information sent in the Share final image step (5) has been received by the display node. Using this information, the display node writes the final image into the framebuffer and displays it to the viewer.

There is a tradeoff in the *TpR* algorithm. To save time during fragment shading, and to minimize the amount of data communicated, geometry must be rendered twice. Also, to save communication costs, in the second sharing step a node identifier is sent along with the depth. This node identifier can be very small, depending on the number of nodes. A single byte is enough for up to 256 different rendering nodes. If we were to require 3 bytes/pixel for colour information, and 2 bytes/pixel for depth information adding a byte for node identification would increase communication costs by 20%. If this drawback is significant, less than a byte can be used for node identification, however this may increase CPU computational cycles while merging data in the two sharing phases for using unaligned data.

3.1.3 Multi-phase Rendering (MpR)

The most noticeable overhead of the *TpR* technique is that the geometry of the whole scene is rendered twice. We have tried to reduce this overhead in the *Multi-phase Rendering* technique (*MpR*). This may be a problem in scenes that have both expensive vertex (extra computation) and fragment computations (otherwise just use *Sort-last*). In our first attempt in reducing this, we used an approach similar to Santilli's Vortex (Section 2.2.5). This approach shares depth during rendering using specialized hardware to hide latency and allow communication and rendering to happen concurrently. We were obligated to take a different approach from Vortex, because current hardware [18] does not allow reading from a GPU while rendering, nor did we have access to the specialized communication hardware he described.

To emulate Santilli, we implemented a similar algorithm that works on current hardware. As in the Early-z algorithm (Section 2.4.1), to minimize the amount of geometry rendered in front of an already rendered object, rendering is done *front-to-back* keeping reordering costs reasonable. This is achieved by reordering octree "buckets" rather than each individual primitive to be rendered. Instead of constantly updating a global depth, each node updates this depth when the network is free.

Our modified implementation of Santilli's algorithm is:

Step (1): Set view information.

Step (2a): Render subset of geometry.

Step (2b): If network free, share depth.

Step (2c): If receive depth, save onto GPU.

Step (2d): If not all geometry rendered, goto 2a.

Step (3): Share final image.

Step (4): Display final image.

The Set view information step (1) is the same as previous algorithms.

Each node then renders a small subset of the work allocated to it in a front-to-back manner (step 2a). By rendering the closer geometry first, we are able to reduce the number of fragments that will be rendered fully, but not visible in the final image.

If the network is free, the current GPU depth is then shared (step 2b) using Binary swap. This is performed asynchronously so that the system can continue rendering.

If the node has received a depth map from another node, the GPU depth will then be updated (step 2c) with the new data.

Since the global depth has not been fully shared, steps (3) and (4) are the same as in the original *Sort-last* algorithm.

Although this algorithm appeared promising, our implementation was less successful than the other implementations due to our inability to employ the necessary specialized graphic hardware described by Santilli [23] and thus the cost of sharing the depth was very high. There are multiple reasons for this. There is a cost due to the GPUs used in the cluster; the Nvidia Quadro FX 4600 is unable to read the depth or update a currently used texture without stopping the whole pipeline. Also, there is no means to determine how much work is in the current pipeline.

As a result, we were only able to send small packets of work to the GPU at a time, wait for the GPU to have finished rendering, then check if the node has received data from other nodes, or the network is free and the node can send new

depth information. There is thus a tradeoff between the cost of waiting for the packet of work to be finished and the overhead of starting and stopping the rendering pipeline for each packet sent. As a result, the communication cost is relatively high (15-25% of the whole rendering time 1024x1024 resolution and 15 fps).

In summary, due to the communication costs, the depth can be shared at most 4 to 6 times and one of these is spent sharing the pixel information at the end of the *TpR* pipeline. This means that the system will have spent at least a quarter of the rendering time without having access to any kind of shared depth. Stopping the pipeline multiple times as well as reading and writing the depth information on the GPU results in a relatively expensive situation in which no rendering can be performed. This was borne out by our testing. This algorithm never performed better than both *Sort-last* (for scenes with a small fragment shader), or *TpR* (for scenes with a large fragment shader).

3.1.4 Minimal Phase-2 Rendering (mP2R)

Since the *TpR* implementation just described did not perform well in our preliminary testing, another approach is needed to reduce the amount of redundant computations in the first and second rendering phases. In the *Minimal Phase-2 Rendering* (*mP2R*) approach we will try to reduce the extra geometry calculations performed in the second rendering phase. This is the Shared Depth Algorithm described in our paper [4]. We renamed the algorithm here since we extend it to a number of different algorithms which share the depth between nodes.

By giving the geometry a colour in the first phase, we can infer the geometry that

produced each visible fragment after sharing the depth. For example, if the colour of the visible fragment were #000001 then this fragment was produced by the first geometric primitive. Unfortunately, scenes that require a cluster for rendering would usually have a very large number of primitives. There is a cost to allocating a colour to each of the primitives and we may run out of indices. For these reasons, we group our primitives in buckets, each with a unique identifier. To help cull primitives that are out of the view angle, and to increase the chance of culling primitives that fall behind another, each bucket includes primitives that are close to each other. These buckets can be created using various partitioning techniques such as k-dimensional partitioning [15], or octrees [12]. We chose to employ octree partitioning for the real-world model (Section 4.3.2.4). We believe that other partitioning algorithms would have produced similar results.

This algorithm can be seen as a parallel version of the Index Rendering algorithm 2.4.4 and has the following steps:

Step (1): Set view information.

Step (2): Phase-1 rendering.

Step (3): Share Phase-1 output.

Step (4): Phase-2 rendering.

Step (5): Share final image.

Step (6): Display final image.

The *mP2R* algorithm has the same format as the original *TpR* except for the following changes. In step 2 each primitive set (bucket) is given a unique colour to

designate its ID. The rendering differs from that of *TpR* in that it generates both depth, and the primitive set identifier. The system needs to read both the depth to share (step 3) and colour to identify which geometry to render (step 4).

To render only the visible geometry (in step 4), the node ID of each pixel location must be checked. If a fragment is visible in the final image (the node ID of the pixel location is the same as the current render node), then the bucket with the same ID of that pixel location must be rendered. Any geometry which would not generate pixels in the final image are not rendered.

Since the data is also shared with the display node in step 3, the display node can deduce which particular node renders each ID, so that only these nodes send the colour information to the display node to share the final image (step 5).

To display the image to the user (step 6), the display node iterates through every pixel in the final image. This display node then maps the colour ID to its rendering node and uses the next (the equivalent to a `pop()` operation) pixel colour sent in the previous step (5), then allocates it to the final colour of the final image pixel. When this is completed, the display node displays the final image.

The communication overhead for this algorithm is similar to *Two phase Rendering* when the index size is small, however, for larger indexes, more data must be sent. In the case of an index of 4 bytes, a colour depth of 3 bytes, and a depth of 2 bytes, the communication overhead will be 57% greater than that for *Sort-last*. Hence, this algorithm should perform best when the gains through the reduction in the time spent rendering in Phase-1 is greater than the time lost in producing and sharing the indices.

3.1.5 Minimal Phase-1 Rendering (mP1R)

Many algorithms use the idea of frame coherence (Section 2.6) to perform various optimizations. In graphics, this coherence is known as *inter-frame coherence* since each frame coincides with a specific timeframe. Inter-frame coherence makes the assumption that most frames will not differ drastically from their predecessor because neither the geometry nor the eye tend to change very much between frames. However, under certain circumstances a small change in eye position and/or geometry may greatly change the visible geometry. We use this technique to reduce the time spent in the first phase in the *Minimal Phase-1 Rendering* algorithm (*mP1R*).

In each of the previous algorithms, *total* knowledge was used to cull fragments. It is not uncommon to have algorithms that employ *fuzzy* logic or incomplete knowledge to minimize work. Since the final output must be correct (render exactly what was requested), only the first phase can use this kind of algorithm. To minimize the time spent on Phase-1, only the geometry which will most-likely be visible is rendered to produce the shared depth used in the second phase. During this second phase, the CPU (an under-utilized resource in many rendering systems) uses a bounding-box technique (an implementation is described on page 57) to choose the extra geometry that might be required to render the final image:

Step (1): Set view information.

Step (2): Phase-1 rendering.

Step (3): Share partial depth.

Step (4a): Phase-2 rendering.

Step (4b): Test bounding-box.

Step (4c): Phase-2 rendering of potentially visible buckets.

Step (5): Share final image.

Step (6): Display final image.

The first step, setting the view information, is the same as in all the previous algorithms.

Phase-1 Rendering, (step 2) renders all the geometry visible during the last frame's Phase-1 rendering in addition to the geometry in step 4b flagged as potentially visible. In our implementation, only one half of the scene is rendered during the first frame. The reasoning behind this choice will be explained after describing the algorithm in further detail. Similarly to *mP2R*, each geometry bucket is given its own colour ID.

The sharing of partial depth (step 3) is very similar to the Share depth step for the simple *TpR*, except that the depth involved is not the final depth of the scene, but only of what was rendered in the previous step.

The Phase-2 rendering step (4a) is also quite similar to that of the *mP2R* algorithm where only the geometry that produced the fragments in the shared depth are rendered.

It is necessary to identify previously un-rendered buckets that might be visible. A bounding-box test will accomplish this. First, we determine a bounding-box which completely encloses all the geometry within each bucket. Then, we transform each of these vertices on the CPU to match the view of the camera. Next, we test each of the pixels within this area to determine if there are pixels that fall in front of the

shared depth computed in step 3. If there are any potentially visible pixels for a given bucket it is flagged for rendering.

All buckets flagged as potentially visible (in step 4b) are rendered (step 4c).

In previous Share final image steps (5), the display node had the final depth of the final image by this point. This is not the case for the *mP1R* algorithm. The simplest solution would be to use the Binary swap algorithm to share both the depth and colour information. However, in most cases, the partial depth information can be used to reduce the amount of data which would have to be communicated.

As long as inter-frame coherence holds, most of the visible pixels will be the same as those rendered in the Phase-1 rendering (step 2). All the pixels that would be visible after step (4a) are sent to the display node. However, we are missing the pixels which were generated from the bucket that were flagged as potentially visible (step 4b). These pixels are sent separately. Any pixels closer to the eye than the partial depth computed (step 3) are sent directly (colour, position and depth) to the display node.

If Binary swap was used to share the final image then the Display final image step (6) follows the same process as *Sort-last*. Otherwise, the process is similar to *mP2R*. The display node iterates through each pixel of the final image, maps the colour ID produced in Phase-1 to the node that has rendered that pixel, then the display node pops the top colour value sent by that node in the Share final image step (5) and allocates it to the pixel. The display node then goes through all the *extra* pixel information sent by the nodes for each of these pixels. The display node checks if the depth is smaller than the shared depth and, if so, updates this shared depth with the new depth and updates the colour of the pixel.

The reason we have chosen to render half the scene in the Phase-2 rendering step (2) is to try to retain a similar amount of geometry rendered in each step (2). The following discusses the situation if we had chosen to render the whole geometry in the first frame and rendered a static image. In each Phase-1 rendering step (2): the first frame would render the whole scene, the second frame would only render the visible geometry, the third frame would render the visible geometry and all the geometry which failed the bounding-box test in the previous frame, the fourth frame would render the same geometry as the second frame, and so on. In other words, step 2 for even frames will be shorter (possibly much shorter, depending on the scene) than step 2 for odd frames. By rendering half the geometry in the first frame, this effect will, on average, be much less pronounced.

The assumption that the visible geometry between two consecutive frames will be similar (or inter-frame coherence) will not always hold. Sometimes even a small change in object location and/or eye location can change what geometry is visible and the number of pixels that each piece of geometry will render (see Section 2.6 for more detail).

This algorithm may also cause stuttering, as the time spent on frames will increase greatly (both in rendering times and in communication time) when the image changes drastically. Communication in this situation is similar to *mP2R*, except for the overhead of communicating the pixels produced by the elements that fail the bounding-box test (step 4b). In our tests, we found that this extra communication overhead was not overly cumbersome.

3.2 Load-balancing

3.2.1 Inter-frame load-balancing

Inter-frame load-balancing involves rebalancing the workload between frames. Similar to *mP1R* (Section 3.1.5), these algorithms assume that inter-frame coherence holds for most frames.

Since the computational cost is mostly generated through pixel colour shading, a small change in the number of pixels generated by each node will produce a large difference in the load balance of the system. The number of visible pixels rendered in the final image is not easily approximated without rendering the scene. Therefore, it may not be possible to balance the system without performing some intra-frame load-balancing.

3.2.1.1 Per-frame load-balancing

To better balance the system, per-frame load-balancing (load-balancing over the whole frame) is a common load-balancing technique [8]. This approach uses the concept of frame coherence, and this assumes that the behaviour of the current frame will be similar to the previous one. This allows the system to balance a certain amount of work. The technique has several drawbacks. There is no assurance that the views between frames will be similar, since new objects may come into the view or an object that was visible in the previous frame may be occluded by another object. It is quite inexpensive to perceive when an object/bucket will enter or exit the view (which can be approximated by taking the bounding-box of the bucket and seeing if it falls within the view frustum). Occlusion on the other hand, is completely different. Even if each

node has a similar number of primitives, occlusion may force the system to produce a vastly different number of pixels in each node and drastically increase load imbalance.

3.2.2 Intra-frame load-balancing

When the final image differs enough between frames, Per-frame load-balancing will not perform adequate load-balancing. The number of pixels each node will render in Phase-2 will have changed. Since we have been assuming that the main computational cost is in the fragment shader, the change in the number of pixels each node renders in Phase-2 means each node will have a different workload than the previous frame, even when Per-frame load-balancing has been performed.

In this situation, we may wish to redistribute work within the generation of a frame. There are two basic approaches to do this, the first involves using the information employed in Phase-1 to balance the work in Phase-2. The other involves rebalancing work during Phase-1 and/or Phase-2.

3.2.3 Phase-2 load-balancing

When performing *mP1R* (Section 3.1.4) or *mP2R* (Section 3.1.5), after the first "Share" Step, each node can identify what geometry is visible, and how many pixels each "bucket" of geometry covers in the final image. Using this information, we have devised four different methods of performing intra-frame load-balancing.

3.2.3.1 Naïve redistribution

The simplest, most naïve solution, would not involve redistributing the "buckets" between different nodes. Instead, during the Phase-2 rendering phase, each render node would simply render the visible geometry that had already been distributed in Phase-1. This will reduce the amount of redundant fragment shading work over the whole system, but may result in poor load balance.

It does not take very long to redistribute the geometry in each of the following redistribution techniques described, but there may be some outlying cases where this technique might be used. A few possible reasons might include: the geometry being cached on the GPU, the geometry being only available to a single node, or there being too many "buckets" to redistribute in a given amount of time.

3.2.3.2 Redistribution by bucket

The Naïve redistribution technique will often unevenly distribute the amount of geometry. By identifying which "buckets" are visible and distributing these "buckets" evenly between the nodes, each node could be made to render a similar amount of geometry in Phase-2.

3.2.3.3 Redistribution by pixel

Although, the Redistribution by bucket technique distributes geometry evenly (assuming each "bucket" contains the same amount of geometry), each render node might have to render a different number of fragments. The following algorithm makes a *best effort* to distribute with regards to the number of pixels rendered per node.

The redistribution by pixel algorithm requires two sorted lists, a list of "buckets" sorted in descending order with regards to the number of pixels rendered in the final image (bucket-list), and a list of nodes sorted in ascending order with regards to the number of pixels that it will be required to render (node-list). The "bucket" with the largest number of pixels is allocated to the node that has the least number of pixels allocated to it, until each "bucket" is allocated. This will optimally distribute the number of pixels per node, but does not take the amount of geometry each node must render into account.

Since the bucket-list does not require updating, it can be sorted by any sorting algorithm (we used `quicksort()`). However, the node-list must be updated very frequently. We considered two implementations, the first would use a heap so that the node with the fewest pixels would always be available, and the second would use insertion sort.

The heap datastructure is an efficient datastructure where the largest (or smallest) element is always easily available, updates to the datastructure (insertion and removal) are also very quick, often taking order $log(n)$ operation (where n is the number of nodes). Insertion sort, when dealing with an array with only one out-of-order element, will move that element until it occupies a position where the previous element is smaller, and the next element is larger or equal. In the worst case, this will require traversing the whole array once, taking order n operations. Insertion sort is not a very good choice for unsorted arrays, but is very quick when working on lists where there are few unsorted elements. Our implementation used insertion sort. Insertion sort is very easily implemented. Both datastructures should perform similarly for this use when there are a small number of nodes.

Both insertion sort and heaps are well understood algorithms discussed in any introductory book or class on datastructures and algorithms. Hence, explanations of either algorithm in further detail is unnecessary.

3.2.3.4 Round-robin redistribution

Neither Redistribution by Bucket nor Redistribution by Pixel techniques take into account both geometric load and fragment load when distributing the "buckets." The Round-robin algorithm is similar to the Redistribution by bucket using a sorted bucket-list. This bucket-list is generated by sorting the "buckets" in descending order with regards to the number of pixels rendered in the final image. This is the algorithm which we had described in our paper [4].

Each node is allocated a "bucket" in a round-robin fashion, each node should be required to render nearly the same number of pixels, and will optimally distribute the number of "buckets" between nodes (as long as each "bucket" contains the same amount of geometry). Implementation details of this algorithm can be found in section 3. This algorithm was used in the testing of the *mP1R* and *mP2R* shown in the next chapter due to it distributing work well when rendering scenes with large amounts of geometry and with expensive fragment shaders.

3.2.4 Dynamic reassignment

Unlike the previous intra-frame redistribution techniques, Dynamic reassignment can be implemented during any rendering step. Nevertheless, and for the same reason as for *TpR* (Section 3.1.3), there are performance hurdles imposed by the current generation GPUs. For example, there is no way of asking the GPU how much work is

left to be rendered. Currently, the only way of knowing how much work is left in the rendering pipeline is to `glFinish()` to wait for the pipeline to finish. To keep latency down, only a small amount of data can be sent to the GPU for each flush command.

In our implementation used in preliminary testing, each node would try to render $(x-y)$ "buckets", where x is some predefined number of buckets, and y is the number of buckets which would take longer to render than the time spent redistributing. When a render node completes this work, it would start rendering the next octree bucket and send a request for work redistribution. The cost of this redistribution is partly mitigated by still having at least y buckets of work in each node while redistributing. In each test case, we discovered in preliminary testing that either the latency was too large for much of an effect (for larger amounts of work per `glFinish()` command), or that the cost of flushing the GPUs buffer is too expensive (for smaller amounts of work per flush command).

3.3 Summary of contributions

We have introduced an algorithm which splits the rendering pipeline into multiple phases, *TpR*. This algorithm reduces the redundant fragment operations and improves performance in scenes with costly fragment shaders. However, it introduces some computation and communication overheads. We, therefore, introduce a number of algorithms which tackle the computation overheads by reducing the second phase (*mP2R*), and by reducing the first phase through the use of *temporal coherence* (*mP1R*). Since the reduction in redundant operations may not be uniform between nodes, we also introduce a number of load-balancing algorithms to redistribute the

work.

A discussion of the implementation and performance of each of these algorithms can be found in the next two Chapters.

Chapter 4

Implementation

4.1 Goals

Most benchmarks for the performance of rendering algorithms focus on the number of frames per second (fps) that the algorithm can sustain with a set problem size. In our opinion, this is not the key factor. A better approach is to ask how complex can we make a scene and still retain a smooth viewing experience. This is doubly true when rendering on a cluster environment.

A scene being rendered on a cluster, is an indication that this scene is too complex for a single machine to handle. Which ever algorithm used, there will always be a large communication overhead. The law of diminishing returns (and the well known Amdahl's law in parallelization [20]) states that, as we increase the number of parallel resources, this overhead will take a larger and larger portion of the total execution time. In the ideal case, the overhead never increases with the increase in compute resources and the parallel work is perfectly distributed. Even with an infinite number

of computers the best time will be equal to this overhead.

Implementing an algorithm more complex than *Sort-last* implies the need for extra overhead. If the problem size stays the same, then this extra overhead may undermine the benefit of that using this algorithm, but might allow for a much more complex problem while staying within a set time limit.

Therefore, we have chosen to test each algorithm's performance with respect to (1) the maximum amount of geometry and (2) the maximum complexity of fragment shader that can be used to render a specific scene in an average of $1/15^{\text{th}}$ of a second. This allows us to test the maximum complexity of the geometry and fragment shading for a given scene while keeping a relatively smooth 15 fps. Our focus relates to the use of an expensive fragment shader. However, understanding the behaviour of the system under non-optimal conditions (large amounts of geometry) is also of some importance.

4.2 Environment

We ran our code on Concordia University's *Cirrus* Cluster [5] which has 32 graphics nodes. The cluster is now nearing its *end-of-life* and only 26 out of the 32 render nodes were available during the period of our research. Each node consists of two dual core 2.2 Ghz AMD Opteron processors, a single Nvidia Quadro FX 4600 graphics card, from 8 to 16 gigabytes of memory, and an InfiniBand$^{\text{TM}}$ 4x SDR (8 gigabits per second) network interface.

When the processors were released in 2005 [2], they were the fastest Opteron processors available.

The system's GPU, the Nvidia FX 4600 [16], is based on the G80 chip, the same as the Nvidia 8800 GTX desktop chip. However, it has more memory (768 MB as opposed to 500 MB) and the driver emphasizes quality of image rather than frame rate. The FX 4600 processor includes version 4.0 of the Vertex and Pixel Programability Shader Model.

Although Cirrus' website [5] states that there are between 8-24 gigabytes of memory in the visualization nodes, however when we run an "`sinfo -Nle`" command on the cluster, the largest amount of memory in any visualization node is only 16 gigabytes. Since there are a very limited number of nodes that have graphics cards, our testing is done on a heterogeneous cluster with respect to memory. As it turns out, none of the examples require more than 8 gigabytes of memory, so this imbalance is expected to have little impact on our results.

As mentioned earlier, the nodes are connected through an InfiniBand^{TM} network [1] with 4x SDR interfaces. This means that the theoretical peak bandwidth is 8 gigabits per second (or 8 times gigabit Ethernet). InfiniBand^{TM} is often used in cluster environments and has very low latencies with respect to Ethernet. It's switches and routers have point-to-point serial connections [19], meaning that, as long as each node resides on the same switch, communication between nodes will not be affected. While looking at the performance of our implementation this would appear to bear this out.

4.3 Implementation

The code is written in C++ with the MPI, GLUT and GLSL libraries. We were obliged to use a variety of libraries and compilers during the course of our research. The implementation used in our testing included the x86-open64 compiler and HP's implementation of MPI. In the early stages of this study, we compiled the code with the Pathscale compiler and the multi-threaded version of MPI. After some system updates, the multi-threaded version of MPI no longer worked, so not all the methods described in the previous chapter could be tested.

In interest of simplicity we chose to split up the nodes into a single display node with multiple render nodes. The render nodes render their portion of the scene and send the results to the display node which merges and displays the pixels received, as well as reacts to user input.

Currently, the system's state information is kept in a single `struct` and is sent to each render node. This struct is quite small, and is sent asynchronously to each of the render nodes. To reduce the impacts, the amount of data being sent is kept small and sent asynchronously to keep communication costs low and a single frame latency is introduced to mitigate the synchronization cost between the display and render nodes.

We attempted to test each of the algorithms described in the previous chapter. However, we abandoned the Multi-phase technique early because there were a number of technical and performance barriers which could not be avoided, as discussed in the following subsection.

4.3.1 Barriers in Multi-phase Rendering (MpR) implementation

For the *MpR* technique to perform well (Section 3.1.3), the depth information must be updated often. There were two reasons this could not be accomplished. The GPU driver does not allow the fragment shader to read and write to the same texture, resulting in extra overhead in the fragment shader as the system cannot use an up to date depth while rendering. In addition, it is not yet possible to both read and write textures without stopping execution. Since it takes roughly 5% of the allocated time to read and write to the depth texture, updating the depth multiple times quickly becomes too expensive. Sharing the depth information is also expensive. Even if rendering the scene were to be done in parallel to the depth information communication, having a latency of 10-25% of the time spent rendering the whole frame means the number of times that this depth can be shared cannot occur often.

During our testing, no matter the number of times the depth texture was updated, the multi-phase algorithm performed worse than some or all of the other algorithms. When fragment shader costs are low the *Sort-last* approach performs much better than multi-phase, and when fragment shader costs are high, the overhead costs and the inability to discard all non-visible fragments made multi-phase perform very poorly.

We considered reducing the resolution of the depth texture. However, to obtain a correct final image, the lower resolution depth texture can only include the furthest depth within the area covered in the higher resolution, "real" depth, reducing the number of fragments that can be discarded. Also, to get the correct final image, the full resolution depth must still be read and communicated.

These obstacles forced us to abandon development of this algorithm. It is not currently in a form that would make it easy to run thorough tests.

4.3.2 Structure

The implementation of the testing program consists of a GLUT event handler, Renderer interface and implementations, Order interface and implementations, Model interface and implementations, a testing module as well as a suite of utilities related to MPI communication, profiling, camera controls, and GLSL shader handling.

The code is too voluminous, so we have not chosen to include it. Many of the algorithms are described in detail in previous chapters. We shall, however, discuss a few of our design choices relating to the rendering, ordering, and GLSL modules.

4.3.2.1 Rendering module

The rendering module consists of 18 steps. Each step is contained within a code-block, the time spent in each code-block is noted and saved to a file after each test. The first number shown within the parentheses describes the phase, while the second are for steps which occur more than once in a phase (during *mP1R*, 3.1.4). Note that all the second phase steps are skipped in the *Sort-last* implementation. The module called in each step is written within parentheses.

Init(1) - initializes, and sets the fragment shader for this phase (GLSL Module).

Order(1) - determines what the render node will render in this phase (Order Module).

GPURender(1) - renders the scene.

GPUEnd(1) - calls `glFinish()` to clear the GPU buffer.

GPURead(1) - reads data from the GPU. The data includes depth and any other required information.

Synchronize(1) - calls an MPI_Barrier over all the render nodes. Although this step is not required since sharing the depth imposes synchronization, it is necessary to accurately calculate the system's imbalance. Otherwise, the time spent synchronizing would be combined with the communication step.

Share(1) - shares/communicates the data read from the GPU (Binary swap).

Init(2) - initializes, and sets the fragment shader for the second phase (GLSL Module).

GPUWrite(2) - writes the depth to the GPU.

Order(2,1) - determines what the render node will render in the second phase (Order Module).

GPURender(2,1) - renders the scene (Model Module).

Order(2,2) - determines if any geometry not rendered in GPURender(1) might be visible in the final image (Order Module).

GPURender(2,2) - renders the elements of the scene which were determined to be potentially visible in the previous step.

GPUend(2) - calls `glFinish()` to clear the GPU buffer.

GPURead(2) - reads data from the GPU. The data includes pixel colour as well as depth information, if required.

Share(2) - sends the pixel colour information to the display node.

Share(2)Update - if any pixels are closer than the depth computed in Phase-1 then the pixel colour and the depth of each of these pixels are sent to the display node. This can occur if GPURender(2,2) has rendered any geometry.

Cleanup & Merge - cleans up any dynamically allocated memory. For Order methods that use inter-frame coherence, data is saved for the next render call.

4.3.2.2 GLSL module

As specified earlier, the fragment shaders should be kept as simple as possible in the first phase. Most GLSL textbooks have an example of the simplest vertex and fragment shaders, so they will not be described here. However, there does not appear to be a common fragment shader with enough complexity to require culling.

We were unable to discover a fragment shader that was available for download and included all the necessary elements, or complex enough to require the use of a cluster. The ability to vary the execution time of this shader is especially important for testing of the scalability and performance of the algorithms. We need to be able to gauge how complex a fragment shader can be and still have a real-time frame rate (15 fps). Therefore we used a dummy loop to emulate a fragment shader in which the execution time varies as we change the number of iterations of this loop. The code for this fragment shader follows:

Code 1 GLSL code for the Phase-2 fragment shader

```
uniform int slowfront; //Slow down if the fragment is visible?
uniform int slowback;  //Slow down if the fragment is NOT visible?
uniform int speed;     //the size of the slowdown loop
uniform sampler2D depth_tex; //depth texture produced in Phase-1
uniform vec2 winsize;  //the size of the window
uniform float epsilon; //there seems to be some kind of rounding error
                       //so any fragment "epsilon" away from the depth
                       //value is chosen to be visible

void slow(int i)
{
  //Loop to increase fragment "cost"
  int m;
  for(m=0; m<i+1; m+=2)
    m--;
}

void main()
{
  vec3 n;
  vec4 color;
  vec4 pos = gl_FragCoord;

  float d = texture2D(depth_tex, gl_FragCoord.xy / winsize).r;

  if(d<=(pos.z-epsilon))
    {
      if(slowback == 1) //NOT VISIBLE
        slow(speed);
      gl_FragColor = vec4(1,0,0,1);
    }
  else
    {
      if(slowfront == 1) //VISIBLE
        slow(speed);
      gl_FragColor = vec4(0,1,0,1);
    }
}
```

In this sample code, non-visible fragments are not discarded, but are given a different colour (red) so that they are visible in the final image. This allows us to visually inspect if any of the algorithms discarded any visible fragments.

The sample code's loop is not a simple loop because, when the contents of the loop are left blank, no slowdown occurs because of some loop optimization. There seems to be a large slowdown after the loop becomes very large ($\sim 19,000$). Also, this may be due to GLSL unrolling the loop and ballooning the size of the fragment shader, producing a lack of available instruction memory on the GPU.

We refer to the loop size as the fragment shader "cost". This is somewhat of a misnomer as the loop is not the only computation related to the fragment shader. However, when the loop size is sufficiently large, the fragment shader does spend most of its time in the loop. In the fragment-bound performance section (5.3) this seems to be the case.

Our *Sort-last* fragment shader is very similar to the previous code. Since there is no shared depth, there is no need to test the depth, and the `slow(int i)` loop must be called for every fragment.

4.3.2.3 Order module

Other than splitting the rendering pipeline into two phases, the main differentiating aspect of all the algorithms is in how they distribute the work.

The distribution method used by the *Sort-last*, *TpR*, and *mP2R* algorithms is very simple. The code for the *mP1R* algorithm differs and is described further in Code 4.

Code 2 Order(1) algorithm for all algorithms except mP1R

```
num_r //number of render nodes
n     //current render node index (range [0 .. num_r-1])

bucket[]          //bucket (or sphere) to be rendered
num_buckets       //number of buckets (or spheres)

//render node 'n' must render
for(i=n;i<num_buckets;i+=num_r)
   Render(bucket[i])
```

In our example (Code 2, above), the geometric load will be well distributed as long as the amount of geometry in each bucket is uniform. An added benefit is that this algorithm takes very little time to perform.

Code 3 mP2R and mP1R Order(2,1) algorithm (Round-robin)

```
depth[] //depth information at every pixel location (after Binary swap)
index[] //index of bucket which produced that pixel (after Binary swap)

bucket[].index //index of each bucket
bucket[].count //number of pixels each bucket generated

num_r       //number of render nodes
n           //current render node index (range [0 .. num_r-1])

num_buckets //number of buckets (or spheres)

for(i=0;i<num_buckets;i++)
   bucket[i].index = i;

for(i=0;i<image_size;i++)
   if depth[i] > 0 //scene may not cover every pixel location
       bucket[i].count++;

Sort() buckets by number of pixels generated //We used quicksort

//Set what each node must render
for(i=n;i<num_buckets;i+=num_r)
{
   if (bucket[i].count == 0)
      break;
   Render(bucket[i].index)
}
```

This is an implementation of the Round-robin distribution algorithm described in Section 3.2.3.4. This algorithm goes through each pixels location of the frame shared in Phase-1, and counts the number of pixels that each bucket produced. The buckets are then ordered by the number of pixels each bucket produces. Each render node renders a bucket in a round-robin fashion, i.e., the first node renders the bucket with the highest number of visible pixels, the second node with the second most, and so

on.

While testing this algorithm, the first node must render slightly more pixels, but the actual number of pixels rendered by each node are quite similar. geometrys is distributed as well as possible, as long as the amount of geometry within each bucket is uniform.

As noted above, the *mP1R* algorithm's order modules are slightly more complicated than those of the other algorithms.

Code 4 mP1R Order(1) algorithm

```
num_r //number of render nodes
n     //current render node index (range [0 .. num_r-1])

bucket[]     //bucket (or sphere) to be rendered
num_buckets  //number of buckets (or spheres)

r_last_frame[] //for every bucket during the last frame:
               //=1 if generated a pixel in Phase-1
               //=1 possibly visible (Order(2,2), bounding-box test)
               //=0 otherwise


if(this is the first frame) //render only half the scene
   for(i=n;i<num_buckets;i++)
      r_last_frame[i] = i%2

//Set what each node must render
for(i=n;i<num_buckets;i+=num_r)
   if r_last_frame[i] == 1
      Render(bucket[i])
}
```

This algorithm's code (Code 4, above) distributes half the scene for the first frame, and from then on only renders the geometry that was either: visible in the last Phase-1

Rendering phase, or potentially visible in the final image.

The *mP1R* algorithm Order(2,1) is the same as for *mP2R* Order(2,1) (Code 3).

Code 5 mP1R bounding-box test

```
num_r //number of render node
n     //current render node index (range [0 .. num_r-1])

depth[][]      //current depth at pixel location [x][y]

bucket[]       //bucket (or sphere) to be rendered
bucket[].box[] //the 8 points which enclose all the geometry
               //within the bucket (x,y,z)

num_buckets    //number of buckets (or spheres)

r_last_frame[] //same values as in Previous Order(1) algorithm

BoundingBoxTest(int i)
{
   //check the bounding-box - is it possibly visible?
   gluProject: each of the bucket[i].box.[x,y,z] vertecies into
               pixel locations

   for (x = min(bucket[i].box.x); x < max(bucket[i].box.x); x++)
      for (y = min(bucket[i].box.y); y < max(bucket[i].box.y); y++)
         if min(bucket[i].box.z) < depth[x][y]
             return TRUE
   return FALSE
}

//Set what each node must render
for(i=n;i<num_buckets;i+=num_r)
   if r_last_frame[i] == 1 //was not dealt with in Phase-1
      if BoundingBoxTest(bucket[i]) == TRUE //might be visible
          Render(bucket[i])
```

Computing the bounding-box of any bucket is straight-forward. It is the smallest box which completely encloses all the geometry within a bucket. Our `BoundingBoxTest()`

(Code 5, above) is very simple. In essence, it draws a rectangle at the depth of the closest vertex on the screen which encloses all the geometry of a bucket and checks the depth of each of the pixels within that rectangle. If any pixels is closer to the eye than the closest vertex of the bounding-box then the bucket is deemed to be potentially visible and must be fully rendered. Although more complicated algorithms might require fewer pixels to be tested, ours is adequate to illustrate the concept.

As we know, half the scene is rendered when rendering the first frame. The implications are important, the bounding-box test will often declare that a bucket might be potentially visible when it is not. For example, when rendering a static image, if all the geometry is rendered in the first frame, then during the second frame only the r_last_frame array elements with value 1 will be visible within the final image. This sounds fine, but during the third frame the r_last_frame array elements with value 1 will be those visible within the final image, i.e., those that failed the bounding-box test! The time spent rendering each even number frame will differ greatly from the odd number frames because of the difference between the rendering time spent in Phase-1 and the bounding-box test time. In general, if only half the geometry in the first frame is rendered in Phase-1, this problem is greatly mitigated.

4.3.2.4 Model module

We tested two very different models. The first is a synthetic model, where the main focus was the ability to test performance by increasing and decreasing the amount of geometry within the model. Second, a large realistic 3D scene, used to illustrate the effect of the algorithms in real applications.

The Sphere model is a synthetic model (fig. 4.1) composed of 5000 spheres con-

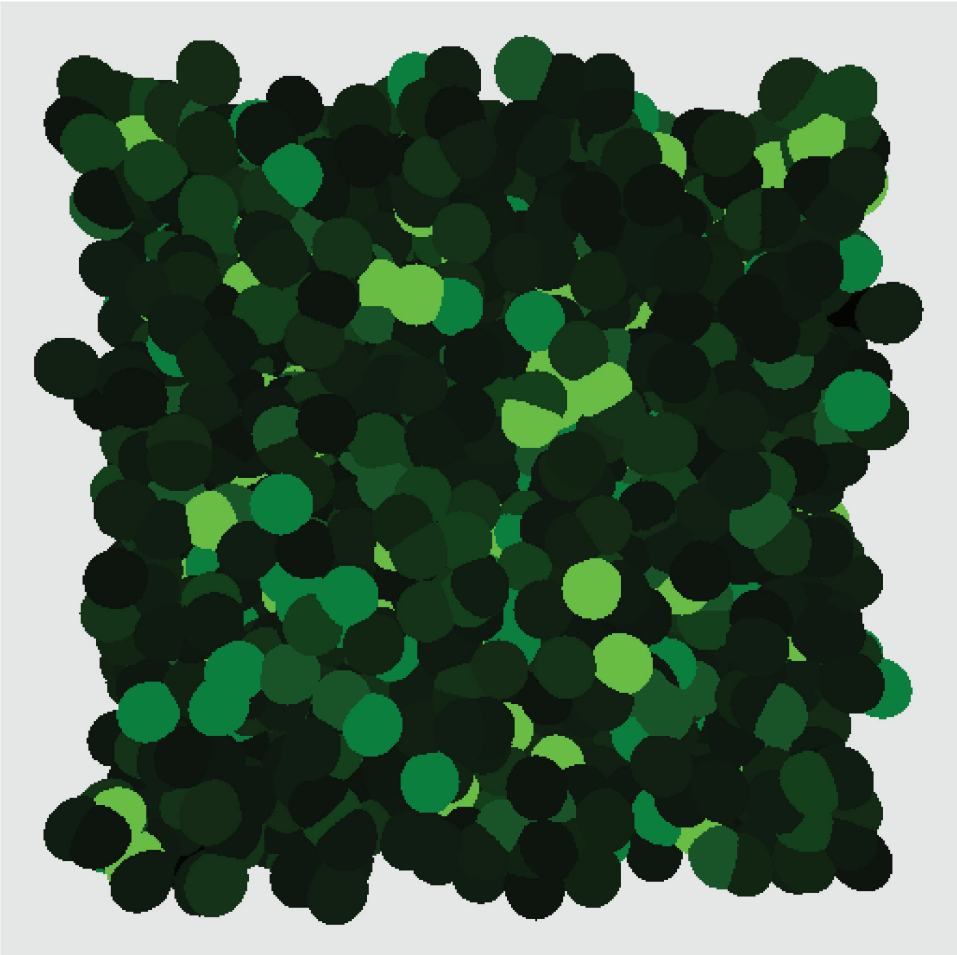

Figure 4.1: Sphere model

tained within a cube volume. The spheres are produced by calling the `gluSphere()` function. We use a very simple sphere (a cube) for the fragment-bound performance test (Section 5.1 and 5.3). In the geometry-bound performance test (Section 5.1 and 5.2) the geometric complexity can be modified by increasing or decreasing the parameters used to tessellate each sphere.

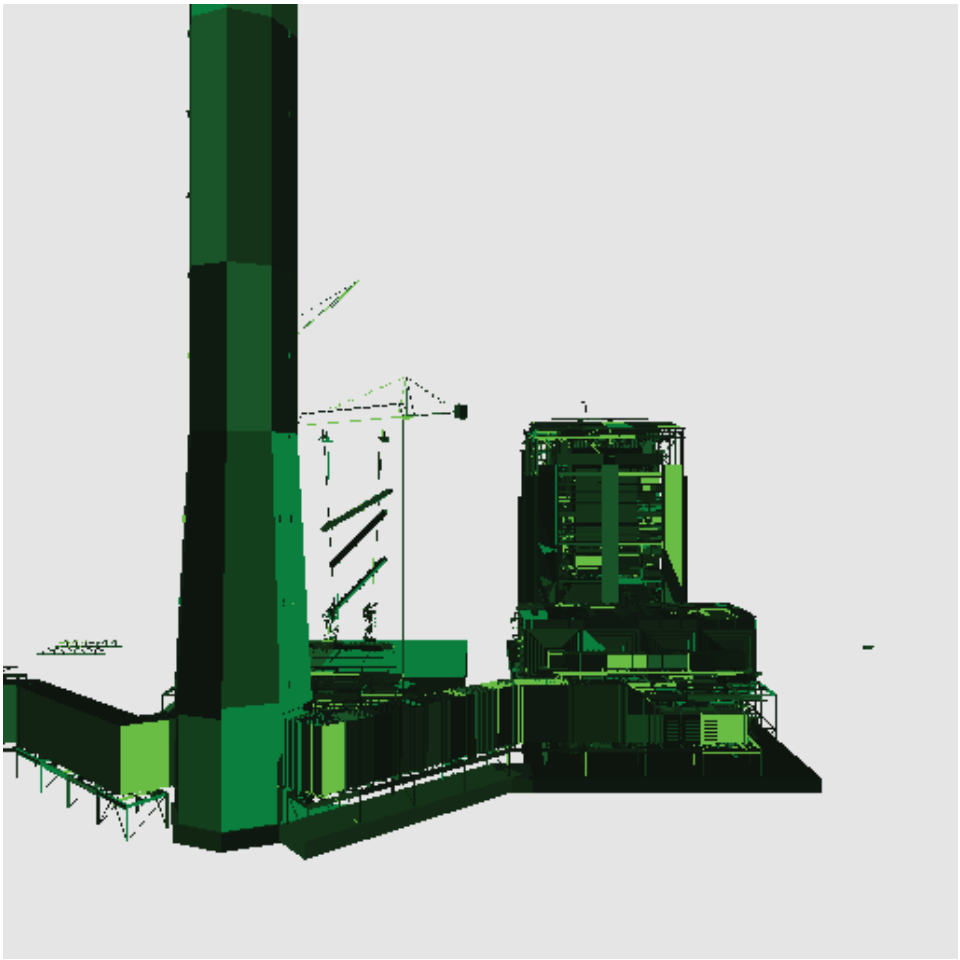

Figure 4.2: UNC Power Plant model

The University of North Carolina's (UNC) Walkthu Project's Power Plant model (fig. 4.2) [26], consisting of 12,748,510 triangles was our choice for the realistic 3D

scene. We used a `GL_VERTEX_ARRAY` and `glDrawElements()` to render this model. Without these, rendering at 15 fps cannot be performed using the *TpR* algorithm on a single node. For indexing purposes, the model was partitioned using octree algorithm, a common algorithm that can be found in most graphics textbooks. With this model, we were unable to include the *mP1R* algorithm in our tests because we had made an earlier decision to improve performance by not breaking up nor duplicating triangles which fell into multiple octree nodes. The presence of many large triangles meant that the bounding-box tests could not be performed without large overheads.

There are very few large models readily available to the public or for research purposes. We are greatly indebted to UNC for making their models available (the Power Plant model was used in this project). Most of the large models that are available are polygon meshes that tend to have very little occluded geometry. The Sphere model began as a number of *Stanford Bunnies* [25], however, it does not have many levels of detail, and it would be difficult to modify the amount of geometry being rendered without having to increase the number of bunnies. However, for the gemetry-bound tests, we wanted to be able to gauge the amount of geometry each algorithm could render in real-time while keeping the rest of the model as similar as possible. Changing the level of detail of a set number of spheres covers both of these requirements.

4.3.2.5 Binary swap

The Binary swap algorithm was described in detail within the 2.3.1 Section. We are only mentioning it here since it is integral in the communication model of our system.

4.3.2.6 Testing module

Testing does not require user input. The render node starts with the smallest amount of geometry and fragment shader costs. Depending on what is being tested, either the fragment shader cost is increased (fragment-bound) or the number of subdivisions used to generate the spheres (geometry-bound) is increased until the desired frame rate is observed.

Testing using a static image provides little useful information because it is very easy to improve load-balancing and rendering performance when there is perfect frame coherence. To approximate real-world usage, the model is placed in the middle of the screen and the camera rotates around it. We chose to have the camera to move in 5 degree increments over a complete circular trajectory. Thus each test is performed over 72 frames ($^{360}/_{5}$). At 15 frames per second, this should take 4.8 seconds.

If the test takes less than 4.8 seconds, either the fragment shader cost is increased, or the number of subdivisions in the glutSphere() used in the Sphere model is increased and the test is run again. This is repeated until the average frame rate is less than 15 fps (i.e., longer than 4.8 seconds).

These tests are performed using different number of render nodes. Since the Binary swap algorithm used to share the depth performs best when the render nodes are a power of 2, we tested each algorithm with 1, 2, 4, 8 and 16 render nodes. Each test was performed with a resolution of 1024x1024.

The test results are discussed in the next chapter.

Chapter 5

Experiments and Results

In this chapter we will describe the performance of the various algorithms, beginning with the overall performance of each of the algorithms rendering the Sphere model, when the rendering is either geometry-bound (many triangles), or fragment-bound (expensive fragment shader). Next, we shall go into greater depth into exactly where each algorithm is spending its time. This will help illustrate how each algorithm behaves, and the kind of problems that might be best suited for each algorithm, as well as how each algorithm might behave on a larger number of nodes. Finally, we shall run the fragment-bound test on UNC's Power Plant, a realistic model.

The test scene will have a great influence over the performance of each algorithm, especially since the performance gains of the algorithms we are using is due to the culling of fragments (and geometry for the *mP1R* algorithm). The amount of work that can be culled is an important factor when determining performance. Although the Sphere model is completely synthetic, and rendering this scene may not reflect the performance in the "real-world", the performance on a realistic scene will be analyzed

in Section 5.4.

5.1 Overall performance

When the Sphere model is rendered using an expensive fragment shader at 15 fps (fig. 5.1), of all the algorithms tested *Sort-last* performs the worst. In each case, the *mP2R* algorithm performs better than *mP1R*, which performs better than *TpR* which performs better than *Sort-last*. Except when rendering with only a single render node, when *mP1R* performs better than *mP2R*.

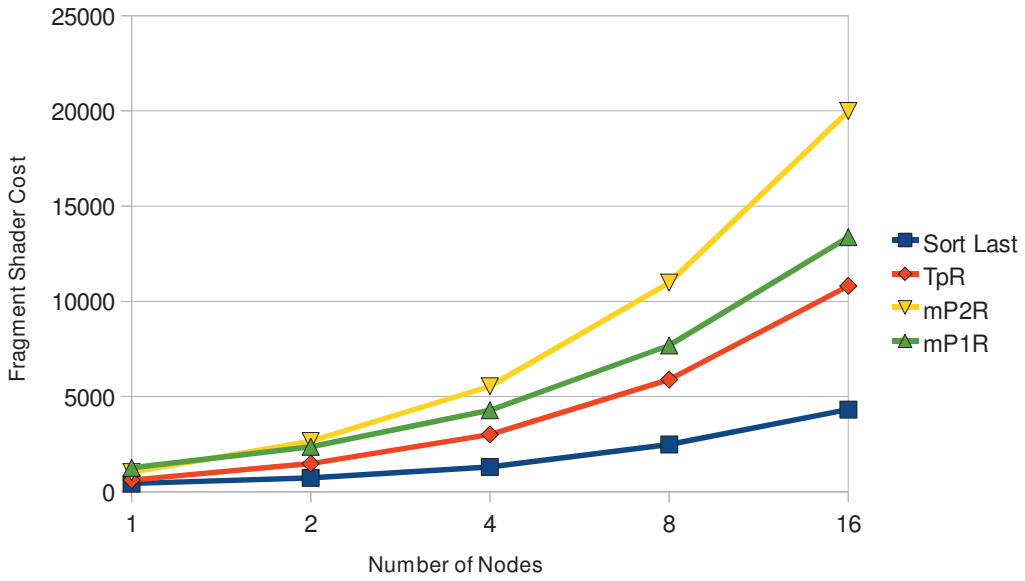

Figure 5.1: Fragment-bound performance (at 15 fps)

The relative performance of each algorithm is roughly what was expected when rendering a scene where a large number of fragments can be culled. Since it can be assumed that a large portion of the execution time is spent in the fragment shader (see

fig. 5.10 for an example), each algorithm performs better than *Sort-last*. However, the time saved by not having to render the geometry twice allowed both *mP2R* and *mP1R* to perform better than *TpR*. *mP1R* performed better than expected, we had felt that the bounding-box test and not having an exact depth after Phase-1 (extra fragment operations and network usage) would make the *mP1R* algorithm perform worse in testing.

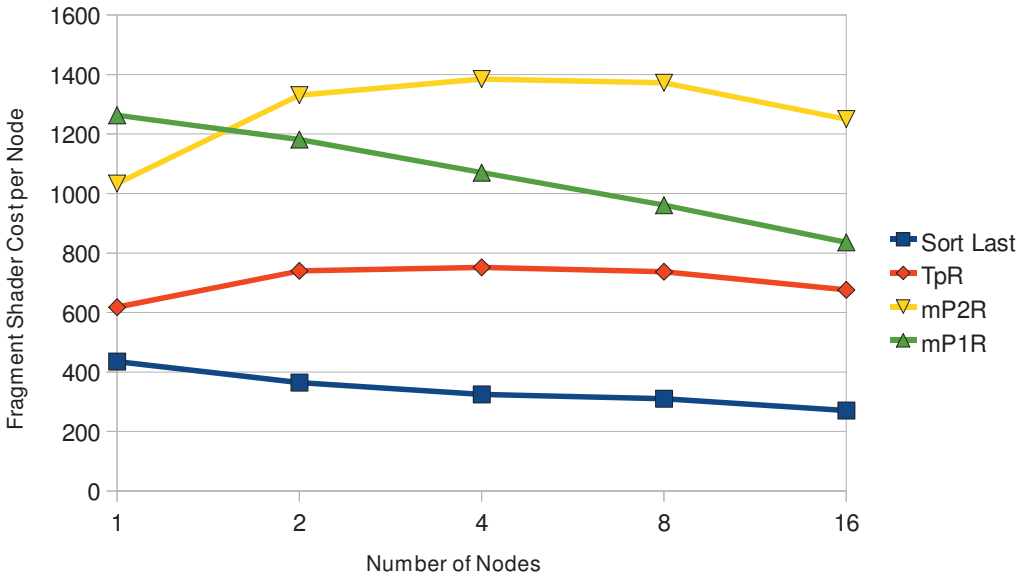

Figure 5.2: Fragment-bound performance per node (at 15 fps)

Figure 5.2 illustrates the general behaviour of each algorithm with respect to the change in render nodes. This graph depicts the fragment shader cost divided by the number of nodes and helps depict the amount of work each node can perform with increasing render nodes. In all but one instance, performance per render node starts to degrade after 8 render nodes. This degradation will always occur at some point when running any algorithm in a parallel fashion. Overheads often increase,

and there is almost always some sequential code that is not parallelizable (for more details refer to Ahmdal's law in any Parallel Programming book, for example the book Parallel Computing: Theory and Practice by Michael J. Quinn [20]). In the case of *mP1R*, performance starts to degrade much faster, starting at 2 render nodes. Reasons for this are best described in the Fragment-bound performance section 5.3 where we cover how long is each step of the *mP1R* pipeline with different number of render nodes. However, it seems to be due to having (1) a larger amount of overhead and (2) the Phase-1 Rendering step being easily parallelizable. This would suggest that, after a certain number of render nodes, the *TpR* algorithm may perform better than the *mP1R* algorithm.

These graphs do not indicate whether *Sort-last* would perform better or worse than any other algorithm if the number of nodes were to be increased significantly.

In the case where the Sphere model is rendered using an inexpensive fragment shader, and where performance is geometry-bound (fig. 5.3), the performance of our algorithms versus *Sort-last* do not perform as well as in the previous case. This is expected since our algorithms are specifically addressed to situations in which fragment shading takes time.

The relative performance of each algorithm in geometry-bound rendering is roughly what would be expected. *Sort-last* performs better than the *mP2R* and *TpR* algorithms. However, the *mP1R* algorithm performed better than *Sort-last* when running on more than one render node.

In this situation that *Sort-last* performs better than *mP2R* and *TpR* since they both render some of the geometry twice. *mP1R* performed better than *Sort-last* because a large amount of so-called "hidden" geometry was culled. Had there been

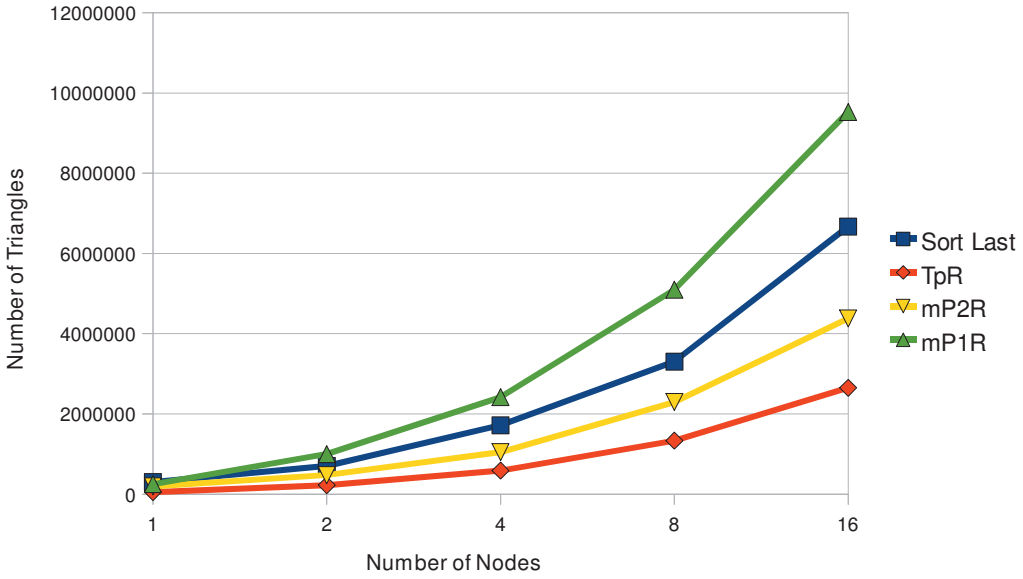

Figure 5.3: Geometry-bound performance (at 15 fps)

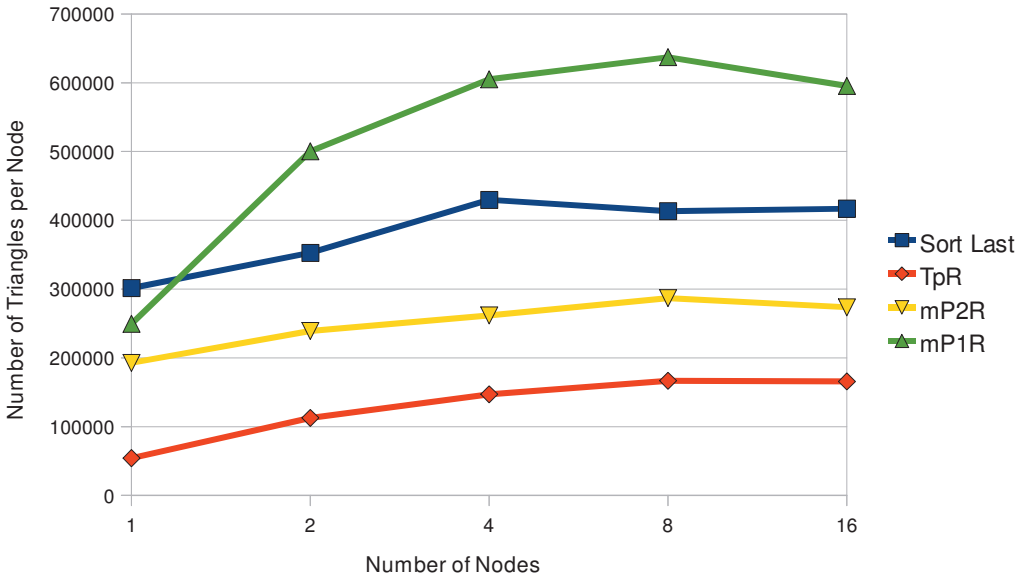

Figure 5.4: Geometry-bound performance per node (at 15 fps)

less hidden geometry, then *mP1R*'s results would have been worse and *Sort-last* might have performed better than all three proposed algorithms.

Sort-last for 4 render nodes produced unexpected results. The irregular curve, rather than the expected bell shape, might be due to the specific choice of testing data.

The slope of *mP1R* curve decreases at a much greater rate than any of the other algorithms. This is probably due to the same factors as in the fragment-bound test case (extra overheads in the implementation and improvements in Phase-1 are less important as the work is distributed to a larger number of nodes).

5.2 Geometry-bound internal performance

All the graphs show the percentage of the total time spent (percent time) in each rendering step with an increase in the number of render nodes. Percent time instead of actual time is chosen since rendering at exactly 15 fps is very difficult to achieve. Some of the steps are combined to make the graph more readable. The combined steps follow:

Render - The first rendering phase.

Share - Reading the data produced in the first rendering pass and communication to other nodes.

Order - Redistribution of work used for rendering the second phase.

Render(2) - The second rendering phase.

Order(2) - Bounding-box test (*mP1R* algorithm).

Send Pixels - Sending of pixels information to the render node.

Synchronization - Overhead resulting to load imbalance.

Other - Miscellaneous processing. In no instance did this step consists of more than 1% of the time.

In every instance, the synchronization cost increases as the number of nodes increases, the time spent synchronizing a single render node being 0%. The case of the single render node is self-evident as all render nodes will reach the same point at the same time. The increase in time spent synchronizing is expected as it becomes harder and harder to have a balanced workload when increasing the number of nodes. Another noticeable difference in the graphs, is that the time spent sharing information between nodes also increases. The time spent reading and writing to the GPU remains the same (a percent time of roughly 5%). As expected, the time spent in the Binary swap algorithm increases. As noted in the Binary swap discussion (Section 2.3.1), only half the screen has to be sent when running this algorithm on two nodes, while $^{15}/_{16}{}^{\text{th}}$ of the screen must be communicated between render nodes when rendering on 16 nodes.

For *Sort-last* (fig. 5.5), the time spent rendering decreases since the overheads (Synchronization and Sharing) have increased, and time must be made up somewhere. The time spent rendering starts at 89.8% when rendering on a single node, which is reduced to 73.7% on 16 nodes.

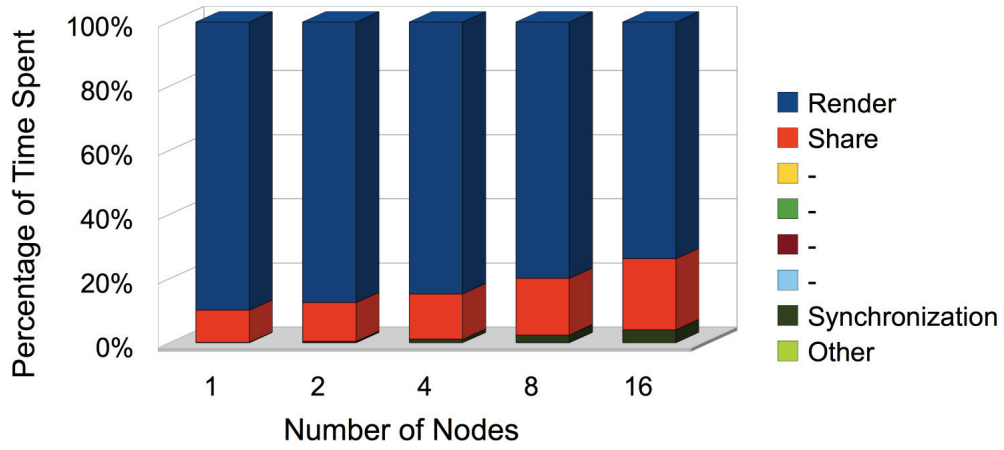

Figure 5.5: Time spent in Sort-last steps during the geometry-bound (at 15 fps)

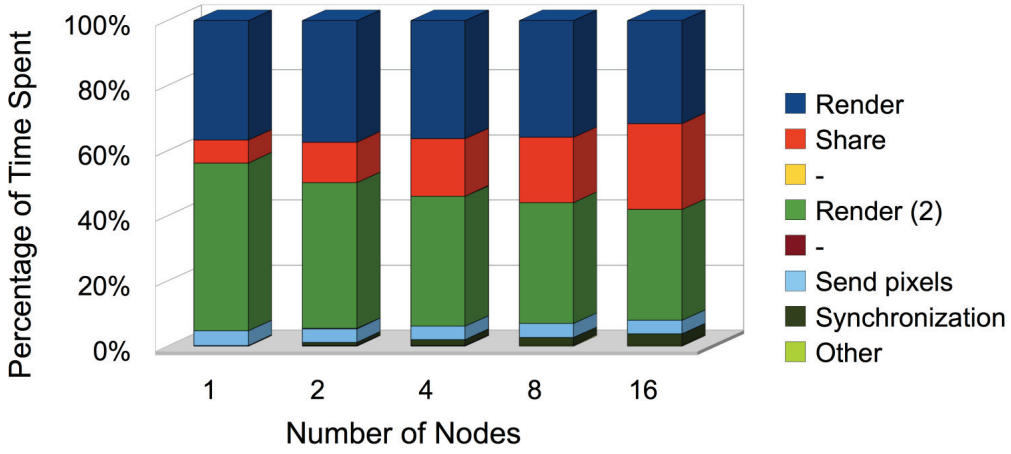

Figure 5.6: Time spent in TpR steps during the geometry-bound (at 15 fps)

For the *TpR* implementation (fig. 5.6), there are two communication phases. The first is similar to *Sort-last*, but slightly longer. The second, a send-pixels phase, stays roughly constant at 4.1% to 4.5%. The synchronization cost is similar to *Sort-last* (1.0% to 3.4% compared to 0.3% to 4.0%). The two other phases change with the amount of geometry rendered, both steps start at 88.2% (very similar to *Sort-last*), but goes down to 65.6% at 16 nodes due to the extra communication overhead.

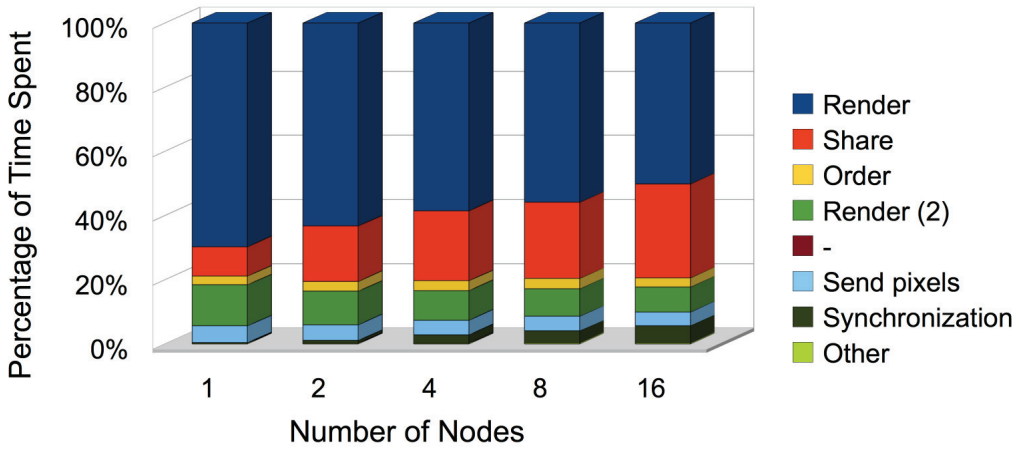

Figure 5.7: Time spent in mP2R steps during the geometry-bound (at 15 fps)

In the *mP2R* implementation (fig. 5.7), pixels communication, ordering, and ordering phases stay roughly static as the number of render nodes increases. Time spent communicating is slightly higher than in the *TpR* algorithm due to the extra overhead of sharing and using indices to geometry instead of node numbers (see the description of the algorithm for more details in Section 3.1.4). With regards to the two rendering phases, most of the time is spent in the first phase due to not having to render the whole scene in the second phase. This is unlike *TpR* where both render

phases took roughly the same amount of time. Synchronization overhead is slightly higher than the *TpR* algorithm (0.9% to 5.3% instead of 1.0% to 3.4%). This may be due to there being much more geometry to render(less granularity) and to having the second rendering phase distributing geometry primarily with respect to the number of fragments being rendered.

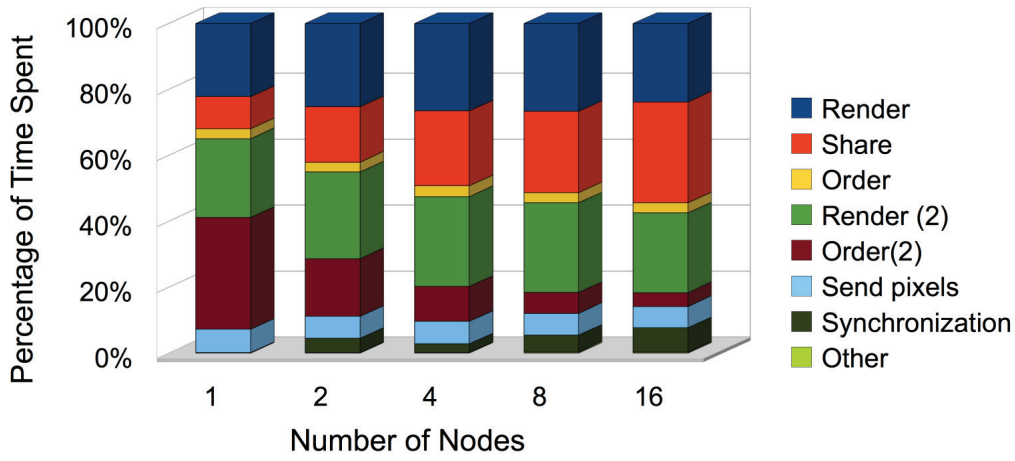

Figure 5.8: Time spent in mP1R steps during the geometry-bound (at 15 fps)

The *mP1R* run's (fig. 5.8) behaviour is quite different from the other algorithms. The first render step stays almost completely stable instead of changing as in the other algorithms. The order step is slightly quicker than in the *mP2R* algorithm, and it also remains almost constant as the number of nodes increase. The second order phase starts out very expensive at 33.9% but quickly diminishes to a very quick 4.2% when the render nodes reaches 16. Note that the second order phase occurs in parallel with the rendering of the visible geometry of the first phase. This means that the delineation between both phases may not be exactly as shown in the figures 5.8 and

5.12.

The Send pixels step takes longer than in the *mP2R* algorithm. This is because the final merged depth information is not available to the render node and extra pixels must be sent to the display node. However, the total time spent in this step stays nearly constant (from 5.2% at 1 node to 4.3% at 16 nodes). Performance could be improved when rendering on a single node since sending extra pixels information is not necessary (the whole image is rendered on that single node). This was not investigated because our focus did not include optimizing single render performance.

The Synchronization overhead of the *mP1R* algorithm differs from the other algorithms tested. This overhead is considerably higher than in the *mP2R* algorithm (2.6% to 7.4% instead of 0.9% to 5.3%). It starts quite high at 4.3% at 2 render nodes and decreases to 2.6% at 4 nodes.

The second order step is very hard to balance in the case of *mP1R*. The current form of the algorithm does not take into consideration the size of the bounding-box for each bucket. In addition, if any pixels within the bounding-box is potentially visible, then the test ends early. There is therefore, no assurance that each node will spend similar amounts of time in the second order phase. Because Coherence takes roughly half the time in the second order step (33.9% versus 17.5%), the overhead stemming from testing the bounding-box will be reduced. The reason for the greater imbalance at 8 and 16 nodes between *mP1R* and *mP2R* is probably due to the second rendering phase. The number of buckets which are potentially visible and must be rendered may not be the same between nodes. Finally, the amount of geometry rendered is much higher than in the *mP2R* algorithm, compounding the previous imbalance.

In summary, each of the algorithms described spend less time per frame rendering the actual image because they generate extra overheads. Increasing the number of nodes increases overheads further. At one point, each algorithm will no longer be able to produce an image at 15 fps. Based on our results, we expect that this point will occur quickest for *mP1R*, then *mP2R*, then *TpR* and finally *Sort-last* due to their relative overheads.

5.3 Fragment-bound internal performance

As in the previous subsection, all the graphs show the percentage of the total time spent in each rendering step with an increase in the number of render nodes. A Percent time was used instead of actual time since rendering at exactly 15 fps is very difficult to achieve. The descriptions of the steps can be found near the beginning of section 5.2.

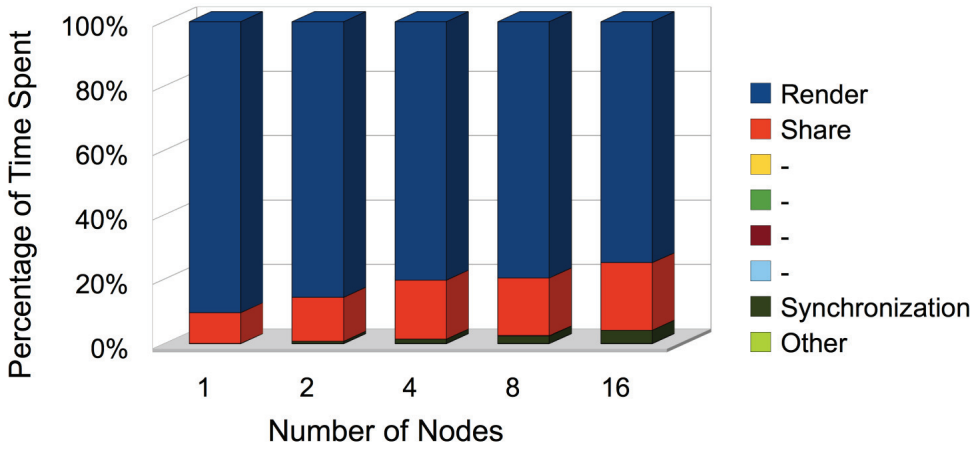

Figure 5.9: Time spent in Sort-last steps during the fragment-bound test (at 15 fps)

Sort-last (fig. 5.9) shows very similar results to those obtained when rendering the geometry-bound test on the Sphere model (fig. 5.5).

In all the algorithms the synchronization and share costs increase with the number of render nodes. Please refer to Section 5.2 for an explanation as to why this is to be expected.

In all algorithms having two render phases, the first render phase decreases in a near-linear fashion as the number of nodes increases. The reason this step seems to decrease geometrically is that the number of nodes increases geometrically: 1, 2, 4, 8 and 16. This is expected, the amount of geometry stays constant and is distributed evenly to each render node, so the amount of work given to each node is inversely proportional with the number of nodes.

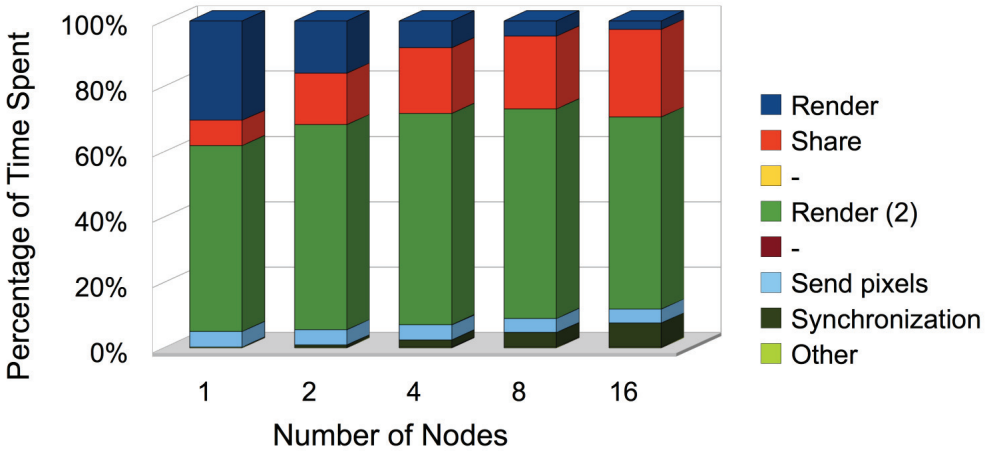

Figure 5.10: Time spent in TpR steps during the fragment-bound test (at 15 fps)

Regarding the *TpR* algorithm (fig. 5.10), other than the first render phase being faster as render nodes increases, there are two visible differences between the frag-

ment-bound and the geometry-bound (fig. 5.6) tests. First, there is considerably more time spent rendering in the second phase, and second, the imbalance is much worse. Time spent in the synchronization step was from 0.7-4.1% as opposed to 0.7-7.4%. Each render node renders a different number of visible fragments and, since a large portion of the rendering time is spent shading these fragments, the imbalance can be expected to balloon. *Sort-last*'s fragment shader is much less expensive and thus mitigates the imbalance effect. There might also be less of an imbalance in the number of fragments each render node must shade as each render node must shade all the fragments, not just the ones which are visible.

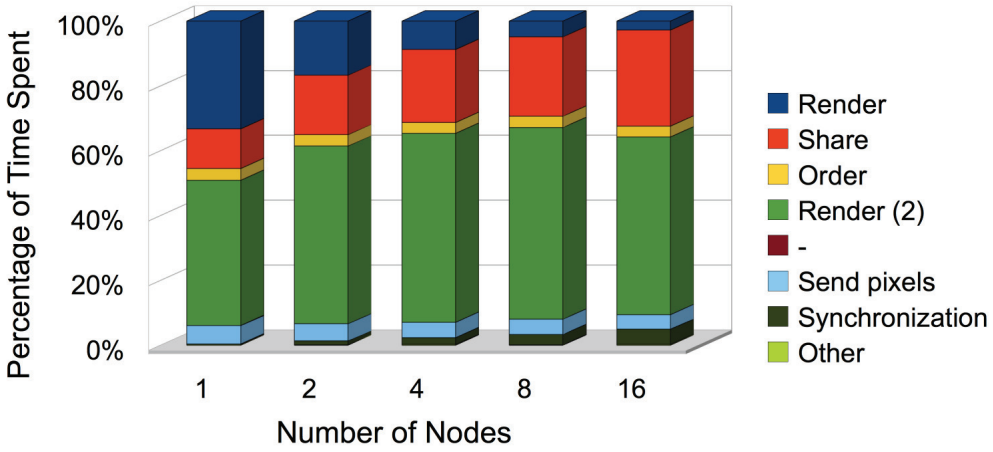

Figure 5.11: Time spent in mP2R steps during the fragment-bound test (at 15 fps)

mP2R allowed the most expensive fragment shader to be used while running the fragment-bound test. A significant amount of time was saved by culling fragments. Figure 5.11 shows that very little time is spent other than on shading fragments. Even with an expensive fragment shader, the imbalance is similar to that of *Sort-last*

(1.1% to 4.6% instead of 1.0% to 4.1%). Even when rendering with 16 nodes, over half the time was spent in the second rendering phase (57.6%) most of which is fragment shading.

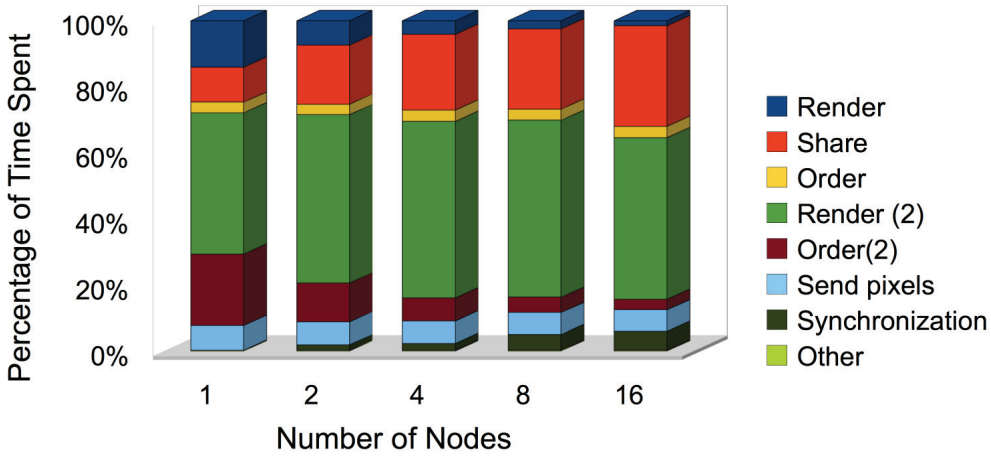

Figure 5.12: Time spent in mP1R steps during the fragment-bound test (at 15 fps)

Rendering using the *mP1R* algorithm on scenes with costly fragment shaders (fig. 5.12) is very different than in a geometry-bound scene (fig. 5.8). In the fragment-bound case, the first phase is faster, taking roughly half the time than any of the other algorithms. The bounding-box test is also much improved. This is mostly due to the implementation decisions. Since the bounding-box test is run on the CPU and fragment shader operations run on the GPU, both are run in parallel. However, the `gluSphere()` function spends most of its time calculating vertices on the CPU when using a simple fragment shader. If geometric complexity were to be produced through a loop within the vertex shader instead of increasing the stacks and slices, then we would observe parallelism in Figure 5.8.

mP1R's synchronization takes more time than that for the *mP2R* algorithm for the same reasons as discussed in the geometry-bound test (Section 5.2). However, synchronizing time is considerably less than with the *TpR* algorithm (1.6% to 5.8% instead of 0.7% to 7.4%). Although no dip appears in the synchronization time between 2 and 4 render nodes, this is probably due to there being less time spent in the second ordering step. Some of the reasons this algorithm cannot render as expensive a fragment shader as the *mP2R* algorithm for render nodes greater than 1 is illustrated in Figure 5.12. The first render phase is very short, so improvements to this phase will not have a significant impact on the overall performance, and the extra overheads (send pixels takes longer, as well as having an Order (2) step). In addition, is that the depth information is not exact after the first phase. Because more fragments must be shaded, the second render phase is not as effective.

Figure 5.12 shows that the choice of bounding-box bucket is of utmost importance to the *mP1R* algorithm. Too many, and the time spent in the test will be too lengthy. Too few or badly chosen bucket will not allow the system to cull enough geometry for the algorithm to perform well.

5.3.1 Conclusion

After introducing so many algorithms, choosing the correct one to render a specific scene can be difficult, yet of prime importance. A poorly chosen algorithm will not perform optimally due to redundant operations or overheads. Please refer to 5.13 for a visual approximation of when to use each algorithm.

In cases where there is little to no possible fragment culling, or where there is a

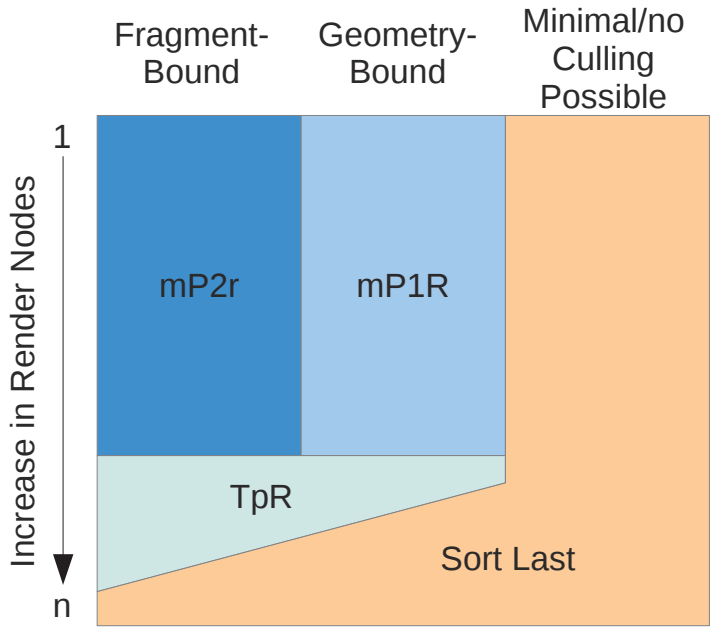

Figure 5.13: When to use each algorithm (not drawn to scale, only for demonstration purposes)

very large number of render nodes, *Sort-last* is a good option. It is a very simple algorithm. When the scene has little opportunity to cull fragments (little overlapping geometry), little time is gained in using a more complex algorithm that will split the rendering pipeline into phases. *Sort-last* has a smaller sequential workload (less communication, less time spent distributing work, and often less geometric workload) and will therefore perform better with a large number of nodes. On the other hand, if we are required to keep the number of render nodes reasonable, and there is an opportunity to cull a significant number of fragments, the other three algorithms will perform better.

When a high geometric load is of primary consideration, *mP1R* algorithm may be the best performer (see fig. 5.3).

In contrast, when the fragment shader is the primary workload, the *mP2R* algorithm performs very well (see fig. 5.1 for an example).

In more complex situations where there is a mix of geometric and fragment workload, the choice will be between the *mP1R* or *mP2R* algorithms.

However, in cases where there is little time to render fragments due to Communication and not due to rendering geometry, the *TpR* algorithm may perform best, its reduced non-geometric overheads being expected to trump *mP2R*'s and *mP1R*'s requirement for more bandwidth and extra ordering steps.

5.4 Power Plant example

Although we were able to identify situations where one more of our algorithms produced better results than *Sort-last*, it should not be forgotten that our conclusions are based on synthesized models designed for testing purposes. For this reason, we chose to perform a series of tests using the UNC's Power Plant 3D scene [26].

UNC's Power Plant has been used for research in "collision detection, interactive rendering, simplification, texture and image-based techniques, virtual environments, visibility culling, and other research areas" [26]. It should, therefore, be a good candidate for a so-called "realistic" test. Our aim was to observe whether our results would vary significantly from the Sphere model. Note that the geometry-bound test was not performed. Power Plant models with smaller number of triangles and level of detail were neither available nor within the scope of our research.

Due to the reasons explained above (Section 4.3.2.4), it was not possible to use the *mP1R* algorithm on the Power Plant model. We have also not chosen to show a

detailed analysis of the time spent in each step while rendering the Power Plant scene because the behaviour is very similar to that of the Sphere model.

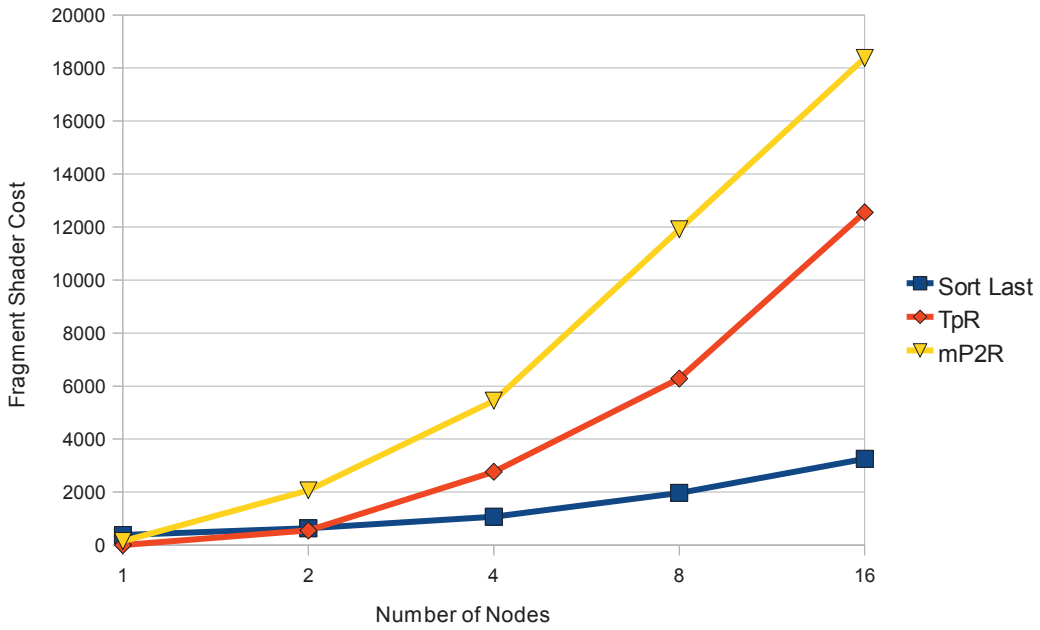

Figure 5.14: Power Plant fragment-bound performance (at 15 fps)

As we had expected, the Power Plant performance (fig. 5.14), is similar to the Sphere model test scene (fig. 5.1) with (when rendering with four or more nodes) *mP2R* performing best, *TpR* is second place, and *Sort-last* performs the worst. The performance difference using 1 or 2 render nodes is not very visible in this graph but is much more visible in Figure 5.15 which illustrates the amount of work done by each node.

When the results are examined on a per-node basis, Power Plant performance (fig. 5.15) appears considerably different from that of the Sphere model (fig. 5.2) especially in the case when rendering with few nodes. Since most of the time is spent

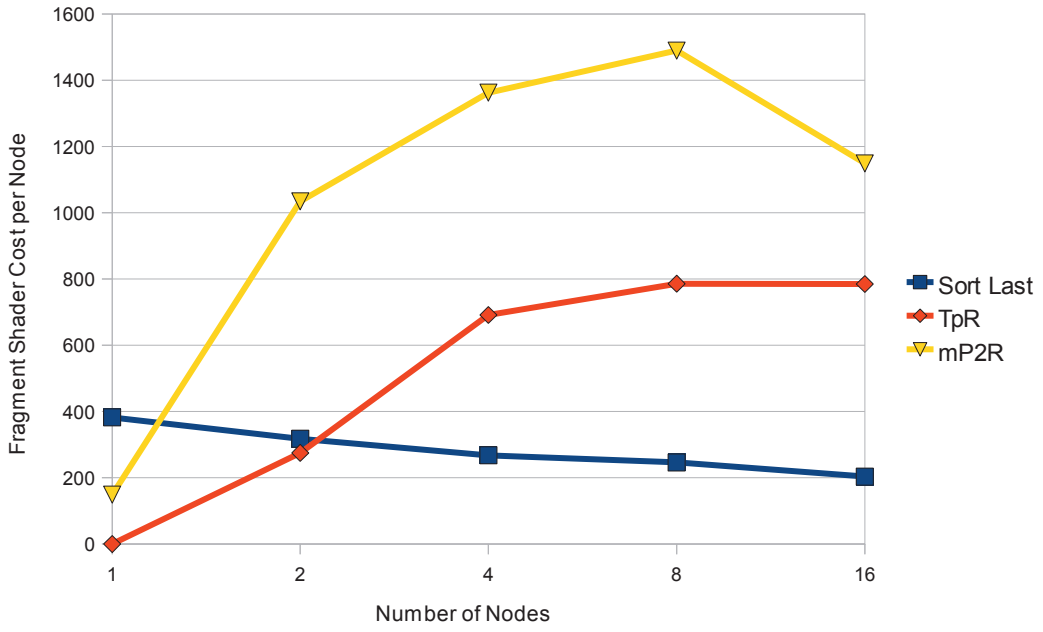

Figure 5.15: Power Plant fragment-bound performance per node (at 15 fps)

rendering the geometry when rendering with such a low number of nodes, it should not come as a surprise that *TpR* and *mP2R* would perform so poorly. With so little time left to render the expensive fragment shader, overheads imposed by rendering some (or all) the geometry twice will reduce this time even further, and even culling fragments may not mitigate the problem by making up the time.

The results for *mP2R* also show a dip in the fragment shader cost per node when rendering with 16 nodes. We expect this is due to the GPU performing poorly when the fragment shader cost becomes too large (Please refer to the GPU Module implementation in Section 4.3.2.2 for more details).

The increase in performance when using both *mP2R* and *TpR* algorithms is much higher when rendering the Power Plant model. This may be due to having more occluded fragments. In other respects, our results for the two models are quite similar.

We expect that the *mP1R* algorithm would have performed similar to the Sphere model since the *mP2R* algorithm also acted similarly. We expect that *mP1R*'s performance would fall somewhere between that of *mP2R* and *TpR*. However if the geometry partitions are chosen poorly and little geometry is culled, then *mP1R* might perform worse than *TpR*, but still better than that of *Sort-last* when rendering with 4, 8 or 16 nodes. Our assumption is that *mP1R* would act like *TpR* with some extra overhead imposed by indexing and running the bounding-box test. *mP1R* would have to render a fragment shader of less than half the complexity of *TpR* to perform worse than *Sort-last*. We consider it highly unlikely that these extra overheads would result in such a reduction in performance.

In summary, the testing of our algorithms on realistic 3D scene yielded some surprising results. We had expected a greater discrepancy between the algorithms' performance on the Power Plant and Sphere models. However, we found the results to be similar, especially when using 4 and 16 nodes, and significantly better when using 8 nodes (fragment "cost" of 11914 versus 10976 for *mP2R* and 6284 versus 5895 for *TpR*).

Based on this testing in a real application, we can conclude that our three algorithms *mP2R*, *mP1R* and *TpR* are of merit.

5.5 Load-balancing Results

Having shown the merit of our algorithms, we will demonstrate how the *mP2R* algorithm behaves under different load-balancing algorithms. As described earlier, load-balancing techniques which redistribute within a single Phase and those which redis-

tribute between different frames are not covered due to time and implementation constraints. However, we have implemented and tested the four Phase-2 load-balancing techniques described in Section 3.2.3.

We chose to run our tests on the *mP2R* algorithm instead of the *mP1R* algorithm due to there being fewer variables, the dynamic nature of the distribution within Phase-1 and the bounding-box test would produce a large amount of noise in our results. The Sphere model was used since we wish to determine the synchronization costs when rendering a large amount of geometry and when rendering using an expensive fragment shader.

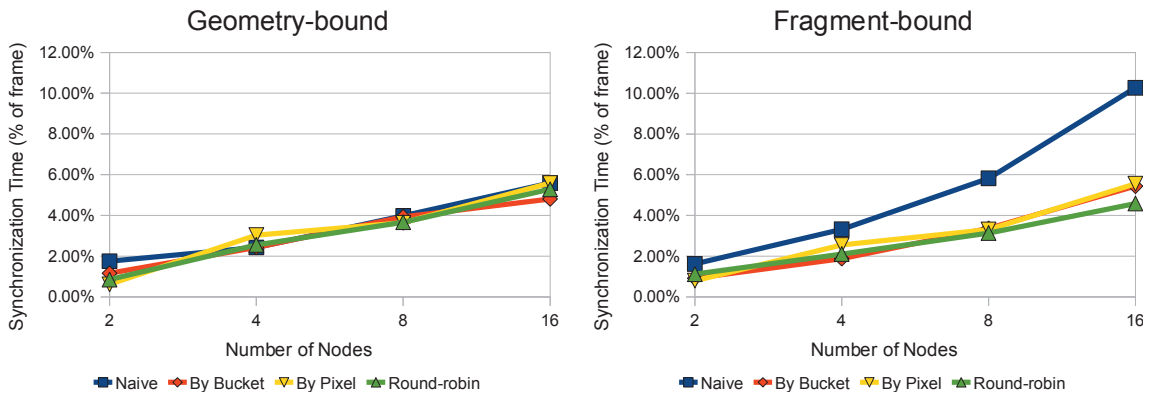

Figure 5.16: Load-balancing performance of the various load redistribution techniques while performing mP2R at 15 fps.

Figure 5.16 shows the synchronization costs (directly related to system imbalance) when rendering the Sphere model for geometry-bound and fragment-bound tests using the four different load-balancing algorithms.

The synchronization cost when rendering a geometry-bound scene is almost static, no matter the load-balancing technique. This is probably due to the fact that most of

the time is being spent rendering geometry in Phase-1 (see fig. 5.7), a small imbalance in Phase-2 will have little impact over the whole rendering pipeline.

This is not the case when rendering a fragment-bound scene. Since a large portion of the time is spent rendering fragments in Phase-2 (see fig. 5.11) an imbalance in the Render(2) Phase will impose a much larger imbalance over the whole pipeline. The Naïve redistribution has much greater imbalance than any of the other algorithms. This is due to performing no real load-balancing, but having different workloads due to having to render different numbers of "buckets" and pixels for each render node. The Round-robin distribution performed slightly better than the other three algorithms. In our tests, the redistributing by "bucket" and by pixels behaved similarly.

With roughly a 6% synchronization time when using 16 render nodes it shows that other load-balancing techniques might be an interesting avenue to investigate when rendering on larger cluster environments, but may not be that interesting when rendering on smaller clusters.

Chapter 6

Conclusions and further work

This thesis examined some of the algorithmic choices when rendering a 3-dimensional scene in a cluster environment under conditions where traditional cluster-based algorithms fail to perform well. As mentioned in Section 2.5, each traditional algorithm has various performance concerns when a large portion of the workload is related to fragment shading. Our work consisted of borrowing ideas from various sources such as single GPU fragment culling techniques, and modifying them to run on a cluster while performing load-balancing, reducing redundant operations and imposing limits to extra communication.

After our testing, we found merit in the various *Two-Phase Rendering* techniques proposed in this research. Through various experiments and analysis, we showed where and when they should be considered. These results should hold with current generation hardware. No significant paradigm shifts have occurred in any of the technologies used (GPU, CPU and networking). The synthetic tests can be made as simple or complex as necessary. We have shown a quantifiable increase in performance

given a set type of problem.

6.1 Contributions

The main contributions of this thesis are:

- Parallel fragment culling algorithms that perform reasonably well in a cluster environment without imposing too much extra overhead due to redundant work or communication. For example, in our results at 16 nodes, the amount of geometry rendered can be increased by up to 43%, and the fragment shader can be over 4.5 times more complex while keeping the same frame rate.

- Identification of load-balancing issues related to implementing these algorithms in a cluster environment, and development of methods to better balance the algorithms. Our use of the round-robin redistribution method reduced synchronization costs by up to 55%.

- Implementation of these algorithms on a graphics cluster.

- Compare and analyze the performance of these algorithms with respect to each other and to *Sort-last*, a well known older cluster rendering algorithm on an experimental basis.

6.2 Further work and limitations

Due to various implementation and time constraints there are still a number of interesting avenues to explore in this domain.

Although mentioned in Section 2.5, a number of load-balancing techniques implemented on more general cluster computing were not implemented due to time, hardware and software constraints. We encountered three difficulties in performing redistribution during the rendering phase: in the OpenGL drivers on the cluster, all the OpenGL calls must be done using a single thread, the multi-threaded MPI library is not currently working on the cluster, and there is no way to poll how much work is left on the GPU. These difficulties make the communication model overly complex, and since the only method to know if the GPU is free is to call `glFinalize()` which only returns when the GPU has emptied its pipeline, the GPU will regularly be starved for work.

Another avenue not taken is to take into account heterogeneous graphics clusters with non-uniform hardware, or scenes with non-uniform fragment shader costs. It might be interesting to try and identify the time spent rendering each bucket on the various hardware and distribute work along those lines.

It might also be interesting to see if these techniques can be implemented on a single machine with multiple GPUs such as ATI's Crossfire, or NVIDIA's SLI.

One of the limitations of these algorithms is the inability to handle transparency. To deal with this, the system must generate and communicate multiple fragments at a single pixels location. Tackling this problem might require a complete change of focus, to reduce the imposed communication overheads.

Bibliography

[1] InfiniBand™ Architecture Specification Volume 1. Technical report, InfiniBand℠ Trade Association, November 2007.

[2] Advanced Micro Devices. Amd announces worlds first 64-bit, x86 multi-core processors for servers and workstations at second-anniversary celebration of amd opteron processor. `http://www.amd.com/us/press-releases/Pages/Press_Release_97108.aspx`, February 2012.

[3] L. Bavoil, S. P. Callahan, A. Lefohn, J. a. L. D. Comba, and C. T. Silva. Multi-fragment effects on the GPU using the k-buffer. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 97–104, New York, NY, USA, 2007. ACM.

[4] A. Beaudoin, D. Goswami, and S. Mudur. Two-phase load distribution for rendering large 3D models on a graphics cluster. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –10, September 2009.

[5] Concordia University. ENCS HPC cluster specifications. `http://users.encs.concordia.ca/~cirrus/1.html`, February 2012.

[6] S. Eilemann and R. Pajarola. Direct send compositing for parallel sort-last rendering. In *SIGGRAPH Asia '08: ACM SIGGRAPH ASIA 2008 courses*, pages 1–8, New York, NY, USA, 2008. ACM.

[7] D. Filion and R. McNaughton. Starcraft ii: Effects and techniques. In *ACM SIGGRAPH 2008 classes*, SIGGRAPH '08, pages 133–164, New York, NY, USA, 2008. ACM.

[8] M. E. Gröller. *Coherence in Computer Graphics*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 1992.

[9] X. Jiang, K. Lei, H. Xiong, Y. Li, and J. Shi. A parallel framework for interactive rendering of massive complex scenes on PCs cluster. In *ICIG '07: Proceedings of the Fourth International Conference on Image and Graphics*, pages 978–983, Washington, DC, USA, 2007. IEEE Computer Society.

[10] B.-S. Liang, Y.-C. Lee, W.-C. Yeh, and C.-W. Jen. Index rendering: hardware-efficient architecture for 3-D graphics in multimedia system. *Multimedia, IEEE Transactions on*, 4(3):343 – 360, September 2002.

[11] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Comput. Graph. Appl.*, 14(4):59–68, 1994.

[12] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129 – 147, 1982.

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

[14] T. D. Nguyen and J. Zahorjan. Scheduling policies to support distributed 3D multimedia applications. In *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 244–253, New York, NY, USA, 1998. ACM.

[15] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9:38–71, March 1984.

[16] NVIDIA. NVIDIA Quadro FX 5600 and 4600. `http://www.nvidia.com/object/quadro_fx_5600_4600.html`, February 2012.

[17] J. Olbrantz. Multi-fragment deferred shading using index rendering. `http://quantam.devklog.net/p/Multi-Fragment%20Deferred%20Shading%20Using%20Index%20Rendering.pdf`, January 2011.

[18] H. Peng, H. Xiong, and J. Shi. Parallel-sg: research of parallel graphics rendering system on pc-cluster. In *VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, pages 27–33, New York, NY, USA, 2006. ACM.

[19] O. Pentakalos. An Introduction to the InfiniBand Architecture. `http://www.oreillynet.com/pub/a/network/2002/02/04/windows.html`, February 2012.

[20] M. Quinn. *Parallel computing: theory and practice.* McGraw-Hill computer science series: Networks–parallel and distributed computing. McGraw-Hill, 1994.

[21] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 97–108, New York, NY, USA, 2000. ACM.

[22] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. P. Singh. Load balancing for multi-projector rendering systems. In *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 107–116, Aug. 1999.

[23] A. Santilli. *Vortex: Deferred Sort Last Parallel Graphics Architecture.* PhD thesis, 2006.

[24] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2 (5th Edition) (OpenGL).* Addison-Wesley Professional, 2005.

[25] Stanford University. The stanford 3D scanning repository. `http://graphics.stanford.edu/data/3Dscanrep/`, February 2012.

[26] UNC Computational Geometry Group. Power plant model. `http://gamma.cs.unc.edu/POWERPLANT/`, February 2012.

[27] J. Williams and R. Hiromoto. A proposal for a sort-middle cluster rendering system. In *Intelligent Data Acquisition and Advanced Computing Systems: Tech-*

nology and Applications, 2003. Proceedings of the Second IEEE International Workshop on, pages 36 –38, September 2003.

Glossary

bucket A closely grouped collection of primitives. 22, 28, 31, 32, 34, 35, 37–42, 54–58, 73, 78, 85, 88

cluster A computer system comprised of multiple closely coupled nodes connected by a high speed network. The primary concern of such a system is performance. 1, 2, 6, 8, 9, 12, 17, 21, 25, 29, 31, 44–46, 51, 85–88

CPU Central Processing Unit, the hardware used for general computing tasks. 1, 27, 33, 34, 77, 86

fragment Defines the colour of a small volume (screen position and depth). Created during rasterization (see Section 2.1), visible fragments are used to define the colour of each pixel. 2, 5, 7–9, 14–16, 20, 21, 24–29, 31–34, 39, 41, 42, 45, 48, 53, 60, 62–66, 68, 72, 75–80, 82–88

fragment shader Code executed within the GPU that modifies fragments. 3, 5, 15, 16, 20, 21, 26, 27, 30, 38, 41, 42, 45, 48–51, 53, 62–66, 76–78, 80, 82–84, 87, 88

frame All the computation used to generate the final 2D image. 1, 4, 8, 10–12, 17–19, 22, 23, 27, 33–38, 41, 44, 46–48, 51, 55, 56, 58, 62, 74, 84

geometry A collection of vertices and edges that defines a 3D shape or model. 2, 4, 5, 7, 15, 16, 19–22, 24–30, 32–34, 36, 38–41, 45, 50, 51, 54, 56–58, 60–63, 65, 66, 68, 71–73, 75–80, 82–85, 87

GPU Graphics Processing Unit, highly parallelized hardware optimized to generate 2D images from 3D information. 3–7, 9, 14, 15, 25–30, 39, 41, 42, 46, 48, 50, 51, 53, 69, 77, 82, 86, 88

model Collection of primitives, with other properties that defines a 3D object. 1, 2, 4–6, 14, 21–23, 31, 49, 58, 60–64, 66, 75, 80–84, 88

node A piece of equipment attached to a network. In our visualization cluster, a node is a single computer, with multiple CPUs and a single GPU. 6–14, 17, 20, 22–30, 32, 35, 37–42, 45–47, 49–51, 55, 56, 61–66, 68, 69, 71–79, 81–83, 85

pixel Picture element. Defines the colour of a small 2D area of the monitor or image. 2, 5, 7–10, 12, 13, 15, 16, 24–27, 30, 32, 34–41, 47, 51, 55, 56, 58, 69, 71, 73, 78, 85, 88

primitive A single 3D element, a point, line or polygon. 5–8, 15, 22, 28, 31, 32, 38

render To generate a 2D image from 3-dimensional and related data (see Section 2.1 for more details), Usually performed through the use of specialized hardware. 1, 2, 4, 6–10, 15, 17, 20, 22–29, 32–42, 44, 45, 47, 49–51, 55, 56, 58, 61, 62, 64–66, 68, 69, 71–76, 78–83, 85

rendering See render. 1, 2, 4, 6–10, 12, 14, 16, 17, 20–37, 39, 41, 42, 44, 48, 49, 53, 57, 58, 61–64, 66, 68, 69, 71–77, 79–88

scene The input data and output image which has been/will be rendered. 1, 2, 4, 16, 20–23, 25, 28, 30, 31, 34, 36, 37, 41, 42, 44, 45, 47–50, 56, 58, 61, 63, 64, 71, 77–81, 83–86, 88

vertex shader Code executed within the GPU that modifies vertices. 4, 5, 77

Appendix A

Publication Resulting from our Research

Two-phase load distribution for rendering large 3d models on a graphics cluster [4].

Appendix B

Detailed Tables for the Performance of each Algorithm

	1 node	**2 nodes**	**4 nodes**	**8 nodes**	**16 nodes**
Sort Last	435	729	1298	2483	4323
TpR	618	1479	3007	5895	10812
mP2R	1033	2661	5540	10976	19995
mP1R	1263	2363	4282	7687	13371

Table B.1: Sphere model, Fragment-Bound Performance (cost of fragment shader) at 15 fps

	1 node	2 nodes	4 nodes	8 nodes	16 nodes
Sort Last	301523	705333	1719298	3306638	6669767
TpR	54125	225353	588264	1333661	2651032
mP2R	192614	478185	1047169	2295003	4377255
mP1R	249635	1000901	2420000	5096751	9528286

Table B.2: Sphere model, Geometry-Bound Performance (number of triangles) at 15 fps

	1 node	2 nodes	4 nodes	8 nodes	16 nodes
Sort Last	382	635	1071	1970	3256
TpR	0	550	2765	6284	12555
mP2R	149	2068	5449	11914	18380

Table B.3: Power Plant model, Fragment-Bound Performance (cost of fragment shader) at 15 fps

	2 nodes	4 nodes	8 nodes	16 nodes
Naive	1.75%	2.43%	3.98%	5.59%
By Bucket	1.16%	2.43%	3.94%	4.81%
By Pixel	0.62%	3.03%	3.64%	5.59%
Round-robin	0.85%	2.56%	3.67%	5.29%

Table B.4: Sphere model, Geometry-bound, Time spent synchronizing while performing load-balancing algorithms using mP2R at 15 fps.

	2 nodes	**4 nodes**	**8 nodes**	**16 nodes**
Naive	1.62%	3.32%	5.82%	10.27%
By Bucket	0.92%	1.88%	3.36%	5.44%
By Pixel	0.80%	2.55%	3.31%	5.56%
Round-robin	1.12%	2.10%	3.14%	4.59%

Table B.5: Sphere model, Fragment-bound, Time spent synchronizing while performing load-balancing algorithms using mP2R at 15 fps.

Appendix C

Detailed Tables for the Time Spent in each Phase in the Sphere Test

	1 node	2 nodes	4 nodes	8 nodes	16 nodes
01-Init(1)	0.0%	0.0%	0.0%	0.0%	0.0%
02-Order(1)	0.0%	0.0%	0.0%	0.0%	0.0%
03-GPURender(1)	89.6%	87.4%	84.6%	79.8%	73.6%
04-GPUend(1)	0.2%	0.1%	0.1%	0.1%	0.2%
05-GPURead(1)	3.7%	3.4%	3.2%	3.6%	3.7%
06-Synchronize(1)	0.0%	0.3%	1.0%	2.3%	4.0%
07-LoadBalance(1)	0.0%	0.0%	0.0%	0.0%	0.0%
08-Share(1)	6.4%	8.8%	10.9%	14.0%	18.5%
09-Cleanup&Merge	0.0%	0.0%	0.0%	0.0%	0.0%

Table C.1: Geometric Times - Sort Last

	1 node	2 nodes	4 nodes	8 nodes	16 nodes
01-Init(1)	0.0%	0.0%	0.0%	0.0%	0.0%
02-Order(1)	0.0%	0.0%	0.0%	0.0%	0.0%
03-GPURender(1)	36.4%	37.1%	36.1%	35.8%	31.5%
04-GPUend(1)	0.2%	0.2%	0.1%	0.0%	0.1%
05-GPURead(1)	1.7%	1.8%	1.9%	1.8%	1.8%
06-Synchronize(1)	0.0%	1.0%	1.8%	2.3%	3.4%
07-Share(1)	3.4%	8.6%	13.6%	16.0%	22.4%
08-SendLoad(1)	0.0%	0.0%	0.0%	0.0%	0.0%
09-Init(2)	0.1%	0.1%	0.1%	0.1%	0.1%
10-GPUWrite(2)	2.0%	2.1%	2.2%	2.2%	2.2%
11-Order(21)	0.0%	0.0%	0.0%	0.0%	0.0%
12-GPURender(21)	51.3%	44.7%	39.9%	37.1%	34.0%
13-Order(22)	0.0%	0.0%	0.0%	0.0%	0.0%
14-GPURender(22)	0.0%	0.0%	0.0%	0.0%	0.0%
15-GPUend(2)	0.3%	0.1%	0.1%	0.1%	0.1%
16-GPURead(2)	1.5%	1.6%	1.6%	1.6%	1.5%
17-Synchronize(2)	0.0%	0.0%	0.0%	0.0%	0.0%
18-LoadBalance(2)	0.1%	0.1%	0.1%	0.3%	0.2%
19-Share(2)	3.0%	2.7%	2.5%	2.7%	2.6%
20-Share(2)-Update	0.0%	0.0%	0.0%	0.0%	0.0%
21-Cleanup&Merge	0.0%	0.0%	0.0%	0.0%	0.0%

Table C.2: Geometric Times - TpR

	1 node	2 nodes	4 nodes	8 nodes	16 nodes
01-Init(1)	0.0%	0.0%	0.0%	0.0%	0.0%
02-Order(1)	0.0%	0.0%	0.0%	0.0%	0.0%
03-GPURender(1)	69.4%	63.0%	58.3%	55.7%	49.9%
04-GPUend(1)	0.3%	0.2%	0.1%	0.2%	0.1%
05-GPURead(1)	3.3%	3.5%	3.5%	3.5%	3.5%
06-Synchronize(1)	0.1%	0.9%	2.6%	3.7%	5.3%
07-Share(1)	3.9%	11.6%	16.1%	18.1%	23.6%
08-SendLoad(1)	0.0%	0.0%	0.0%	0.0%	0.0%
09-Init(2)	0.1%	0.1%	0.1%	0.1%	0.1%
10-GPUWrite(2)	2.0%	2.2%	2.2%	2.2%	2.2%
11-Order(21)	2.7%	3.1%	3.1%	3.2%	2.9%
12-GPURender(21)	12.7%	10.3%	9.1%	8.5%	7.7%
13-Order(22)	0.0%	0.0%	0.0%	0.0%	0.0%
14-GPURender(22)	0.0%	0.0%	0.0%	0.0%	0.0%
15-GPUend(2)	0.1%	0.1%	0.1%	0.1%	0.1%
16-GPURead(2)	1.5%	1.6%	1.6%	1.6%	1.6%
17-Synchronize(2)	0.0%	0.0%	0.0%	0.0%	0.0%
18-LoadBalance(2)	0.1%	0.1%	0.1%	0.2%	0.2%
19-Share(2)	3.7%	3.3%	3.0%	3.0%	2.7%
20-Share(2)-Update	0.0%	0.0%	0.0%	0.0%	0.0%
21-Cleanup&Merge	0.0%	0.0%	0.0%	0.0%	0.0%

Table C.3: Geometric Times - mP2R

	1 node	2 nodes	4 nodes	8 nodes	16 nodes
01-Init(1)	0.0%	0.0%	0.0%	0.0%	0.0%
02-Order(1)	0.0%	0.0%	0.0%	0.0%	0.0%
03-GPURender(1)	22.1%	25.2%	26.5%	26.5%	23.6%
04-GPUend(1)	0.1%	0.1%	0.1%	0.1%	0.2%
05-GPURead(1)	3.4%	3.3%	3.8%	3.6%	3.6%
06-Synchronize(1)	0.0%	4.3%	2.6%	5.1%	7.4%
07-Share(1)	4.4%	11.3%	16.7%	18.9%	24.7%
08-SendLoad(1)	0.0%	0.0%	0.0%	0.0%	0.0%
09-Init(2)	0.1%	0.1%	0.1%	0.1%	0.1%
10-GPUWrite(2)	2.1%	2.1%	2.2%	2.2%	2.2%
11-Order(21)	3.0%	3.0%	3.3%	3.1%	3.1%
12-GPURender(21)	14.5%	16.1%	16.7%	16.5%	14.6%
13-Order(22)	32.1%	15.8%	8.8%	4.7%	2.5%
14-GPURender(22)	9.3%	10.0%	10.5%	10.5%	9.4%
15-GPUend(2)	0.1%	0.1%	0.1%	0.1%	0.2%
16-GPURead(2)	3.4%	3.4%	3.6%	3.5%	3.6%
17-Synchronize(2)	0.0%	0.0%	0.0%	0.0%	0.0%
18-LoadBalance(2)	0.1%	0.1%	0.1%	0.3%	0.2%
19-Share(2)	3.7%	3.3%	3.2%	3.0%	2.9%
20-Share(2)-Update	1.5%	1.5%	1.6%	1.6%	1.6%
21-Cleanup&Merge	0.3%	0.2%	0.2%	0.2%	0.2%

Table C.4: Geometric Times - mP1R

	1 node	2 nodes	4 nodes	8 nodes	16 nodes
01-Init(1)	0.0%	0.0%	0.0%	0.0%	0.0%
02-Order(1)	0.0%	0.0%	0.0%	0.0%	0.0%
03-GPURender(1)	33.9%	18.2%	10.1%	5.1%	2.7%
04-GPUend(1)	56.5%	67.4%	70.2%	74.5%	72.1%
05-GPURead(1)	3.5%	3.2%	3.5%	3.4%	3.4%
06-Synchronize(1)	0.0%	0.7%	1.4%	2.4%	4.1%
07-LoadBalance(1)	0.0%	0.0%	0.0%	0.0%	0.0%
08-Share(1)	6.0%	10.3%	14.7%	14.5%	17.5%
09-Cleanup&Merge	0.0%	0.0%	0.0%	0.0%	0.0%

Table C.5: Fragment Times - Sort Last

	1 node	2 nodes	4 nodes	8 nodes	16 nodes
01-Init(1)	0.0%	0.0%	0.0%	0.0%	0.0%
02-Order(1)	0.0%	0.0%	0.0%	0.0%	0.0%
03-GPURender(1)	30.1%	15.6%	7.9%	4.3%	2.3%
04-GPUend(1)	0.2%	0.3%	0.2%	0.2%	0.2%
05-GPURead(1)	1.9%	1.9%	1.9%	1.9%	1.9%
06-Synchronize(1)	0.0%	0.7%	2.2%	4.5%	7.4%
07-Share(1)	3.7%	11.7%	16.0%	18.1%	22.7%
08-SendLoad(1)	0.0%	0.0%	0.0%	0.0%	0.0%
09-Init(2)	0.1%	0.1%	0.1%	0.1%	0.1%
10-GPUWrite(2)	2.1%	2.2%	2.2%	2.3%	2.3%
11-Order(21)	0.0%	0.0%	0.0%	0.0%	0.0%
12-GPURender(21)	39.7%	20.5%	10.3%	5.4%	2.7%
13-Order(22)	0.0%	0.0%	0.0%	0.0%	0.0%
14-GPURender(22)	0.0%	0.0%	0.0%	0.0%	0.0%
15-GPUend(2)	17.2%	42.2%	54.4%	58.7%	56.1%
16-GPURead(2)	1.6%	1.6%	1.6%	1.6%	1.6%
17-Synchronize(2)	0.0%	0.0%	0.0%	0.0%	0.0%
18-LoadBalance(2)	0.1%	0.1%	0.1%	0.1%	0.1%
19-Share(2)	3.2%	3.1%	3.0%	2.7%	2.6%
20-Share(2)-Update	0.0%	0.0%	0.0%	0.0%	0.0%
21-Cleanup&Merge	0.0%	0.0%	0.0%	0.0%	0.0%

Table C.6: Fragment Times - TpR

	1 node	2 nodes	4 nodes	8 nodes	16 nodes
01-Init(1)	0.0%	0.0%	0.0%	0.0%	0.0%
02-Order(1)	0.0%	0.0%	0.0%	0.0%	0.0%
03-GPURender(1)	32.9%	16.4%	8.5%	4.5%	2.4%
04-GPUend(1)	0.3%	0.2%	0.2%	0.3%	0.3%
05-GPURead(1)	3.6%	3.7%	3.7%	3.6%	3.5%
06-Synchronize(1)	0.2%	1.1%	2.1%	3.1%	4.6%
07-Share(1)	6.3%	12.3%	16.7%	18.5%	23.8%
08-SendLoad(1)	0.0%	0.0%	0.0%	0.0%	0.0%
09-Init(2)	0.1%	0.1%	0.1%	0.1%	0.1%
10-GPUWrite(2)	2.4%	2.3%	2.3%	2.3%	2.3%
11-Order(21)	3.6%	3.5%	3.3%	3.5%	3.3%
12-GPURender(21)	10.0%	5.0%	2.6%	1.4%	0.7%
13-Order(22)	0.0%	0.0%	0.0%	0.0%	0.0%
14-GPURender(22)	0.0%	0.0%	0.0%	0.0%	0.0%
15-GPUend(2)	34.9%	49.9%	55.7%	57.7%	54.1%
16-GPURead(2)	1.6%	1.7%	1.7%	1.6%	1.6%
17-Synchronize(2)	0.0%	0.0%	0.0%	0.0%	0.0%
18-LoadBalance(2)	0.1%	0.1%	0.1%	0.1%	0.2%
19-Share(2)	4.0%	3.6%	3.1%	3.1%	2.9%
20-Share(2)-Update	0.0%	0.0%	0.0%	0.0%	0.0%
21-Cleanup&Merge	0.0%	0.0%	0.0%	0.0%	0.0%

Table C.7: Fragment Times - mP2R

	1 node	2 nodes	4 nodes	8 nodes	16 nodes
01-Init(1)	0.0%	0.0%	0.0%	0.0%	0.0%
02-Order(1)	0.0%	0.0%	0.0%	0.0%	0.0%
03-GPURender(1)	13.8%	7.2%	3.8%	2.1%	1.2%
04-GPUend(1)	0.2%	0.2%	0.3%	0.3%	0.2%
05-GPURead(1)	3.6%	3.5%	3.8%	3.7%	3.7%
06-Synchronize(1)	0.0%	1.6%	2.1%	4.8%	5.8%
07-Share(1)	4.8%	12.1%	16.9%	18.4%	24.5%
08-SendLoad(1)	0.0%	0.0%	0.0%	0.0%	0.0%
09-Init(2)	0.1%	0.1%	0.1%	0.1%	0.1%
10-GPUWrite(2)	2.2%	2.2%	2.2%	2.3%	2.3%
11-Order(21)	3.2%	3.2%	3.4%	3.3%	3.4%
12-GPURender(21)	9.0%	4.6%	2.4%	1.3%	0.7%
13-Order(22)	19.6%	9.9%	5.2%	2.8%	1.5%
14-GPURender(22)	8.0%	4.3%	2.4%	1.5%	0.9%
15-GPUend(2)	25.9%	42.2%	48.6%	50.8%	47.3%
16-GPURead(2)	3.6%	3.5%	3.6%	3.7%	3.7%
17-Synchronize(2)	0.0%	0.0%	0.0%	0.0%	0.0%
18-LoadBalance(2)	0.1%	0.1%	0.1%	0.1%	0.1%
19-Share(2)	3.9%	3.4%	3.2%	3.0%	2.8%
20-Share(2)-Update	1.7%	1.7%	1.6%	1.6%	1.6%
21-Cleanup&Merge	0.3%	0.2%	0.2%	0.2%	0.2%

Table C.8: Fragment Times - mP1R