Eurographics Symposium on Parallel Graphics and Visualization (2011)
T. Kuhlen, R. Pajarola, and K. Zhou (Editors)

Cross-Segment Load Balancing in Parallel Rendering

Fatih Erol[1], Stefan Eilemann[1,2] and Renato Pajarola[1]

[1]Visualization and MultiMedia Lab, Department of Informatics, University of Zurich, Switzerland
[2]Eyescale Software GmbH, Switzerland

Abstract

With faster graphics hardware comes the possibility to realize even more complicated applications that require more detailed data and provide better presentation. The processors keep being challenged with bigger amount of data and higher resolution outputs, requiring more research in the parallel/distributed rendering domain. Optimizing resource usage to improve throughput is one important topic, which we address in this article for multi-display applications, using the Equalizer *parallel rendering framework. This paper introduces and analyzes cross-segment load balancing which efficiently assigns all available shared graphics resources to all display output segments with dynamical task partitioning to improve performance in parallel rendering.*

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics; I.3.m [Computer Graphics]: Miscellaneous—Parallel Rendering

Keywords: Dynamic load balancing, multi-display systems

1. Introduction

As CPU and GPU processing power improves steadily, so and even more increasingly does the amount of data to be processed and displayed interactively, which necessitates new methods to improve performance of interactive massive data visualization systems. Parallel and distributed computing is being utilized in a wide range of applications that require high processing power on massive data. Consequently, the demand for more research in this domain produces improvements in efficiency of algorithms as well as better presents solutions to system optimization issues, like dealing with heterogeneity and inconsistent availability of resources, communication and I/O bottlenecks, changing workloads, etc.

High-resolution, large multi-display systems are becoming more and more commonplace in various contexts of interactive and/or immersive visualization installations as well as virtual reality systems. Built from arrays of projectors or flat-panel monitors driven by a distributed set of graphics workstations, the aim is to provide large screen real-estates and high image resolutions. Used in conjunction with massive amounts of data to be rendered, these display systems impose serious performance demands due to the required processing of large geometric/graphics data sets and delivery of massive pixel amounts to the final display destinations at interactive frame rates. The graphics pipes (GPUs) driving the display array, however, are typically faced with highly uneven workloads as the vertex and fragment processing cost is often very concentrated in a few specific areas of the data and on the display screen. Balancing the workload among the multiple available GPUs – which are inherently available in multi-display or projector walls – for optimal throughput becomes extremely important in these circumstances and is the core topic of this work, see also Figure 1.

A number of algorithms and techniques for parallel rendering have been proposed in the past, however, only a few truly generic (cluster-)parallel rendering APIs and systems have been developed and are available, including e.g. VR Juggler [BJH*01] (and its derivatives), Chromium [HHN*02], OpenGL Multipipe SDK [BRE05] or Equalizer [EMP09]. We have chosen the latter in this work for its resource usage configuration flexibility and its extensibility with respect to dynamic load balancing.

A serious challenge for all cluster-parallel rendering systems driving a large multi-display system is to deal with the varying rendering *density* per display, and thus the graphics load on its driving GPU(s). In this article, we present a novel dynamic load balancing approach built into the se-

DOI: 10.2312/EGPGV/EGPGV11/041-050

lected parallel rendering framework, namely *Cross-Segment Load Balancing* (CSLB), and analyze its performance in the case of a multi-display setup for polygon rendering applications. The next section gives background information about related work in parallel rendering. Section 3 explains how cross-segment load balancing works, and Section 4 compiles our performance improvement analysis for a set of test cases. Section 5 concludes the paper with a summary of results along with ideas for future improvements of the system.

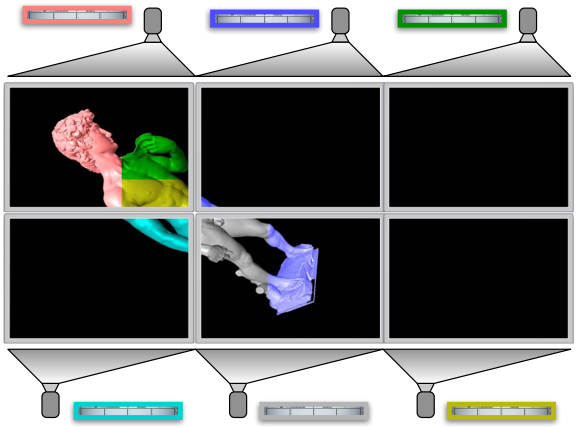


Figure 1: *In a multi-display setup,* Equalizer *can use cross-segment load balancing, which dynamically assigns the available resources to the output displays. When a resource has higher workload than others, it gets help from less loaded resources to improve performance. The top left segment is comparably overloaded, thus, help is acquired from less busy display resources connected to the right segments to equalize the workload.*

2. Related Work

A large amount of literature is available on parallel rendering. In the following we concentrate on the most relevant and related work, and we only roughly sample the remainder of the vast amount of proposed approaches to the various problems of distributed parallel rendering. Some approaches propose generic solutions for a wider range of applications, while others try to address very specific problems.

2.1. Parallel Rendering

Parallel, distributed rendering can either be a solution to efficiency problems through the distribution of workload among graphics resources, or it can be enforced by the application due to task constraints or the nature of the application. While some applications make use of parallel resources to simply increase rendering throughput, others may be forced to deal with distributed data and display resources, as the input data may only be available from different sources and the output image destination consists of multiple separate destination display channels.

In the context of Molnar et al.'s parallel-rendering taxonomy [MCEF94] on the sorting stage in real-time parallel rendering, various generally applicable concepts and results on cluster parallel rendering have recently been presented: e.g. [SFLS00b,SFLS00a,CKS02,BHPB03] on sort-first and sort-last architectures, or [SWNH03,CMF05,CM06,FCS*10] on scalability. On the other hand, many application specific algorithms have been developed for cluster based rendering. However, only a few generic APIs and libraries exist that support the development of a wide range of parallel rendering and visualization applications.

VR Juggler [BJH*01] is a graphics framework for VR applications which shields the application developer from the underlying hardware architecture and operating system. Its main aim is to make VR configurations easy to set up and use without the need to know details about the devices and hardware configuration, but not specifically to provide scalable parallel rendering. Chromium [HHN*02] provides a powerful and transparent abstraction of the OpenGL API, that allows a flexible configuration of display resources, but its main limitation is that it is focused on streaming OpenGL commands through a network of nodes, often initiated from a single source. OpenGL Multipipe SDK (MPK) [BRE05] implements an effective parallel rendering API for a shared memory multi-CPU/GPU system. It handles multi-pipe rendering by a lean abstraction layer via a conceptual callback mechanism and it runs different application tasks in parallel. However, MPK is not designed nor meant for distributed cluster based rendering. CGLX [DK11] intercepts a few GLX and OpenGL calls to reconfigure the application's output frusta for multi-display rendering. It does not concern itself with distributing and synchronizing the application data, and does not perform any scalable rendering or load-balancing. DRONE [RLRS09] is a high-level framework for GPU clusters, using a scene graph for rendering. [MWP01] presents another system that focuses on sort-last parallelization of very large data sets on tiled displays. Its performance is largely dependent on image compositing strategies among the available resources.

Equalizer [EMP09] was developed as a parallel rendering solution for porting OpenGL-based applications to deploy on multiple graphics nodes and multi-pipe systems through a run-time configurable client/server architecture. While it requires minimally-invasive changes to existing applications, it has become a productive standard middleware that enables applications to benefit from multiple computers and graphics cards in a scalable and flexible manner to achieve better rendering performance, visual quality and bigger display sizes. Due to its architecture, an application can run without modifications on any visualization system, from a simple work-

station or VR installations to large scale graphics clusters and display walls that employ multiple GPUs and nodes.

Besides its key advantages of scalable resources usage and flexibility of task decompositions, it also features integrated load balancing components that can dynamically distribute tasks among resources according to various parallel rendering modes (sort-first, sort-last, time- and view-multiplex etc.). Our cross-segment load balancing approach and results are demonstrated in the context of this parallel rendering framework.

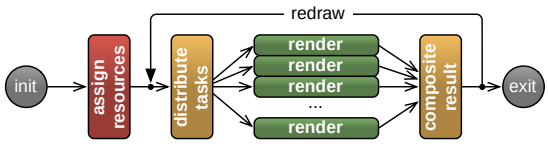
2.2. Load Balancing

In parallel, distributed rendering systems, it is important that graphics resources assigned to the destination display channels are proportional to the rendering load to achieve optimal performance. By maximizing resource utilization, interactive applications can benefit from minimized response times. In an ideal case all resources share the work volume equally and finish their tasks at the same time. Only few, often embarrassingly parallel tasks can easily achieve this through a static load balancing scheme.

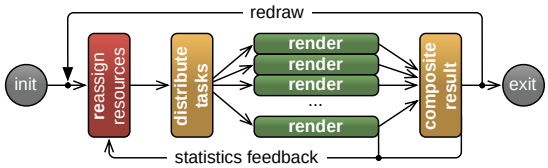
However, in practice this is rarely the case. Not only can the resources have heterogeneous processing power, the data and algorithms may have dependencies, and distribution of tasks can require extra overheads like communication and I/O, which complicates matters especially when there are network bottlenecks. Balancing of load fairly becomes even more difficult when the task partition is complicated or when prediction of parameters like workload cost is not easy to determine exactly.

Dynamic load balancing: Parallel systems that adapt the distribution of workload at runtime according to changing conditions are said to have dynamic load balancing. Such systems must assess the cost of partitions of work as accurately as possible, and assign them to the available resources in order to achieve optimal system throughput. An interactive multi-display system will have a non-uniform workload for its different display segments for the duration of its use and hence requires efficient dynamic load balancing.

Various parameters are to be considered for a dynamic adaptive load balancing scheme to be able to produce a fair distribution of tasks. In fact, running a load-balancing algorithm itself may have a non-negligible cost and thus profitability of adapting the workload within predetermined or dynamical parameters must be assured. Workload estimation may be particularly difficult in interactive visualization systems as it is hard to know in advance how costly the next set of rendered frames will be. Cost estimation requires information not only about the execution time for rendering some data, but also on the associated parallel overhead like synchronization, communication and I/O as well as availability and topology of the resources. Granularity limitations of tasks, priorities and dependencies among tasks are other parameters that display significance in choosing the best approach in load balancing. [OA02] categorizes various dynamic load balancing strategies according to different aspects of the load balancing process, e.g. how it is initiated, whether task queues are kept in a central location or are distributed, etc., and makes suggestions about their suitability to different kinds of applications.



(a) Static load balancing.



(b) Dynamic load balancing.

Figure 2: *Static load balancing partitions and distributes tasks just at the beginning of the application, whereas a dynamic load balancer re-adapts the partitioning and resource allocation during execution to make sure that the resources are utilized evenly to achieve optimal throughput.*

With respect to our targeted scenario of multi-projector and tiled display systems, corresponding work on load-balancing for parallel rendering has been presented in the following literature. In [SZF*99] the fundamental concepts of adaptive sort-first screen partitioning are presented and various tile-based load-balancing strategies are proposed. The usage of 2D tiles for load-balancing introduces an a priori granularity and is not well-suited for OpenGL applications, where geometry submission and processing introduces a significant overhead for small tiles. The authors alleviate this using various strategies to determine the optimal tile size and by merging tiles. Albeit rather relevant to our work due to employing different resources for rendering tiles within other display segments, this work separates from ours by concentrating on performance of some approaches for tiling the global scene in a balanced way through a scene graph, whereas we attack the balancing issue in a display segment based fashion with a more generic way within our parallelization framework transparent to the application. Furthermore, a hybrid extension also including sort-last task partitioning is introduced in [SFLS00a]. Past-frame rendering time is proposed as a simple yet effective cost heuristic in [ACCC04], however, cross-segment sort-first task distribution is not considered, only one display node has been used, which could affect the heuristic and load balancing. Pixel-based rendering cost estimation and kd-tree screen partitioning are exploited in [MWMS07] for dynamic load-balanced

sort-first parallel volume rendering. However, as [ACCC04], also [MWMS07] does not address cross-segment rendering in a tiled display environment. Similarly per-pixel vertex and fragment processing cost estimation and adaptive screen partitioning is proposed in [HXS09] but no cross-segment load balancing is considered.

Load balancing in Equalizer: Equalizer supports a wide variety of task distribution approaches from more easily load-balanced time- and view-multiplexing to more difficult sort-last or sort-first parallel rendering. Based on past-frame rendering times, basic adaptive load-balancing is provided for both sort-last and sort-first rendering. Sort-last load-balancing follows similar principles as in [MMD06] (for volumes) in that the data is split proportionally among the graphics resources based on their last frame rendering cost. Adaptive and dynamic screen-partitioning is supported in the framework as well, similar in principle as in [SZF*99, ACCC04, MWMS07, HXS09] but based on past rendering times. Additionally, dynamic frame resolution is supported, suitable for fragment cost bound applications, which adaptively alters the resolution of the output frame in order to achieve a constant frame rate. Based on this software, application specific optimal (sort-last) load balancing has been demonstrated for multi-resolution real-time rendering of very large terrain data in [GMBP10].

In this work we report on the new cross-segment load balancing mechanism available in the framework, that allows the normally uneven rendering load on N graphics pipes driving $M \leq N$ displays – the typical setup of many multi-panel or -projector display systems – to be redistributed across the different GPUs irrespective of their physical attachment to displays. This in reality very typical display wall situation, see also Figure 1, has not directly been addressed in past sort-first load balancing approaches and no ready-to-use solution exists in other parallel rendering frameworks. Note that our cross-segment load balancing approach also works for $M \geq N$, but this is a less common setup and is thus not specifically discussed in this paper.

3. Cross-Segment Load Balancing

In load balancing, it is as important to decide on the best way to assign tasks to the available resources as how the tasks are partitioned. As a dynamic load balancing approach, *cross-segment load balancing* (CSLB) tries to achieve optimal utilization of available resources through dynamic allocation of N GPUs to a set of M display destination channels, constituting the M segments of a multi-display system, e.g. a display wall or immersive installation. Commonly, each destination channel is solely responsible for rendering and/or compositing of its corresponding display segment.

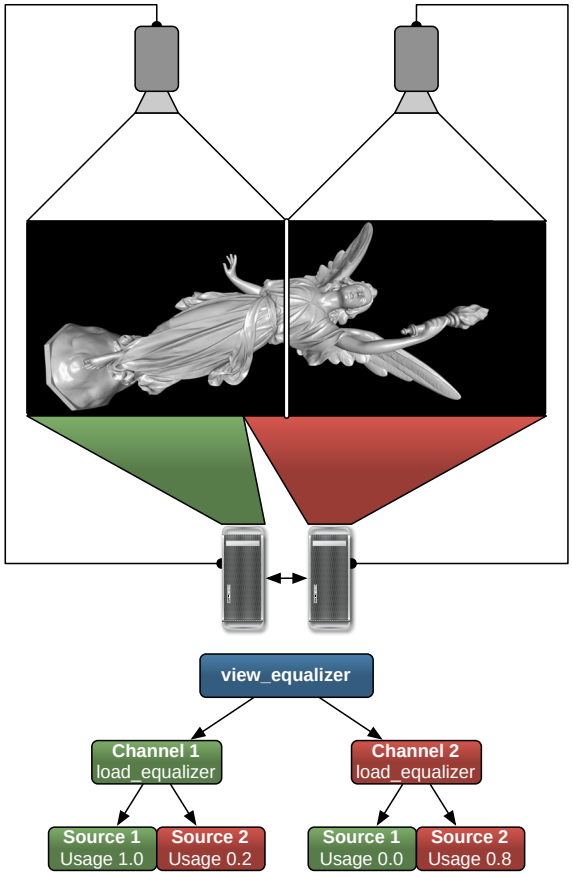
A key element of CSLB is that the M GPUs physically driving the M display segments will not be restricted in a one-to-one mapping to rendering tasks of the corresponding display segment. In fact, CSLB performs dynamic assignment of N graphics resources from a pool to drive M different destination display segments, where the M destination channel GPUs themselves may also be part of the pool of graphics resources. Dynamic resource assignment is performed through load-balancing components that exploit statistical data from previous frames for the decision of optimal GPU usage for each segment as well as optimal distribution of workload among them. The algorithm can easily be extended to use predictive load-balancing based on a load estimation given the application.

CSLB is implemented as two layers of hierarchically organized components, specified in the server configuration file. Figure 3 depicts a snapshot of a simple CSLB setup along with its configuration file. Two destination channels, *Channel1* and *Channel2*, each connected to a projector, create the final output for a multi-projector view. Each projector is driven by a distinct GPU, constituting the source channels *Source1* and *Source2*. But each source channel GPU can contribute to the rendering of the other destination channel segment.

In general, the M destination channels and their display segments are physically driven by M graphics pipes (GPUs). All destination channel GPUs, however, are as well part of the possibly larger pool of N source channel graphics resources. Thus in CSLB any display segment's own GPU can contribute, functioning as a generic source channel, to the rendering of any other display segment's final image at runtime, hence the term cross-segment load balancing.

Resource allocation: For CSLB, a *view_equalizer* component is attached to the top level of the parallel rendering task decomposition compound hierarchy, which handles the resource assignment. Each child of this root compound has one destination channel, constituting a display segment, with a *load_equalizer* component each. Hence the *view_equalizer* component supervises the different destination channels of a multi-display setup. The *load_equalizer* component on the other hand is responsible for the partitioning of the rendering task among its child compounds. Therefore, each destination channel of a display segment has its source channel leaf nodes sharing the actual rendering load. One physical graphics resource (GPU), being assigned to a source channel, can be referenced in multiple leaf nodes and thus contribute to different displays. For performance reason, one resource is at most assigned two rendering tasks, e.g., to update itself and to contribute to another display.

The 3D data, viewing configuration and user interaction will at runtime produce different rendering loads in each segment of a multi-display system. As the slowest segment will determine the overall performance of the system, it is important to dynamically adjust load among segments. The *view_equalizer* component analyzes the load of all segments based on past frame rendering statistics and adapts resource usage for each segment. For each rendered frame,



(a) CSLB resources setup.

```
compound
{
  view_equalizer {}
  compound
  {
    channel "Channel1"
    load_equalizer{}
    compound {} # self
    compound
    {
      channel "Source2"
      outputframe {}
    }
    inputframe{}
    ...
  }
  compound
  {
    channel "Channel2"
    load_equalizer{}
    compound {} # self
    compound
    {
      channel "Source1"
      outputframe {}
    }
    inputframe{}
  }
  ...
}
```

(b) CSLB configuration file format.

Figure 3: *For each destination channel that updates a display segment, a set of potential resources are allocated. The top-level* view_equalizer *assigns the usage of each resource, based on which a per-segment* load_equalizer *computes the 2D split to balance the assigned resources within the display. The left segment of the display has a higher workload, so, both* Source1 *and* Source2 *are used to render for* Channel1*, whereas* Channel2 *makes use of only* Source2 *to assemble the image for the right segment.*

the *view_equalizer* sets the usage of each leaf source channel compound, to activate or deactivate it for rendering.

Algorithm 1 presents how the decision is made to set the usage of resources. The CSLB is initialized with setting up statistic listener objects on the task decomposition and rendering compound nodes. This way the past frame rendering time statistics can be used for load-balancing purposes. The number of graphics resources (GPUs) is also determined during initialization. The dynamic load-balancing is then executed for each rendered frame and reassigns the N rendering source channel GPUs to the M output destination channels. Since image transmission and compositing can run concurrently to rendering (rasterization) [EMP09], the update time is the maximum of rendering and transmission times. Hence the destination channel redraw time is estimated at $t_{dest} \leftarrow \sum(\max(t_{source\_render}, t_{source\_transmit}))$. However, practically it has shown to be more beneficial to take a scaled average source channel rendering time as a heuristic to describe the rendering load, using $t_{dest} \leftarrow (t_{source\_average} \times \sqrt{n_{sources}})$ instead on Line 8 in Algorithm 1. This heuristic accounts for the non-linear relationship between number of resources and achieved performance. On Lines 10 and 11 the total frame rendering time t_{total} and workload per GPU t_{GPU} is determined.

Subsequently, for all destination output channels the required number of GPUs are computed and assigned for rendering of the next frame. First, on Lines 14 to 17 the total number of GPUs n_{dest_GPU} required to cover the workload of that channel is determined, as well as to how much the channel's own and any other GPU can be used, n_{self_usage}

and $n_{dest\_remaining}$, respectively. If the GPUs assigned in the previous frame are not sufficient to cover the current workload, Lines 18 to 21, then other available GPUs are assigned as well to this destination channel, Lines 22 to 24. This analysis and GPU assignment is done for all destination channels. The priority of self GPUs over GPUs used in the last frame over any other GPU optimizes compositing cost and inter-frame data coherency, respectively.

Task partitioning: Once the usage of the resources is set, the *load_equalizer* assigns work to each leaf compound according to its allowed usage. The *load_equalizer* can be configured to either perform sort-last (object space) or sort-first (image space) task partitioning. For partitioning both in 2D image space and in object database space, frame-to-frame coherency is taken into account while distributing the workload. The server keeps records of timing statistics about previous frames, gathered from the client channels that do the actual rendering. This information is used to predict work density, which is then used to repartition the rendering tasks to achieve a balanced load in the child components of a *load_equalizer* compound within the allowed usage boundaries.

Typically, the change in rendering workload on each display and graphics pipe is gradual during interactive visualization, rather than sudden and discontinuous in subsequent frames. For such applications, where frame coherency holds, making use of previous frame time statistics will provide a quite consistent accuracy in prediction of future workload. In practice, the load balancer also quickly catches up with more sudden load changes, and the distribution of workload among child compounds is realized in a quick and fair way.

Example: In Figure 3(a), the right segment, destination *Channel2* and rendering compound *Source2*, has a lower graphics load than the left segment, *Channel1* and *Source1*. *view_equalizer* analyzes the usage of the available resources and *load_equalizer* calculates optimal partitioning of segment rendering tasks, as per the algorithm mentioned above. Consequently, the system assigns 80% of the *Source2* GPU capacity to *Channel2*, and 20% to *Channel1*. The *Source1* GPU is fully assigned to *Channel1*. Therefore, overall a ratio of 1.2 graphics resources are assigned to *Channel1* and 0.8 to *Channel2*, the 'left' and 'right' tiled display segments, respectively, from the 2 physical GPUs in the system.

Cross-segment load balancing thus allows for optimal resource usage of multiple GPUs used for driving the display segments themselves as well as any additional source GPUs for rendering. It combines multi-display parallel rendering with scalable rendering for optimal performance.

Algorithm 1 Cross-Segment Load Balancing Algorithm.

1: Initialization:
2: Set up statistic load listeners on all leaf channels
3: Calculate total number of GPUs available n_{GPU}
4: **for** *each frame* **do**
5: Compute time needed to redraw each destination channel:
6: **for** *each destination compound* **do**
7: Compute its time to redraw:
8: $t_{dest} \leftarrow (t_{source\_average} \times \sqrt{n_{sources}})$
9: **end for**
10: $t_{total} \leftarrow \sum t_{dest}$
11: $t_{GPU} \leftarrow t_{total}/n_{GPU}$
12: **for** *each destination compound* **do**
13: Compute number of GPUs needed:
14: $n_{dest\_GPU} \leftarrow t_{dest}/t_{GPU}$
15: Usage of the destination's own GPU:
16: $n_{self\_usage} \leftarrow \min(1.0, n_{dest\_GPU})$
17: $n_{dest\_remaining} \leftarrow n_{dest\_GPU} - n_{self\_usage}$
18: **if** $n_{dest\_remaining} > 0$ **then**
19: Assign the same $n_{dest\_lastframe}$ GPUs used in last frame (up to $n_{dest\_remaining}$)
20: $n_{dest\_remaining} \leftarrow n_{dest\_remaining} - n_{dest\_lastframe}$
21: **end if**
22: **if** $n_{dest\_remaining} > 0$ **then**
23: Assign any other available GPUs (up to $n_{dest\_remaining}$)
24: **end if**
25: **end for**
26: **end for**

4. Results

Hardware and software setup: To compare performance improvement brought by utilization of cross-segment load balancing, we have run tests with various configurations on a PC cluster driving a 24Mpixel display, a 2×3 array of LCD panels at 2560×1600 resolution each. Each monitor is connected to an Ubuntu Linux node with dual 64bit AMD 2.2 GHz Opteron processors and 4GB of RAM. Each node employs two NVIDIA® GeForce® 9800 GX2 graphics cards, one of which is connected to the display monitor, while the other is used for rendering tasks through frame buffer objects when necessary. All nodes of the PC cluster each have a 1 Gigabit ethernet network interface.

Executables and data files are accessed through a network mounted disk on the gigabit network connection, and the *Equalizer* server is run on one node along with the polygon renderer application, which starts the rendering clients on all six display nodes each of which loads whole of model data to render. The polygonal rendering application is *eqPly*, which is a simple mesh based renderer and comes as an example application with the framework package that loads and displays *PLY* files and can play a recorded camera path from a text file. The application was modified minimally with extra logging for statistical data to make analysis easier, and a camera path was designed to move the model in different regions of the whole display area for a better view of the effects of load balancer under different load condi-

tions. The tests all target a tiled display view with full screen non-decorated windows on all of the monitors, with different configuration files selecting the assignment of resources to segments.

Models: To be able to analyze the results better, we have chosen to use three different sized models, namely *david1mm*, *david2mm* and *lucy*, with 56M, 8M and 20M triangles respectively. The smallest model, *david2mm*, can in fact be rendered at around 35 frames per second on our 2×3 tiled display cluster setup without dynamic load balancing, using one GPU per segment. It can fit into the graphics memory of modern GPUs, thus not requiring frequent and expensive CPU-to-GPU data transfers. In the mid-range, the full *lucy* model renders at around 4 frames per second only, still fitting into graphics card memory but load-balancing can already be beneficial. The most complex model *david1mm* pushes the GPU and its memory to the limit, rendered at only 1.9 frames in one second on our system.

Test configurations: We have used two sets of configurations for testing the performance. Due to further complexity of image assembly in sort-last partitioning, which requires more network traffic and higher level of synchronization points, we opted to use simpler 2D sort-first partitioning for task decomposition in our experiments, avoiding costly $z-depth$ or $\alpha-opacity$ compositing stages that could dilute the analysis of the actual sort-first load balancing.

The first set of configurations consists of *stat_2D_6to6*, *cslb_2D_6to6_2*, *cslb_2D_6to6_4* and *cslb_2D_6to6_6*. The first config, *stat_2D_6to6*, is our base setup, with each GPU statically connected to the display of its node. Network communication is limited to client and server command packets that drive the rendering process by setting view frustum and start of a new frame, and no dynamic balancing of work load occurs among nodes. The other *cslb_2D_6to6_K* configurations also use the six display GPUs for rendering, but do cross-segment load balancing by sharing the GPU resources with other segments so that the total number of available GPU resources for each segment is K. The extra resource, when $K > 1$, is a display GPU assigned to another segment, so making use of extra available resources for a segment will mean communication of the resulting image data over the network for assembly of the final local segment image.

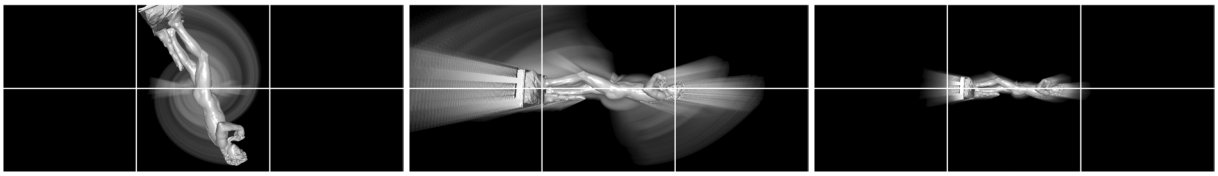
The second set of configurations are chosen similarly, but instead all of the 12 GPUs are made use of. For static distribution, we used *stat_2D_12to6* configuration that assigns half of the image segment on each node to one display GPU and half to the other GPU. Image data does not need to travel over the network, however, the image formed in a frame buffer object on the second GPU needs to be read back and assembled by the display GPU which renders the other half of the segment area locally. We also tested with a similar configuration called *stat_2D_12to6_LB*, which differed only by replacing the static half-half distribution to two GPUs on the same node with a local 2-way dynamic sort-first load balancer. For CSLB, we used *cslb_2D_12to6_2*, *cslb_2D_12to6_4*, *cslb_2D_12to6_6*, *cslb_2D_12to6_8*, *cslb_2D_12to6_10*, *cslb_2D_12to6_12* setups to observe the effect of cross-segment load balancing with a 12 GPU pool, with 2, 4, 6, 8, 10 or 12 GPUs shared and assigned to each segment respectively. Other than the 2-GPU assignment, all segments might be getting help from one or more other GPUs from other nodes, thus requiring network for image data transfers.

Tests: For each configuration and model, we have run the *eqPly* application three times to make sure we have consistent frame timing statistics among the three results for each frame. We observed very similar behavior in graphs plotting the frame timings by using either median, maximum or minimum of these three runs; but we display the graphs using the minimum from three runs in this paper, as the produced graphs are more easily viewable on paper due to smaller fluctuations in the data. Where oscillating curves, which happen in load balanced setups as the load balancer tries to constantly adapt, make it difficult to view, we overlay a moving average trend-line on the original data, and the relationships between different data series still hold while it is much easier to observe.

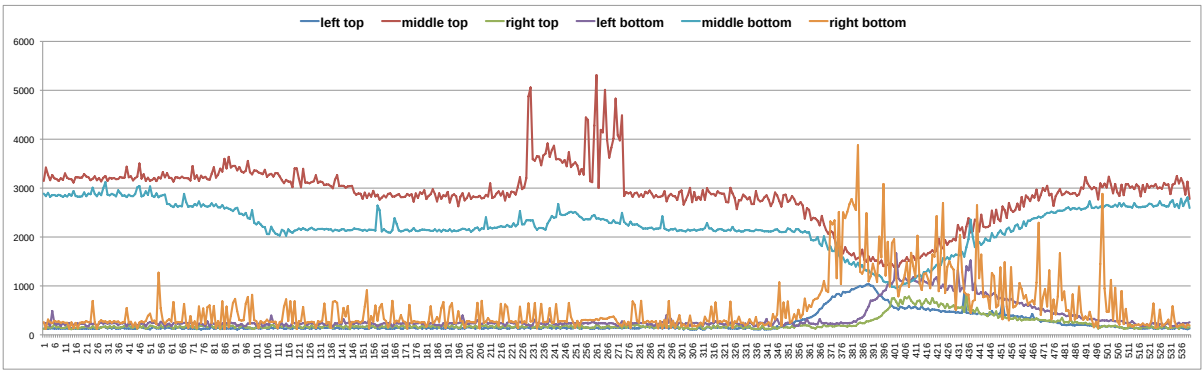
A camera path was replayed twice in each run, and the second loop is analyzed for frame time statistics, to avoid irregular effects on timings due to interference from client execution by initialization and caching processes. One loop of the camera path is designed as a sequence of 540 frames zooming in and out on the displayed model as well as rotating the model. Throughout the replay, the model starts zoomed out in the middle of the display region, and while rotating, it is zoomed in so that it starts covering all segments of the tiled display, and then zooms back out to the original position. By this, we aim to cover the following cases that would imitate a typical user experience with a big display:

- The model occupies the middle region whereas the side regions remain largely empty, which is bound to be a typical use case as users tend to place models centered for viewing in an interactive visualization system.
- The model is zoomed in on to observe more detail, occupying most of the display area.

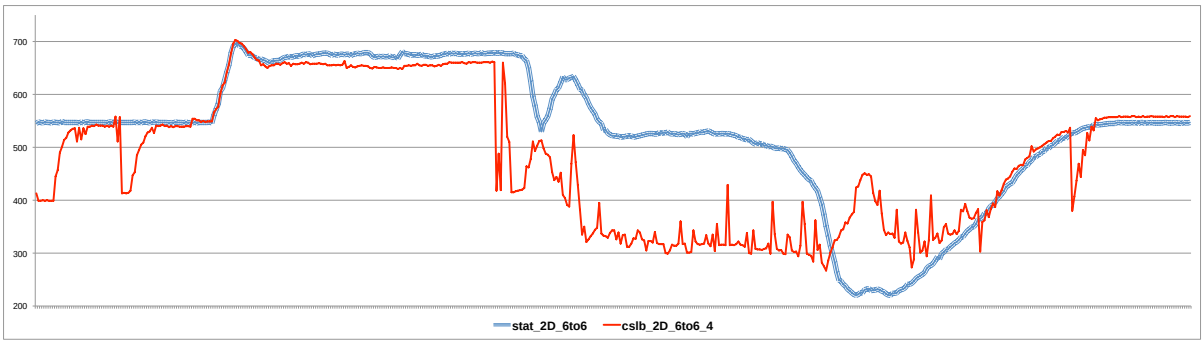
Moving the model to left or right regions will technically perform very similar to the first case in terms of load imbalance, as the nodes have equivalent power, each loads the whole of the model, and each is connected similarly to a switch for networking which would not put any to advantage over another based on locality. That is why we did not extend our camera path to include such a specific case. Figure 4(a), 4(b), 4(c) display three ranges of frames from our camera path, that places the model in different zones of the display region, which one can observe in the behavior of the system through the frame timing graphs of Figures 4(e) and 4(f).



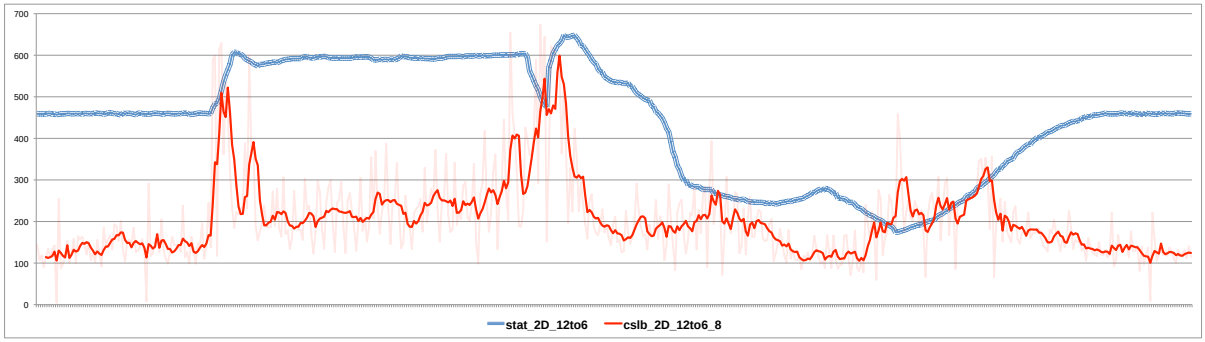
(a) Frames 1 to 345. (b) Frames 346 to 466. (c) Frames 467 to 540.



(d) Pure rendering timings of segments for the static *stat_2D_6to6* setup.



(e) Frame times (ms) for static 6-to-6 compared to best performing CSLB 6-to-6 configuration for david1mm.



(f) Frame times (ms) for static 12-to-6 compared to best performing CSLB 12-to-6 configuration for david1mm.

Figure 4: *One can easily observe the uneven workload distribution of a static tile setup in (d), whereas cross-segment load balancing can outperform static task distribution especially in the regions where load imbalance is high due to the views of the model, (e) and (f).*

Configuration	david2mm		lucy		david1mm	
	Overall FPS	*FPS 270–345*	*Overall FPS*	*FPS 270–345*	*Overall FPS*	*FPS 270–345*
stat_2D_6to6	37.612	35.055	3.611	2.524	1.882	1.918
cslb_2D_6to6_2	35.669	31.509	3.507	2.508	1.865	1.911
cslb_2D_6to6_4	24.289	22.170	4.821	3.818	2.078	3.106
cslb_2D_6to6_6	19.313	18.831	4.297	2.323	1.894	2.221
stat_2D_12to6	24.324	21.739	4.023	3.483	2.265	2.837
stat_2D_12to6_LB	25.366	24.382	5.945	5.729	1.918	2.090
cslb_2D_12to6_2	27.457	27.133	4.954	3.621	1.965	1.649
cslb_2D_12to6_4	26.984	26.518	9.636	8.052	2.514	3.260
cslb_2D_12to6_6	26.294	25.107	11.029	11.411	3.991	3.844
cslb_2D_12to6_8	20.204	19.948	6.254	7.409	4.886	5.170
cslb_2D_12to6_10	20.531	20.127	10.378	10.066	2.926	3.354
cslb_2D_12to6_12	18.165	17.483	9.564	10.594	3.117	3.395

Table 1: *Overall and partial* fps *results for three data sets.* CSLB *improvement is more visible for larger models.*

In Figure 4, based on *david1mm* test runs, we display frame timing graphs for the fastest *stat_** and *cslb_** configurations for 6 and 12 GPU setups respectively, underneath the images showing three frame ranges where the model occupancy is concentrated in different zones. A graph of pure rendering times for each of the six display GPUs of the *stat_2D_6to6* base run is presented alongside in Figure 4(d) to emphasize the uneven load distribution among segments in different situations, thus indicating the performance improvements made possible by cross-segment load balancing.

In Table 1, we list the *fps* rates of our test runs for all three data sets for all configurations. For each model, the left column provides average *fps* rates over the whole camera path, while the right column records averages for the frame range 270 to 345, during which the uneven distribution of load between segments of the display can easily be observed in Figure 4(d).

Our results show that multi-display setups can experience serious load imbalance which will cause the performance to suffer, and cross-segment load balancing improves the overall performance remarkably by making use of idle resources to help overloaded segments. We have seen that smaller models like *david2mm*, which can be rendered very easily, benefit much less, or even suffer, from CSLB due to overhead costs like network communication. Also, when too many extra resources are shared and assigned, communication can cause a bottleneck due to network limits and affect the performance negatively, as seen in Table 1. When rendering time is very small, distributing rendering tasks to nonlocal resources will bring comparably high communication overhead for collecting rendered image tile data for compositing. However, *lucy* and *david1mm* results demonstrate how helpful our cross-segment load balancing system can be, achieving up to more than 2.74 times speed-up when the rendering time dominates communication overhead. The right column for *lucy* and *david1mm* models in Table 1 show that CSLB achieves even higher speeds as collaboration across segments removes the otherwise excessive load imbalance due to the positioning of the model.

5. Conclusion and Future Work

In this paper, we have presented a dynamic load balancing approach integrated into *Equalizer*, named *cross-segment load balancing* (CSLB), which is designed to share the graphics workload across segments of a multi-display system. Imbalance of rendering load across display segments is a very common situation in parallel rendering applications which achieve higher resolutions and bigger display areas through multiple displays built as an array of projectors or flat panel screens driven by a rendering cluster. Using simple frame timing statistics from past frames, CSLB assigns extra resources to overloaded segments, which share workload through dynamic task partitioning in image or object space, all transparent to the application that utilizes the rendering framework.

We have observed remarkable performance improvements in our test runs, although being limited by network resources for cases where a lot of extra resources need to communicate image data to each other. Nevertheless, when rendering costs are higher than communication overhead, dynamically balancing workload of resources across segments of a display proves to be an extremely efficient and useful approach. For very simple rendering tasks, though, overheads might dominate causing a decrease in performance. To overcome such problems, we plan to improve the CSLB algorithm by incorporating other parameters like overhead cost more effectively for better decision making. Similarly, a more complicated cost prediction strategy can make the task partitioning system better by adapting faster to changing conditions, thus

achieving better overall performance with higher throughput for a wider range of applications.

Due to the increased resource utilization, CSLB provides more stable frame rates regardless of the model distribution in screen space. Our tests have shown that the compositing overhead, in particular the network transport, counteracts this equalization effect in some cases. We plan to address this by using more effective compression techniques and faster network interconnects to reduce the compositing cost.

6. Acknowledgments

We would like to thank and acknowledge the the Digital Michelangelo Project and the Stanford 3D Scanning Repository for providing 3D test data sources to the research community. We would also like to thank Maxim Makhinya for his help with Equalizer. This work was supported in parts by the Swiss Commission for Technology and Innovation (KTI/CTI) under Grant 9394.2 PFES-ES and the Swiss National Science Foundation under Grant 200020-129525.

References

[ACCC04] ABRAHAM F., CELES W., CERQUEIRA R., CAMPOS J. L.: A load-balancing strategy for sort-first distributed rendering. In *Proceedings SIBGRAPI* (2004), pp. 292–299.

[BHPB03] BETHEL W. E., HUMPHREYS G., PAUL B., BREDERSON J. D.: Sort-first, distributed memory parallel visualization and rendering. In *Proceedings IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2003), pp. 41–50.

[BJH*01] BIERBAUM A., JUST C., HARTLING P., MEINERT K., BAKER A., CRUZ-NEIRA C.: VR Juggler: A virtual platform for virtual reality application development. In *Proceedings IEEE Virtual Reality* (2001), pp. 89–96.

[BRE05] BHANIRAMKA P., ROBERT P. C. D., EILEMANN S.: OpenGL Multipipe SDK: A toolkit for scalable parallel rendering. In *Proceedings IEEE Visualization* (2005), pp. 119–126.

[CKS02] CORREA W. T., KLOSOWSKI J. T., SILVA C. T.: Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization* (2002), pp. 89–96.

[CM06] CAVIN X., MION C.: Pipelined sort-last rendering: Scalability, performance and beyond. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization Short Papers* (2006).

[CMF05] CAVIN X., MION C., FILBOIS A.: COTS cluster-based sort-last rendering: Performance evaluation and pipelined implementation. In *Proceedings IEEE Visualization* (2005), Computer Society Press, pp. 111–118.

[DK11] DOERR K.-U., KUESTER F.: CGLX: A scalable, high-performance visualization framework for networked display environments. *IEEE Transactions on Visualization and Computer Graphics 17*, 3 (March 2011), 320–332.

[EMP09] EILEMANN S., MAKHINYA M., PAJAROLA R.: Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics 15*, 3 (May/June 2009), 436–452.

[FCS*10] FOGAL T., CHILDS H., SHANKAR S., KRÜGER J., BERGERON R. D., HATCHER P.: Large data visualization on distributed memory multi-GPU clusters. In *Proceedings ACM SIGGRAPH/Eurographics Symposium on High-Performance Graphics* (2010), pp. 57–66.

[GMBP10] GOSWAMI P., MAKHINYA M., BÖSCH J., PAJAROLA R.: Scalable parallel out-of-core terrain rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization* (2010), pp. 63–71.

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P. D., KLOSOWSKI J. T.: Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics 21*, 3 (2002), 693–702.

[HXS09] HUI C., XIAOYONG L., SHULING D.: A dynamic load balancing algorithm for sort-first rendering clusters. In *Proceedings IEEE International Conference on Computer Science and Information Technology* (2009), pp. 515–519.

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4 (1994), 23–32.

[MMD06] MARCHESIN S., MONGENET C., DISCHLER J.-M.: Dynamic load balancing for parallel volume rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization* (2006), pp. 43–50.

[MWMS07] MOLONEY B., WEISKOPF D., MÖLLER T., STRENGERT M.: Scalable sort-first parallel direct volume rendering with dynamic load balancing. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization* (2007), pp. 45–52.

[MWP01] MORELAND K., WYLIE B., PAVLAKOS C.: Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics* (Piscataway, NJ, USA, 2001), PVG '01, IEEE Press, pp. 85–92.

[OA02] OSMAN A., AMMAR H.: Dynamic load balancing strategies for parallel computers. *Scientific Annals of Cuza University 11* (2002), 110–120.

[RLRS09] REPPLINGER M., LÖFFLER A., RUBINSTEIN D., SLUSALLEK P.: DRONE: A flexible framework for distributed rendering and display. In *Proceedings International Symposium on Visual Computing* (2009), Lecture Notes in Computer Science, pp. 975–986.

[SFLS00a] SAMANTA R., FUNKHOUSER T., LI K., SINGH J. P.: Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *Proceedings Eurographics Workshop on Graphics Hardware* (2000), pp. 97–108.

[SFLS00b] SAMANTA R., FUNKHOUSER T., LI K., SINGH J. P.: Sort-first parallel rendering with a cluster of PCs. In *ACM SIGGRAPH Sketches* (2000).

[SWNH03] STAADT O. G., WALKER J., NUBER C., HAMANN B.: A survey and performance analysis of software platforms for interactive cluster-based multi-screen rendering. In *Proceedings Eurographics Workshop on Virtual Environments* (2003), pp. 261–270.

[SZF*99] SAMANTA R., ZHENG J., FUNKHOUSER T., LI K., SINGH J. P.: Load balancing for multi-projector rendering systems. In *Proceedings Eurographics Workshop on Graphics Hardware* (1999), pp. 107–116.