University of Zurich[UZH]

David Steiner
Enrique Gonzalez Paredes
Stefan Eilemann
Fatih Erol
R Pajarola

Dynamic Work Packages in Parallel Rendering

TECHNICAL REPORT – No. IFI-2015.01

2015

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

David Steiner, Enrique Gonzalez Paredes, Stefan Eilemann, Fatih Erol, R Pajarola
Dynamic Work Packages in Parallel Rendering
Technical Report No. IFI-2015.01
Visualization and Multimedia Lab
Department of Informatics (IFI)
University of Zurich
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland
http://vmml.ifi.uzh.ch/

Dynamic Work Packages in Parallel Rendering

David Steiner* Enrique G. Paredes Stefan Eilemann† Fatih Erol Renato Pajarola

Visualization and MultiMedia Lab,
Department of Informatics,
University of Zurich

Blue Brain Project, EPFL

Technical Report IFI-2015.04, Department of Informatics, University of Zurich

ABSTRACT

Interactive visualizations of large-scale datasets can greatly benefit from parallel rendering on a cluster with hardware accelerated graphics by assigning all rendering client nodes a fair amount of work each. However, interactivity regularly causes unpredictable distribution of workload, especially on large tiled displays. This requires a dynamic approach to adapt scheduling of rendering tasks to clients, while also considering data-locality to avoid expensive I/O operations. This article discusses a dynamic parallel rendering load balancing method based on work packages which define rendering tasks. In the presented system, the nodes pull work packages from a centralized queue that employs a locality-aware affinity model for work package assignment. Our method allows for fully adaptive intra-frame workload distribution for both sort-first and sort-last parallel rendering.

1 INTRODUCTION

Research into parallel algorithms and techniques that exploit multiple computational resources in parallel to work towards solving a single large complex problem together has pushed the boundaries of the physical limitations of hardware to cope with ever growing computational problems. While reducing the workload of a single computational unit with parallelism in data or task space, making use of distributed parallel computers brings its own set of issues that need to be addressed for proper functioning of the system. Among the main challenges is the stringent requirement for optimization of partitioning and distribution of tasks to resources with consideration of minimal communication and I/O overheads.

With the dramatic increase of parallel computing and graphics resources through the expansion of multi-core CPUs, the increasing level of many-core GPUs as well as the growing deployment of clusters, scalable parallelism is easily achievable on the hardware level. In a number of application domains such as computational sciences the utilization of multiple or many compute units is nowadays commonplace. Also modern operating systems and desktop application programs more and more exploit the use of multiple CPU cores to improve their performance. Moreover, GPUs are increasingly used to speed up various computationally intensive tasks.

Thus with custom off-the-shelf hardware components affordable computer clusters can be built using open source software [12], which increases the availability of such systems for research in parallel computing. This growing deployment of computer clusters along with the dramatic increase of parallel computing resources has also been exploited in the computer graphics domain for demanding visualization applications, where GPUs are exploited using their data parallel many-core architecture. The combination of cost-effective and integrated parallelism at the hardware level as well as widely supported open source clustering software, has established graphics clusters as a commonplace infrastructure for development of more efficient algorithms for visualization as well as generic platforms that provide a framework for parallelization of graphics applications.

Not unlike other cluster computing systems, parallel graphics systems experience the need to improve efficiency in access to data and communication to other cluster nodes, while achieving optimal parallelism through a most favorable partitioning and assignment of rendering tasks to resources. Parallel rendering adopts approaches to job scheduling similar to the distributed computing domain, and adapts them to perform a well balanced partitioning and scheduling of workload under the conditions governed by the graphics rendering pipeline and specific graphics algorithms. Whereas some applications can be parallelized more easily with a statical a-priori distribution of tasks to the available resources, many real-time 3D graphics applications require a dynamically adapted scheduling mechanism to compensate for varying rendering workloads on different resources for fair utilization and better performance. This article explores a dynamic implicit load balancing approach for interactive visualization within the parallel rendering framework *Equalizer*, comparing and analyzing the performance improvements of a task pulling mechanism against available static and dynamic explicit task pushing schemes integrated in the same framework.

The following Section 2 provides an overview of terminology and related work in parallel rendering. Section 3 outlines the properties of the used parallel rendering framework and describes the details of our dynamic load balancing approach using work packages. After an analysis of test results in Section 4, a summary and ideas for future improvements conclude the article in Section 5.

2 RELATED WORK

With respect to Molnar's parallel-rendering taxonomy [19] on the sorting stage in parallel rendering, as shown in Figure 1, we can identify three main categories of single-frame parallelization modes: sort-first (image-space) decomposition divides the screen space and assigns the resulting tiles to different render processes; sort-last (object-space) does a data domain decomposition of the 3D data across the rendering processes; and sort-middle redistributes parallel processed geometry to different rasterization units.

While GPUs internally optimize the sort-middle mechanism for tightly integrated and massively parallel vertex and fragment processing units, this approach is not feasible for parallelism on a higher level. In particular, driving multiple GPUs distributed across a network of a cluster does not lend itself to an efficient sort-middle solution as it would require interception and redistribution of the transformed and projected geometry (in scan-space) after primitive assembly. Hence we treat each GPU as one unit capable of process-

*e-mail: {steiner, egparedes, erol, pajarola}@ifi.uzh.ch
†e-mail: stefan.eilemann@epfl.ch

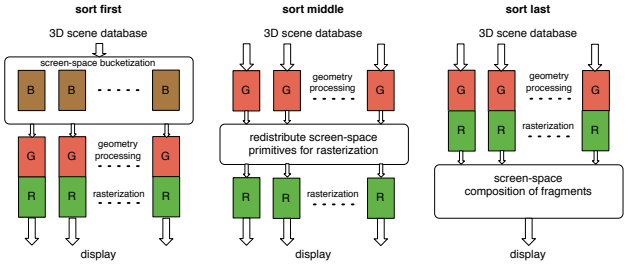


Figure 1: Sort-first, sort-middle and sort-last parallel rendering workflow.

ing geometry and fragments at some fixed rate, and address load balancing of multiple GPUs in a cluster system on a higher level using sort-first or sort-last parallel rendering.

2.1 Parallel Rendering Systems

Besides many application-specific solutions for parallelization on multiple GPUs, few generic frameworks have been proposed to provide an interface for executing visualization applications on distributed systems. One class of such approaches is OpenGL intercepting libraries, which are highly transparent solutions that only require replacing OpenGL libraries with their implementations. The replaced libraries intercept all rendering calls, and forward them to appropriate target GPUs according to different configurations of a cluster of nodes. The Chromium [15] approach can be configured for different setups but often exhibits severe scalability bottlenecks due to streaming of calls to multiple nodes generally through a single node. Follow up systems such as CGLX [8] and ClusterGL [22] try to reduce the network load primarily through compression, frame differencing and multi-casting but retain the principle structural bottlenecks.

Another approach for rendering parallelization takes a more intrusive way, by enforcing specific data structures like scene graphs. Popular scene graph implementations OpenSceneGraph [5, 27] or OpenSG [23, 28] expose interfaces for running on multiple GPUs or a cluster, as well as DRONE [24] and tinySceneGraph [18]. These approaches typically concentrate their functionality on distributed scene graph synchronization and updating rather than on performance scalability, and generally require full data duplication across all nodes.

More generic platforms support flexible resource configurations and shield the developer from most of the complexity of the distributed and networked cluster-parallel system. VRJuggler [4] targets the configuration of immersive VR applications, however, it too suffers from scalability limitations. OpenGL Multipipe SDK [16, 3] implements a callback layer for an effective parallelization, but only for shared memory multi-CPU/GPU systems. IceT [21] represents a system for sort-last parallel rendering of large datasets on tiled displays, focusing specifically on image composition strategies. LOTUS [7] on the other hand is a system which focuses on configurable virtual environments on cluster-based tiled displays.

In contrast to these other approaches, Equalizer [9] represents a unique solution that is both oriented towards scalable parallel rendering as well as flexible task decomposition and resource configuration (see also Figure 2). It supports a fully distributed architecture with network synchronization, generic distributed objects and a large set of parallel rendering features combined with load balancing. Due to its flexibility and supported features, the dynamic work packages load balancing method presented here has been implemented and evaluated within this framework.

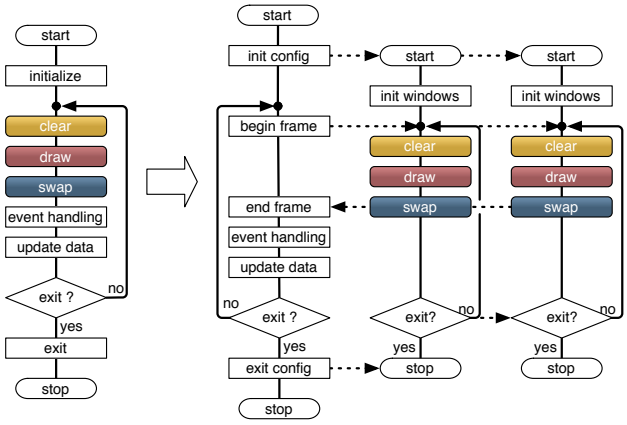


Figure 2: Overview of Equalizer server driving rendering clients based on a resource usage configuration file.

2.2 Load Balancing

Distributing work to multiple resources can improve the performance of an application in general, however, the relationship between the number of resources and performance speed-up is rarely linear. As Amdahl has recognized [2], an application always contains some limiting sequential non-parallelizable as well as overhead code, for synchronization and setting up the parallel tasks. Furthermore, the work between the parallel workers needs to be balanced for optimal speedup, which is rarely easy for real-time visualization applications. The cost of a partitioned task varies over time, e.g., when a displayed model is transformed on screen due to user interaction, different amounts of polygons are to be rendered for different parts of the screen. Dynamic load-balancing of tasks and assigning them to the most appropriate resources is used to achieve a better resource utilization.

Dynamic load balancing can be defined as partitioning and scheduling the work to equalize resource utilization for better overall performance. The task of rendering an image can be partitioned within instruction or data space, i.e., into computational units of execution or subsets of data to be processed, respectively. Moreover, parameters like dependencies between tasks, priorities, locality of the data should be observed while designing a load balancing algorithm. Moreover, computing the task decomposition itself should not demand a lot of resources, since it typically is a sequential portion of the code as per Amdahl's classification.

Various approaches to assign and load balance tasks for multiple resources have been proposed. In the following we will focus primarily on interactive cluster parallel rendering and specifically on dynamic load balancing of sort-first and sort-last parallel rendering on cluster systems. In distributed parallel rendering it is important that the workload task partitioning dynamically adjusts to heterogeneous resources, I/O and communication costs, as well as varying data dependencies and rendering costs.

We can classify load-balancing into *explicit* and *implicit* approaches, where explicit methods centrally compute a task decomposition up-front, before a new frame is rendered, while implicit methods decompose the workload into task units that can dynamically be assigned to the resources during rendering, based on the work progress of the individual resources. Explicit load-balancing can be reactive, based on load distribution in previous frames, or predictive, based on an application-provided cost function. Explicit load-balancing typically assigns a single task to each resource to minimize static per-task costs. Implicit load-balancing generally uses a finer granularity of many more task units than resources to minimize the load imbalance due to a fixed coarse task granularity.

Implicit load-balancing uses central task distribution or distributed task stealing between resources. We therefore propose a classification of load-balancing methods into *reactive explicit*, *predictive explicit*, *centralized implicit* and *distributed implicit*.

In [26], the fundamental concepts of adaptive sort-first screen partitioning and various explicit load-balancing schemes have been introduced, and experimental evidence that the a single task per resource leads to the best performance has been presented. In [25], a predictive explicit approach is used for hybrid sort-first/sort-last parallel rendering. Past-frame rendering time is proposed as a simple yet effective cost heuristic for a reactive explicit algorithm in [1]. Pixel-based rendering cost estimation and kd-tree screen partitioning are used in [20] for improved predictive explicit sort-first parallel volume rendering. Similarly per-pixel vertex and fragment processing cost estimation and adaptive screen partitioning is proposed in [14]. A reactive explicit load-balancing algorithm for multi-display visualization system was further proposed in [10].

Implicit algorithms are more commonly used for off-line raytracing compared to real-time rasterization algorithms due to the practically non-existent per-tile cost in raytracing. In [13], both predictive explicit and implicit algorithms are proposed and compared, and implicit algorithms are shown to be superior for raytracing. In [17], centralized and distributed implicit load-balancing algorithms are compared for radiosity rendering. Centralized implicit algorithms for modern, highly parallel graphics processors are proposed in [6].

Implicit dynamic load-balancing methods for real-time distributed cluster-parallel rendering, however, have not yet been addressed in the research community, and this paper provides a first step and experiments in this direction. The main differentiation from prior work includes

1. a novel implicit rendering task partitioning approach, using
2. a parallel rendering work package and a task pulling mechanism, as well as
3. the introduction of an affinity model for scoring the mapping of tasks to resources.

3 Dynamic Load Balancing

Dynamic load balancing systems must either be able to *a priori* assess the cost of the workload as accurately as possible and decompose it as evenly as possible for explicit task partitioning, or otherwise have flexible granular work units that can dynamically be assigned to the various available resources for implicit task partitioning. In the former, accurately assessing the rendering cost of some given 3D graphics data under a given viewing and illumination configuration, as well as deriving cost-uniform work partitions is non-trivial and can be costly for real-time rendering. Hence, under the assumption of strong temporal frame-to-frame coherence, most approaches use fairly simple previous-frame rendering times and statistics to approximate the expected current frame rendering cost, and correspondingly, adjust the previous rendering task decomposition explicitly before starting to render the frame. However, our *implicit* load balancing approach does neither, allowing for adaptive balancing of workload during the rendering of a single frame, and thus being able to adapt to variable graphics resources even once the work decomposition has been defined (see Figure 3).

Therefore, in this work we explore a flexible implicit load balancing approach (see Figure 3(c)) and exploit the concept of *rendering work packages* as outlined in Figure 4. This allows for a quick-start setup with initial work package assignments, as well as subsequent dynamic (re)allocation of work packages to rendering resources that are ready for more work.

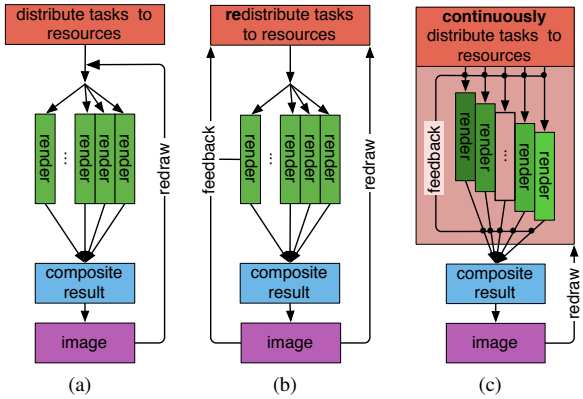


Figure 3: *(a)* Static versus *(b)* an explicit dynamic load balancing that can adjust task decomposition between frames, and *(c)* fully adaptive implicit workload distribution.

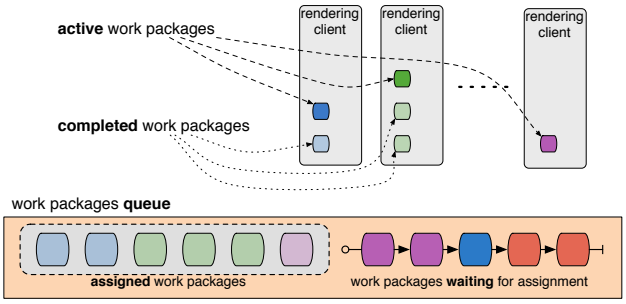


Figure 4: Dynamic load balancing distributed parallel rendering using work packages.

3.1 Parallel Rendering Work Packages

As a generic platform, *Equalizer* supports various modes of rendering task parallelization. A server configuration file declares the available resources (besides automatic detection possibilities), and allows for a flexible description of resource usage, determining the distribution of rendering tasks as well as final image composition (see also Figure 5). The rendering tasks can even be decomposed hierarchically into partitions in the sort-first image or sort-last data space. Moreover, separate eye passes can be assigned to different resources for multi-view visualization, nodes can be chosen to render consecutive frames for a smoother frame rate, as well as sample subpixels for antialiasing.

Focusing on sort-first and sort-last parallel rendering, Equalizer already supports explicit dynamic load balancing in both image and data space by redistributing rendering tasks, based on previous frame time statistics. To further improve resource utilization, one could use a *task pulling* mechanism, an approach that has been employed before in distributed computing. We explore this approach in this work with a *dynamic work packages* implementation within the Equalizer framework. Rather than having the server push tasks to the rendering clients, our dynamic work packages approach works by managing fine grained tasks on the server side, while the clients request and execute the tasks as they become available.

Every rendering client employs a local queue of work packages for caching purposes. During rendering, a client first works on packages from its local queue and requests n_{req} packages from the server whenever the amount of available packages sinks below some n_{min}. According to the employed affinity model, the server will respond with at maximum n_{req} work packages most suitable

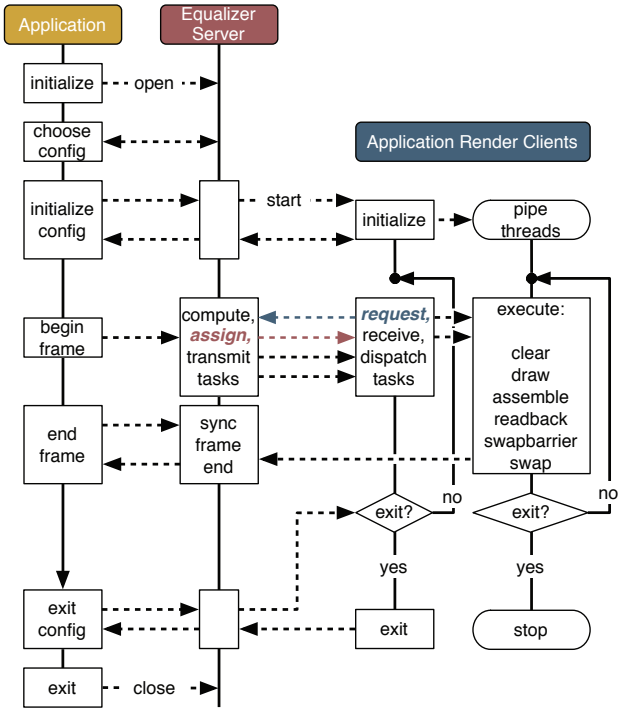


Figure 5: Simplified execution flow of an Equalizer application using our work packages method. Note that clients request work packages from the server, which in turn assigns the packages to the respective client nodes, establishing a work-assignment loop that ends when all packages have been processed, finishing the current frame.

for the requesting client. The client then adds these to its local queue.

The work packages used in our system relate to small, uniformly-sized partitions in object-data or image space. At the beginning of each frame, the server generates the descriptions for all n_{total} required work packages (i.e. regions in image space or index ranges in geometry space) and stores them in an indexed map $\mathscr{M}$. A work package is associated with, and can be retrieved from $\mathscr{M}$ with a key $k \in [0,1]$ that is based on an *affinity* model in either image or data space, as further detailed in the next section.

The key k is calculated from the package's index i and the total number of available packages n_{total} for the current frame as $k = \frac{i}{n_{total}}$. Given the appropriate affinity model, this corresponds to a locality-preserving mapping from data or image space to our work-package key space.

3.2 Work Packages Affinity

For work-package to rendering-node assignments, each rendering node is associated with a linear position $p \in [0,1]$ too, and given a position p, the closest work package $m(p)$ to this position is retrieved from all available ones in $\mathscr{M}$ according to Eq. 2. Here we use a circular addressing scheme, that utilizes a distance function d as defined in Eq. 1, which is exploited in the affinity model as further described below.

$$d(p,x) = min(|1-p+x|, |p-x|, |1-x+p|) \quad (1)$$

$$m(p) = \underset{x \in \mathscr{M}}{\operatorname{argmin}} \{d(p,x)\} \quad (2)$$

To allow the server to select the most suitable set of work packages to serve a given client request, we propose a data-locality and work-load aware affinity model. As work packages are mapped to positions $k \in [0,1]$ in our key space, requesting client nodes are associated with this space as well. In this work package key and node index space we define our affinity model and mapping. The key is to achieve a linear work package mapping that will eventually exploit data locality on rendering clients under a dynamic work package allocation process.

To establish a data-locality preserving work package affinity, we first-most must have a locality preserving linear mapping of the work packages and their data to our linear map $\mathscr{M}$. For both, object-space data as well as image-space screen partitioning, space filling curves (SFCs) offer a locality preserving linear mapping, as illustrated in Figure 6. The z-curve as shown in Figure 6(a), e.g., can be used to map work packages of an object-space 3D data partitioning to linear indices $k \in [0,1]$. For this, the 3D geometry data is arranged and grouped along a 3D SFC. The data locality in sort-last rendering is now achieved as follows: Given that an initial data package k_0 is assigned to a certain rendering node, the work packages $k_0 \pm 1$ will contain spatially close geometry. Thus assigning more data packages close to k_0 to the same rendering node will be favorable due to less random memory accesses, and hence improved pre-fetching and caching benefits. Furthermore, nearby data work packages will be rendered to nearby regions on screen as well, thus further benefits in image compositing may be possible.

Mapping the tiles of an image-space screen partitioning to the linear indices $k \in [0,1]$ of a 2D SFC, together with a spatial locality preserving linearization of the 3D data, data locality in sort-first rendering can also be achieved, as shown in Figure 6(b). The rendering of nearby tiles $k_0 \pm 1$ from the starting tile k_0 of a rendering node, will require further 3D data that is spatially close to the geometry already rendered for tile k_0. Thus locality is also preserved with respect to memory access, and further benefits may arise in the per-tile view-frustum culling stage.

In general, our affinity model then works such that the server maps each client to a position $p \in [0,1]$ in key space and always responds with work packages available from $\mathscr{M}$ closest to p, according to Eq. 2, which are subsequently removed from the map $\mathscr{M}$. Our mapping rules result in client positions and node boundaries continuously being updated as clients consume work packages, which is illustrated by Figure 7. Initially, clients and work packages are being mapped to key space in an equidistant fashion, as shown in Figure 7(a). As packages are consumed, the server continuously updates the boundaries between clients (*node boundaries*), based on the ratio of the work-package consumption rate between neighboring client nodes, as illustrated by Figure 7(b). Subsequently, the server re-centers client positions between adjacent node boundaries, which is shown in Figure 7(c). This has the effect that clients that are faster at consuming work packages will tend to move towards their slower neighbors, eventually consuming packages originally associated with these.

To preserve locality, the server only removes and assigns packages if their distance to the client's position p in key space fulfills the following condition:

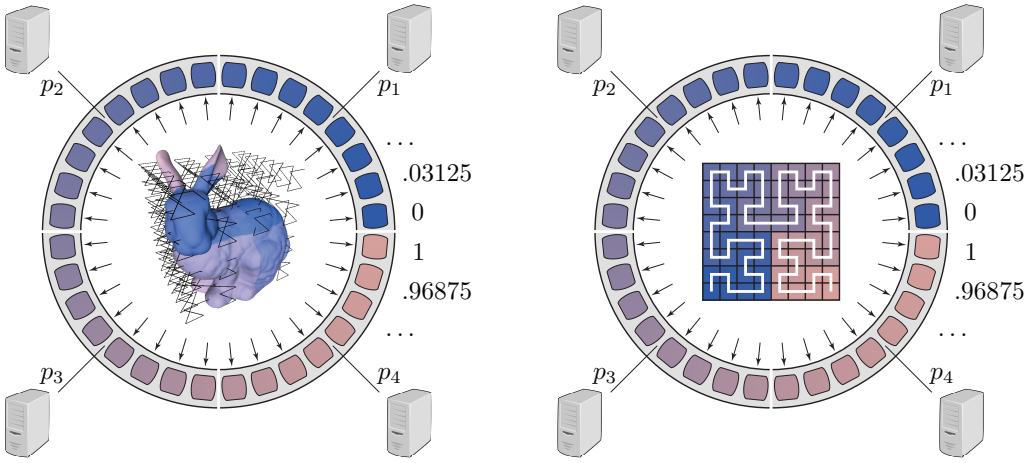
$$d(p,x) \leq d' \quad \text{with} \quad d' = \begin{cases} d(p_{prev},x), & \text{for } x \leq p \\ d(p_{next},x) & \text{for } x > p \end{cases} \quad (3)$$

where x is the position of a candidate package, and p_{prev} and p_{next} are the previous and next client's positions from p in key space respectively.

To adjust for load imbalances, node positions within key space are constantly updated, according to the amount of packages they have consumed in relation to each other, within a time window w. I.e., the number of packages used to calculate a client's position is

$$n(p) = 1 + s(p,w) \quad (4)$$

where $s(p,w)$ is the sum of packages the node at position p received within the last w time steps. Please note that these time steps are not



(a) 3D mapping (b) 2D mapping

Figure 6: Mapping from object/image space to our one-dimensional key space using a space-filling curve. In example (a), object-space geometry segments are mapped to work packages using a 3D z-curve. In (b), screen-space tiles are mapped to work packages using a 2D Hilbert curve. Darker colors indicate a lower, lighter colors a higher position in key space (gray circle). The two lowest and the two highest work package positions are written on the right-hand side. Please note that the successor of the work package at position 1 is at position 0, due to circular indexing. Four rendering clients are mapped to key space positions $p_1 = .125$, $p_2 = .375$, $p_3 = .625$, and $p_4 = .875$.

dependent on frame boundaries but are currently defined as interval between two package requests being served.

The function $n(p)$ can be used to calculate boundaries between the nodes in key space as a weighted sum of neighboring node positions, based on the associated nodes' package consumption. Before serving a request, the server calculates the boundary b between a client at position p and its successor p_{next} in key space as follows, see Figure 7(b) for an illustration of the resulting change in node boundaries:

$$b(p, p_{next}) = \frac{n(p)p_{next} + n(p_{next})p}{n(p) + n(p_{next})} \tag{5}$$

The server then repositions every client in key space by centering it between the new adjacent node boundaries, as shown in Figure 7(c):

$$p_{new} = \frac{b(p_{prev}, p) + b(p, p_{next})}{2} \tag{6}$$

where p is the old client position, and p_{prev}, and p_{next} are the positions of its neighboring nodes in key space.

The role of server and client in creating and distributing work packages can be simplified and summarized as follows in Algorithm 1 and Algorithm 2, respectively. Note that their respective roles within the *Equalizer* platform are illustrated in Figure 5.

Algorithm 1 Role of the server (simplified)

```
1:  while running do
2:      Start frame
3:      let nodes be a list of all clients
4:      Generate package indices and spatial positions from SFC
5:      let n_total be the number of all available packages
6:      for each package x do
7:          k ← x.index / n_total
8:          Insert x into 𝒜 at k
9:      end for
10:     Handle package requests
11: end while
```

More specifically, package request handling on the server is summarized in Algorithm 3.

Algorithm 2 Role of the client (simplified)

```
1:  while rendering frame do
2:      let n_total be the number of locally available packages
3:      if n_total < n_min then
4:          Request n packages
5:      end if
6:      Process server response
7:      if no more packages exist on server then
8:          Stop rendering frame
9:      end if
10:     for each local package x do
11:         Draw x
12:         Process and transmit result
13:     end for
14: end while
```

Algorithm 3 Package request handling

```
1:  procedure HandlePackageRequest(node, n_req)
2:      Calculate boundaries between all node positions
3:      Calculate new positions for all nodes from boundaries
4:      let p_new be the new position of node
5:      let packages be an empty list of work packages
6:      Update d'                                   ▷ see Eq. 3
7:      package ← m(p_new)                          ▷ see Eq. 1
8:      while d(p_new, package.position) ≤ d' and n_total > 0 do
9:          Add package to packages
10:         Remove package from 𝒜
11:         package ← m(p_new)
12:         Update n_total
13:         if packages.count ≥ n_req then
14:             return packages
15:         end if
16:     end while
17:     return packages
18: end procedure
```

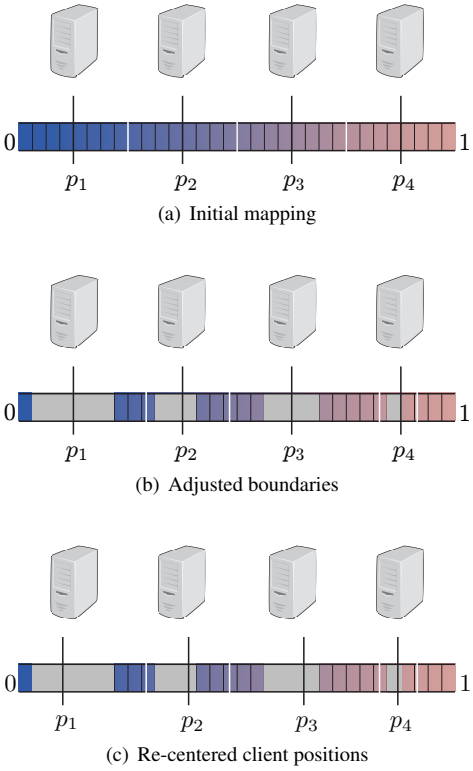


Figure 7: Mapping of rendering clients and work packages to key space at different stages. The example shows four clients at positions $p_{1...4}$. Node boundaries are indicated by white gaps, work packages by slabs, stacked from left (position 0) to right (position 1). The color of these slabs corresponds to work package positions in key space. Grey areas are empty due to client work package consumption. Figure (a) shows the initial mapping, (b) shows node boundaries that have been re-calculated based on client work package consumption, (c) shows client positions re-centered between node boundaries.

4 Performance Analysis

We performed preliminary tests of our system on a multi-GPU graphics workstation with two 8-core *Intel Xeon E5-2650* CPUs, 32*GB* DDR3 RAM, and four *nVidia GeForce GTX 970* GPUs with 4*GB* of VRAM each. We simulated the usage of four independent rendering nodes by running four Equalizer client processes on the machine, each utilizing one GPU.

The data set for conducting our basic tests, *David1mm*, has 28*M* vertices. In order to simulate complex user-induced movement, we let the model quickly rotate along its major axis while moving horizontally across the screen, being very close to the viewer (see also example view in Figure 8).

We compared our implicit dynamic load balancing method with traditional sort-first and sort-last dynamic load balancing that is reliant on frame-to-frame coherence. Within the Equalizer platform, these have already been implemented as *2D Load Equalizer* and *DB Load Equalizer*, respectively [11]. We implemented our method within the Equalizer platform as *Package Equalizer* and tested it with a sort-first configuration of 8*x*8 tiles in screen space, and a sort-last configuration of 8 segments of 3D data in object space.

We further implemented two naive affinity models for comparison to the data-locality and work-load aware model that we propose. The *Equal* affinity model simply segments the key space into constant, equally-sized ranges of work packages and assigns each client to one of these for the entire duration of program execution. The *FCFS* affinity model, conversely, simply maintains a list of work packages and dynamically assigns any requesting client with the first package available in a first-come-first-served fashion.

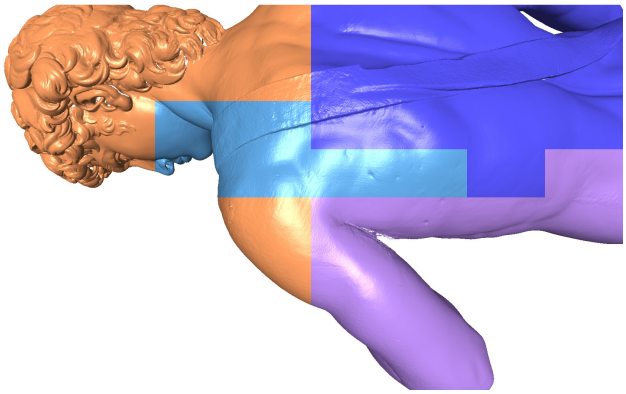
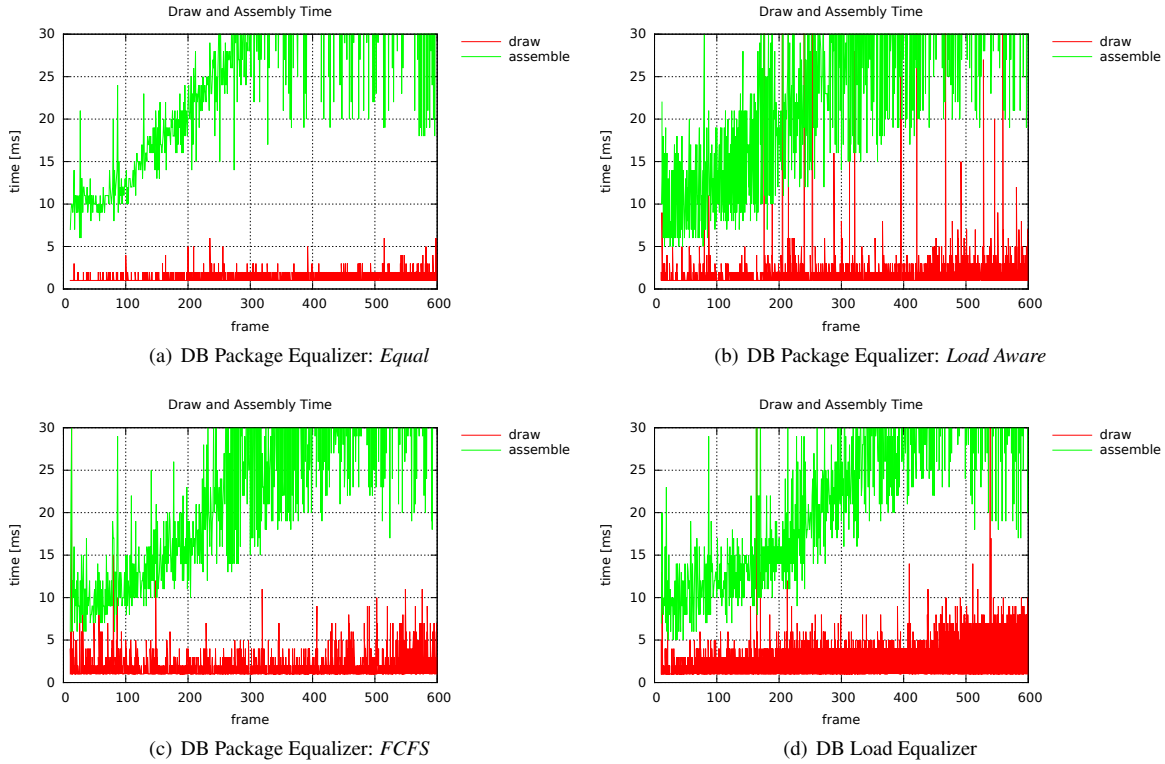


Figure 8: Work-package-based rendering of the *David1mm* model in sort-first configuration. Screen space has been segmented into tiles following a 2D Hilbert curve, according to Figure 6(b). Each tile is a work package processed by one of four rendering clients, whereas a tile's color corresponds to the processing node, for illustration purposes. Due to the linear mapping process and the affinity model described in Section 3.2, this results in four screen space regions that vary in size due to load imbalances; each region is associated with a rendering client. Note that the regions on the left and bottom right are considerably larger than the remaining ones, presumably because these regions largely consist of empty space, hence posing less rendering load to the corresponding client nodes.

Our experiments as summarized in Table 1 show that in the given sort-last parallel rendering scenario, our method exhibits better overall performance than the traditional Load Equalizer method, considering both draw time and assembly time, as shown in Figure 9. Conversely, the performance of our method is lower when compared with Load Equalizer in the given sort-first scenario.

Method	**Affinity**	**Draw**	**Assembly**	**Total**
Package DB	Equal	13687	85159	98846
Package DB	FCFS	16936	86896	103832
Package DB	Load-aware	17976	95197	113173
Load DB	-	33579	92505	126084
Package 2D	Equal	39845	20227	60072
Package 2D	FCFS	48618	19379	67997
Package 2D	Load-aware	41467	11395	52862
Load 2D	-	33601	8044	41645

Table 1: Total draw and assembly time in milliseconds, as well as the sum of these timings for Load- and Package Equalizer in sort-first (2D) and sort-last (DB) configurations with three different affinity models. The values were calculated over the duration of 2390 frames.



(a) DB Package Equalizer: *Equal*

(b) DB Package Equalizer: *Load Aware*

(c) DB Package Equalizer: *FCFS*

(d) DB Load Equalizer

Figure 9: Draw and assembly times for rendering 600 frames of the rotating *David1mm* model with both Package Equalizer and Load Equalizer in sort-last configuration (DB).

5 Discussion and Conclusion

We presented a dynamic load balancing method for parallel rendering that is based on a novel implicit rendering task partitioning approach, using a work package pulling mechanism. We also introduced an affinity model for scoring the mapping of tasks to resources, using a dynamic mapping of clients and work packages to a linear space.

First tests of our dynamic work packages method, rendering the *David1mm* model over a camera path, revealed a performance advantage of our method over the traditional Load Equalizer, in our experimental sort-last configurations. We found that in the given scenario, our method needed less time for drawing and assembly than the traditional method. In the tested sort-first configurations, our method did not show any performance advantage. This is likely to be due to the non-parallelizable overhead associated with the processing of tasks in general [2]. Since the number of tiles created in our sort-first setup greatly exceeds the number of geometry segments created in a sort-last setup, a significant package handling overhead for sort-first configurations is to be expected. This shows that only GPU-bound applications will benefit from such a configuration. This is emphasized by the fact that the affinity model that consistently needed the lowest draw time with our method was the *Equal* model, which is essentially static in nature. This further shows that rendering itself posed little challenge for the strong graphics hardware in place, and seemed to benefit more from coherence and simplicity of rendering tasks than actual dynamic load distribution.

However, the higher performance of our method in experiments with sort-last configurations, in comparison with the traditional Load Equalizer, suggests that our dynamic load balancing method is more adaptive and better at cost prediction since it does not rely on potentially outdated statistics of previous frames. This seems to be emphasized by the fact that in these sort-last configurations, our proposed affinity model generally performs better than the implemented naive dynamic model, which is *FCFS*. Our affinity model also needs less draw time but not less assembly time in sort-first configurations, than the *FCFS* model. This is consistent with aforementioned observations of overhead costs in sort-first and sort-last configurations.

Simulating four independent rendering nodes on a strong graphics workstation gave a first impression of our method's potential. However, following experiments will be performed on a GPU-based cluster, on which effective dynamic load balancing is more critical. Also, to evaluate our method more accurately, truly GPU-bound test scenarios are required. To this end, we plan to perform detailed direct volume rendering (DVR) of high-resolution datasets, a common visualization problem that usually poses a great challenge to the GPU's fragment processing power, justifying the overhead incurred by our work packages approach, especially in sort-first configurations. Similarly, the rendering of more complex surface models, also using complex materials, will pose another suitable test scenario for our method, especially in sort-last rendering configurations.

Acknowledgements

This work was supported in part by the EU FP7 People Programme (Marie Curie Actions) under REA Grant Agreement n°290227. The authors would also like to thank and acknowledge the following institutions and projects for providing 3D test data sets: the Digital Michelangelo Project and the Stanford 3D Scanning Repository.

REFERENCES

[1] F. Abraham, W. Celes, R. Cerqueira, and J. L. Campos. A load-balancing strategy for sort-first distributed rendering. In *Proceedings SIBGRAPI*, pages 292–299, 2004.

[2] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *Proceedings American Federation of Information Processing Societies Joint Computer Conference*, volume 30, pages 483–485, 1967.

[3] P. Bhaniramka, P. C. D. Robert, and S. Eilemann. OpenGL Multipipe SDK: A toolkit for scalable parallel rendering. In *Proceedings IEEE Visualization*, pages 119–126, 2005.

[4] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. VR Juggler: A virtual platform for virtual reality application development. In *Proceedings of IEEE Virtual Reality*, pages 89–96, 2001.

[5] D. Burns and R. Osfield. OpenSceneGraph. http://www.openscenegraph.org/, 1998.

[6] D. Cederman and P. Tsigas. On dynamic load balancing on graphics processors. In *Proceedings ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 57–64, 2008.

[7] Y. Cho, M. Kim, and K. S. Park. LOTUS: Composing a multi-user interactive tiled display virtual environment. *The Visual Computer*, 28(1):99–109, 2012.

[8] K.-U. Doerr and F. Kuester. CGLX: A scalable, high-performance visualization framework for networked display environments. *IEEE Transactions on Visualization and Computer Graphics*, 17(2):320–332, March 2011.

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

[10] F. Erol, S. Eilemann, and R. Pajarola. Cross-segment load balancing in parallel rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*, pages 41–50, 2011.

[11] Eyescale Software GmbH. Load Equalizer. http://www.equalizergraphics.com/scalability/loadEqualizer.html, 2008.

[12] M. J. Galán, F. García, L. Álvarez, A. Ocón, and E. Rubio. 'Beowulf Cluster' for high-performance computing tasks at the university: A very profitable investment. high performance computing at low price. In *Proceedings International Conference of European University Information Systems on The Changing Universities - The Role of Technology*, pages 328–335, 2002.

[13] A. Heirich and J. Arvo. A Competitive Analysis of Load Balancing Strategies for Parallel Ray Tracing. *The Journal of Supercomputing*, 12(1-2):57–68, 1998.

[14] C. Hui, L. Xiaoyong, and D. Shuling. A dynamic load balancing algorithm for sort-first rendering clusters. In *Proceedings IEEE International Conference on Computer Science and Information Technology*, pages 515–519, 2009.

[15] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics*, 21(3):693–702, 2002.

[16] K. Jones, C. Danzer, J. Byrnes, K. Jacobson, P. Bouchaud, D. Courvoisier, S. Eilemann, and P. Robert. SGI®OpenGL Multipipe™SDK User's Guide. Technical Report 007-4239-004, Silicon Graphics, 2004.

[17] M. Korch and T. Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms. *Concurrency and Computation: Practice and Experience*, 16(1):1–47, 2004.

[18] C. Marten. tinySceneGraph. http://www.tinysg.de/, 2008.

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

[20] B. Moloney, D. Weiskopf, T. Möller, and M. Strengert. Scalable sort-first parallel direct volume rendering with dynamic load balancing. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*, pages 45–52, 2007.

[21] K. Moreland, B. N. Wylie, and C. J. Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 85–92. IEEE, 2001.

[22] B. Neal, P. Hunkin, and A. McGregor. Distributed OpenGL rendering in network bandwidth constrained environments. In T. Kuhlen, R. Pajarola, and K. Zhou, editors, *Proceedings Eurographics Conference on Parallel Graphics and Visualization*, pages 21–29. Eurographics Association, 2011.

[23] D. Reiners. OpenSG. http://www.opensg.org/, 1999.

[24] M. Repplinger, A. Löffler, D. Rubinstein, and P. Slusallek. DRONE: A flexible framework for distributed rendering and display. In *Proceedings International Symposium on Advances in Visual Computing: Part I*, volume 5875, pages 975–986. Springer-Verlag, 2009.

[25] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *Proceedings Eurographics Workshop on Graphics Hardware*, pages 97–108, 2000.

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

[27] T. Samuelsen. *Distributed OpenSceneGraph*. University of Tromsø, 2007.

[28] G. Voß, J. Behr, D. Reiners, and M. Roth. A multi-thread safe foundation for scene graphs and its extension to clusters. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*, pages 33–37. Eurographics Association, 2002.