Eurographics Symposium on Parallel Graphics and Visualization (2016)
W. Bethel, E. Gobbetti (Editors)

Dynamic Work Packages in Parallel Rendering

D. Steiner[†1], E. G. Paredes[1], S. Eilemann[‡2], R. Pajarola[1]

[1]Visualization and MultiMedia Lab, Department of Informatics, University of Zurich
[2]Blue Brain Project, EPFL

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 dynamic affinity model for work package assignment. Our method allows for fully adaptive implicit workload distribution for both sort-first and sort-last parallel rendering.

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

1. Introduction

Research in parallel computing that exploits computational resources concurrently to work towards solving a single large complex problem 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. Among the main challenges are the stringent requirement for optimization of task partitioning as well as 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 well supported 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 general computationally intensive tasks.

The 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 and rendering 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 available 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

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

DOI: 10.2312/pgv.20161185

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 *work packages* based task pulling mechanism against available 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 [MCEF94] 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.

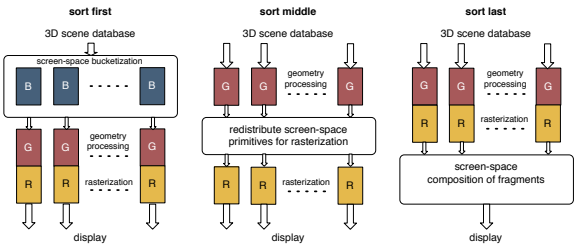


Figure 1: *Sort-first, sort-middle and sort-last parallel rendering workflow.*

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 processing geometry and fragments at some fixed rate, and address load balancing of multiple GPUs in a cluster system on a higher level, exploiting 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 are OpenGL intercepting libraries, which are highly transparent solutions that only require replacing OpenGL libraries with their implementations. These libraries intercept all rendering calls, and forward them to appropriate target GPUs according to different configurations of a cluster of nodes. The Chromium [HHN*02] 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 [DK11] and ClusterGL [NHM11] try to reduce the network load primarily through compression, frame differencing and multi-casting but retain the principle structural bottlenecks.

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 [BJH*01] targets the configuration of immersive VR applications, however, it too suffers from scalability limitations. OpenGL Multipipe SDK [JDB*04, BRE05] implements a callback layer for an effective parallelization, but only for shared memory multi-CPU/GPU systems. IceT [MWP01] represents a system for sort-last parallel rendering of large datasets on tiled displays, focusing specifically on image composition strategies. LOTUS [CKP12], 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 [EMP09] 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.

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 [Amd67], an application always contains some limiting sequential non-parallelizable as well as overhead code,

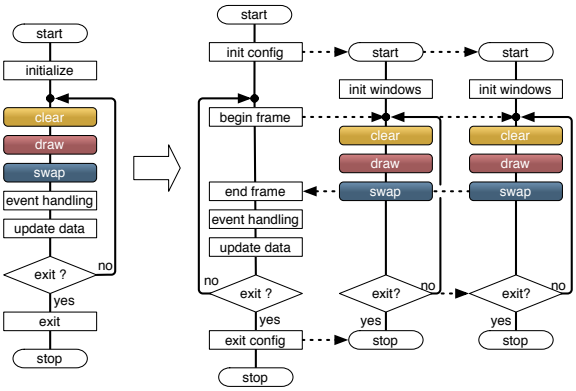


Figure 2: *Overview of Equalizer server driving rendering clients based on a resource usage configuration file.*

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 graphics 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, but doing so will impose a larger per-task overhead cost. Implicit load-balancing may use central task distribution or apply 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 [SZF*99], the fundamental concepts of adaptive sort-first screen partitioning and various explicit load-balancing schemes have been introduced, and experimental evidence that a single task per resource leads to the best performance has been presented. In [SFLS00], 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 [ACCC04]. Pixel-based rendering cost estimation and kd-tree screen partitioning are used in [MWMS07] 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 [HXS09]. A reactive explicit load-balancing algorithm for a multi-display visualization system was further proposed in [EEP11].

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 [HA98], both predictive explicit and implicit algorithms are proposed and compared, and implicit algorithms are shown to be superior for raytracing. In [KR04], centralized and distributed implicit load-balancing algorithms are compared for radiosity rendering. Centralized implicit algorithms for modern, highly parallel graphics processors are proposed in [CT08].

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 a dynamic 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 time statistics to approximate the expected current frame rendering cost, and correspondingly, adjust the previous rendering task decomposition explicitly before starting to render a new 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).

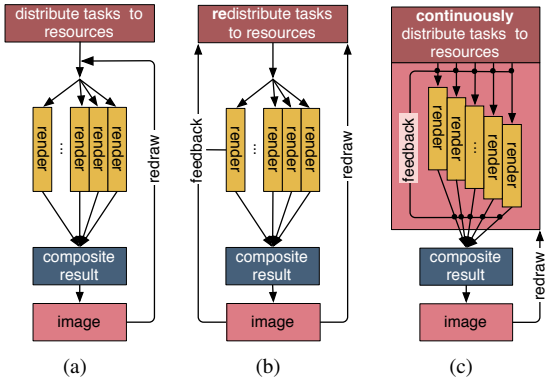


Figure 3: (a) *Static versus* (b) *an explicit dynamic load balancing that can adjust task decomposition between frames, and* (c) *fully adaptive implicit workload distribution.*

Therefore, in this work we explore a flexible implicit load balancing approach (as in 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.

3.1. Parallel Rendering Framework

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

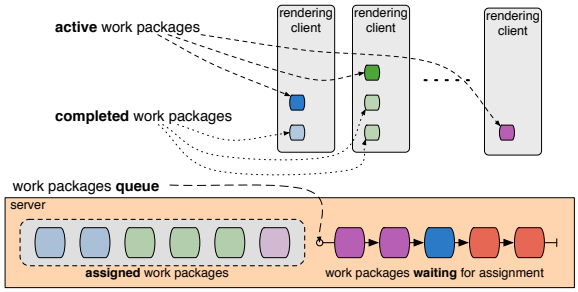


Figure 4: *Dynamic load balancing in distributed parallel rendering using work packages.*

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.

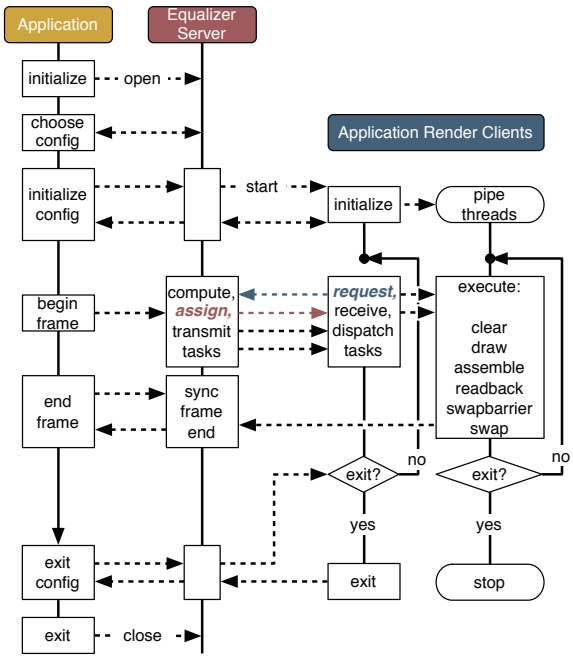


Figure 5: *Simplified execution flow of an Equalizer application using our work packages method. Note that clients request work packages from the server, which in turn assigns the packages to the respective client nodes, establishing a work-assignment loop that ends when all packages have been processed,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

3.2. Parallel Rendering Work Packages

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.

As illustrated in Figure 4, 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 dynamic affinity model, the server will respond with at maximum n_{req} work packages most suitable 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 below.

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.

3.3. Work Package Data Locality

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 $\mathscr{M}$ of work packages. 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 $k_{0\pm1}$ 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 improves 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 indicated in Figure 6(b). The rendering of nearby tiles $k_{0\pm1}$ from the starting tile k_0 of a rendering node, will require further 3D data that is spatially close (in perspective projection) 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.

3.4. Work Packages Affinity

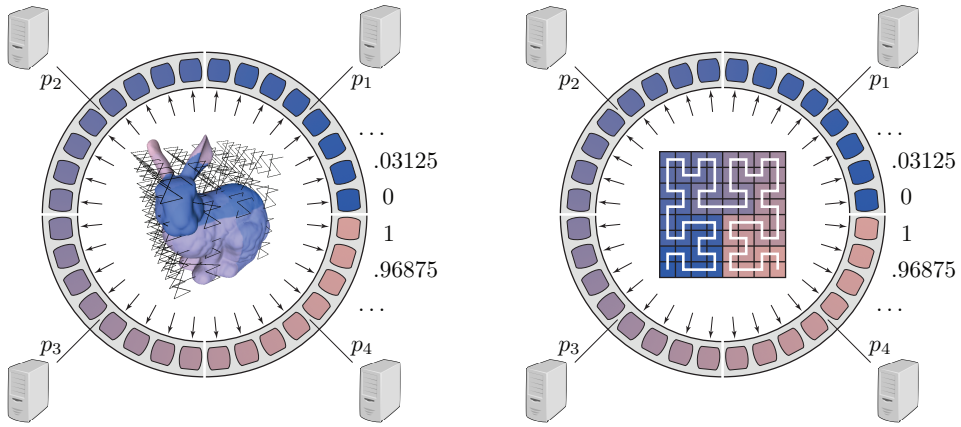
For work package to rendering node assignments, each rendering node is also associated with the linear space, and given a position $p \in [0,1]$ in this space. The work package $m(p)$ closest to this position is retrieved from the available ones in $\mathscr{M}$ according to Eq. 2. Here we use a circular addressing scheme, that utilizes a distance function d as defined in Eq. 1, which is exploited in a dynamic 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 dynamic 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 combined 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. The basic data locality of the work packages is achieved as explained in the previous section.

Our dynamic 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 in 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 work package consumption between neighboring client nodes, as illustrated in Figure 7(b).



(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* $p_4 = .875$*.*

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:

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

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. Consequently, the number of packages used to calculate a client's position is

$$n(p) = 1 + s(p,w) \tag{4}$$

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 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 from Algorithm 1 and Algorithm 2, respectively. Note that their respective roles within the *Equalizer* platform are illustrated in Figure 5.

More specifically, package request handling on the server is summarized in Algorithm 3. Note that node positions are not reset every frame, but are a function of work package consumption of the respective node, taking the previous w time steps into account (see Eq. 4), also across frame boundaries. Only before serving the very first request, this automatically results in an equidistant positioning of nodes.

Algorithm 1 Role of the server (simplified)

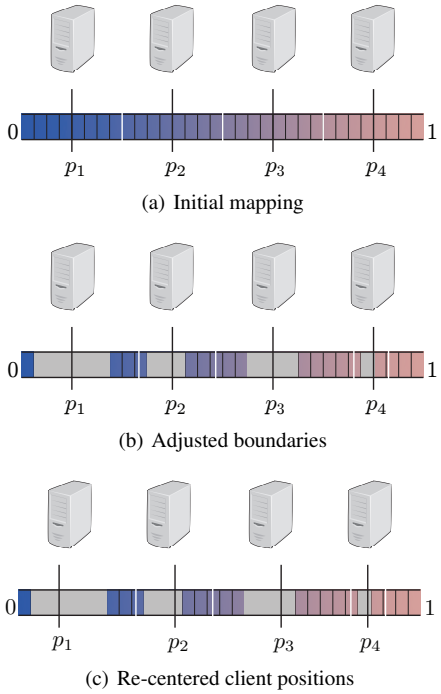
```
1:  while running do
2:      Start frame
3:      Generate package indices and spatial positions
4:      let n_total be the number of all available packages
5:      for each package x do
6:          k ← x.index / n_total
7:          Insert x into 𝒜 at k
8:      end for
9:      Handle package requests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

Algorithm 2 Role of the client (simplified)

```
1:  while rendering frame do
2:      let n_local be the number of locally available packages
3:      if n_local < 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 on the server

```
1:  procedure HandlePackageRequest(node, n_req)
2:      Calculate boundaries between all node positions
3:      Use boundaries to recalculate node positions
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 n_total > 0 do
9:          if d(p_new, package.position) ≤ d' then
10:             Add package to packages
11:             Remove package from 𝒜
12:             package ← m(p_new)
13:             Update n_total
14:             if packages.count ≥ n_req then
15:                 return packages
16:             end if
17:         end if
18:     end while
19:     return packages
20: end procedure
```



(a) Initial mapping

(b) Adjusted boundaries

(c) Re-centered client positions

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

4. Performance Analysis

We tested our system on a rendering cluster with 10 nodes, each equipped with a hexacore 1.8 GHz *Intel Xeon E5 processor*, 16 GB DDR4 RAM, and a *nVidia GTX* 970 GPU with 4 GB VRAM. The nodes are connected via a 40 Gbit/s *Infiniband* network.

For conducting our experiments, we used two different data sets: *David1mm*, with 56.2 M triangles, and *StMatthew*, with 372.8 M triangles. To avoid a trivial fragment processing scenario, both data sets were rendered using a procedural marble shader and simple spherical harmonic lighting (see Figure 8). In all experiments, we rendered our data sets at a final resolution of 1920x1080.

In order to simulate a challenging rendering scenario, we used a complex camera path where the camera is placed to the model very closely and moves along the major axis of the model while, simultaneously, the model rotates quickly around that axis (see also example shots in Figure 8). In this scenario the visibility of different parts of the model varies rapidly from frame to frame and thus the rendering load

Figure 8: *Screenshots of the David1mm model along the camera path. The model seems to enter the screen (right) and revolves along its longest axis while the camera moves along this axis as well (right to left image).*

is not easily predicted by traditional load balancing mechanisms.

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 64 segments of 3D data in object space. In Table 1 we compare our implicit dynamic load balancing method with conventional sort-first and sort-last dynamic load balancing approaches reliant on frame-to-frame coherence. The Equalizer platform already contains implementations of both approaches, respectively named as *2D Load Equalizer* and *DB Load Equalizer* [Eye08].

We additionally implemented two simple affinity models for comparison to the dynamic data locality and workload 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 *first-come, first-served* (FCFS) affinity model, conversely, simply maintains a list of work packages and assigns any requesting client with the first package available.

Our experiments are summarized in Table 1 which includes the draw and assembly time accumulated over all parallel nodes, and in Figure 9 which shows the dynamic development of draw and assembly times over time. In the latter the time reported is the passed wall-clock time for each frame, i.e. considering the maximum draw time needed by any node working in parallel, plus subsequent assembly time. Draw time is the duration that rendering of a frame requires on a node. Assembly time is the duration required to assemble the final image, including the time to wait for all nodes to finish rendering. Increased load and load imbalances can therefore increase assembly time. The increasing assembly time as shown in Figure 9 is likely a consequence of the used camera path, which results in the model entering the screen from one side until covering it completely (see Figure 8), hence steadily increasing the assembly cost. Reaching 600 frames the model increasingly covers the screen. Thus the rendering load is further increased, therefore, the frame statistics plotted in Figure 9 only partially contribute to the timings reported in Table 1 which cover a longer time period. Note that the Table 1 summarizes *total* draw times, i.e. the sum of draw times of all nodes of all frames. Conversely, Figure 9 shows the effective wall-clock draw time per frame; since nodes render in parallel, this is the maximum of all draw times per frame.

(a) Model *David1mm* (56.2 M triangles)

Method	**Draw**	**Assembly**	**Total**
Pack DB Equal	19940	53830	73770
Pack DB FCFS	22167	46089	68256
Pack DB Dynamic	20687	47094	**67781**
Load DB	43280	45848	89128
Pack 2D Equal	21966	10797	**32763**
Pack 2D FCFS	27464	9510	36974
Pack 2D Dynamic	23108	9985	33093
Load 2D	40034	6408	46442

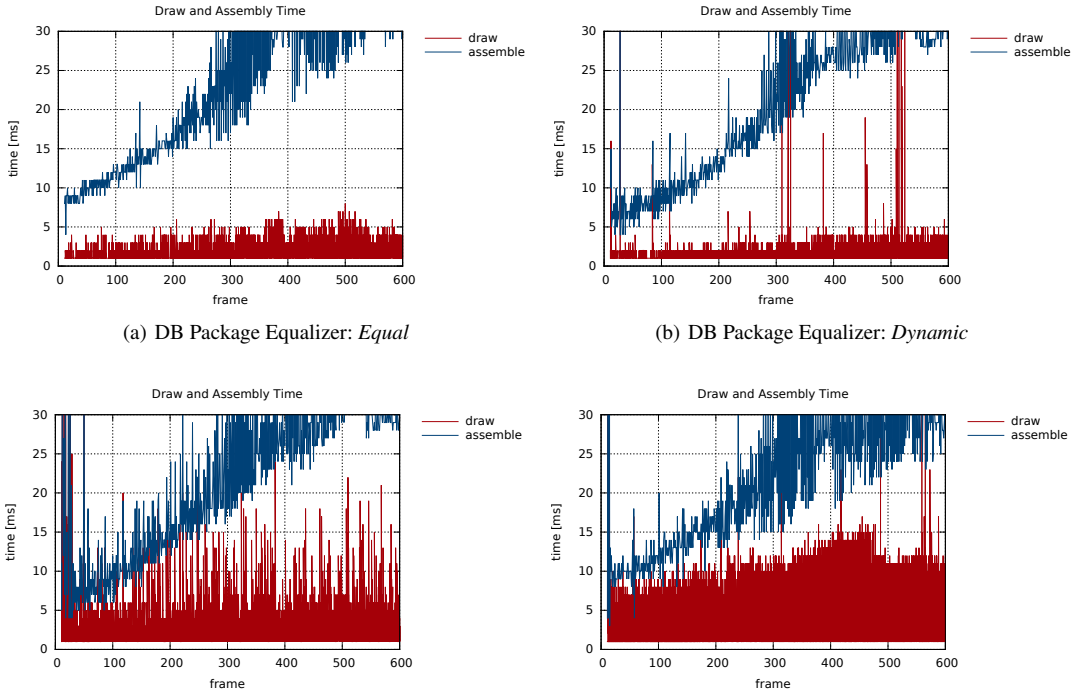
(b) Model *StMatthew* (372.8 M triangles)

Method	**Draw**	**Assembly**	**Total**
Pack DB Equal	28378	65692	94070
Pack DB FCFS	114386	64270	178656
Pack DB Dynamic	31597	58404	**90001**
Load DB	93204	57136	150340
Pack 2D Equal	47193	16913	**64106**
Pack 2D FCFS	126988	26340	153328
Pack 2D Dynamic	93488	26026	119514
Load 2D	96581	9188	105769

Table 1: *Total draw and assembly time in milliseconds for the* StMatthew *and the* David1mm *model, as well as the sum of these timings for our Package Equalizer (*Pack*) and the traditional Load Equalizer (*Load*) in sort-first (*2D*) and sort-last (*DB*) configurations with three different affinity models:* Equal, FCFS, *and* Dynamic. *The values were calculated over the duration of 1990 frames.*

Table 1 indicates that in the given sort-last parallel rendering scenarios (DB), our method exhibits better overall performance than the traditional *Load Equalizer* method, considering both draw and assembly times, as also notable in Figure 9. In the sort-last scenarios, the proposed dynamic affinity model also exhibits the expected improved performance compared to the alternative *Equal* and *FCFS* models.

In the sort-first (2D) scenarios the performance of our dynamic work package method and the affinity model is also better than the performance of Load Equalizer for the smaller *David1mm* model. However, of the tested affinity models, the simple *Equal* model works best in this scenario.



(a) DB Package Equalizer: *Equal*

(b) DB Package Equalizer: *Dynamic*

(c) DB Package Equalizer: *FCFS*

(d) DB Load Equalizer

Figure 9: *Draw and assembly times for rendering 600 frames of the rotating* StMatthew *model with both Package Equalizer and Load Equalizer in sort-last configuration (DB). The graphs result from the measurements summarized in Table 1.*

This can partially be explained by the *Equal* model being implemented using significantly less amount of overhead than the *Dynamic* model, while, unlike the *FCFS* model, still not ignoring data locality.

For the larger *StMatthew* model we can observe a similar behavior for sort-last (DB) rendering in Table 1, with the work packages method improving on the Load Equalizer approach. However, for sort-first (2D) rendering, only the *Equal* work package affinity model is faster than *Load Equalizer*. Among the factors that affect the performance of our method are the costs and effects associated with tile-based rendering of large-scale geometry that relate to data traversal and culling.

5. Discussion and Conclusion

We presented an implicit dynamic load balancing method for parallel rendering using a flexible rendering task partitioning approach and a novel work package pulling mechanism. In particular, we also introduced a dynamic affinity model for scoring the mapping of rendering tasks and computing resources to the same linear indexing space.

The results of our tests using the dynamic work packages method for rendering the given models using a challenging camera path in sort-last (DB) configurations, revealed a performance advantage of our method over a traditional load balancing method, based on the rendering times of previous frames.

In the tested sort-first configurations, the method for partitioning the rendering tasks in small work packages also exhibited overall better performance than a traditional load balancer. However, in this scenario the dynamic affinity model was not superior. Overhead costs and other rendering effects, such as culling costs that grow with geometry complexity, likely contribute to this. However, this may be less the case for large-scale volume rendering where less culling overhead can be expected.

Finally, the higher performance of our method in the tested sort-last configurations, in comparison with traditional load balancing, based on previous frame rendering times, suggests that our dynamic load balancing method is highly adaptive and can react more immediately to rapid changes in the distribution of the rendering load. Our dynamic affinity model also outperforms alternative *first-come, first-served* (FCFS) and *Equal* models in these scenarios. For the sort-first setup, further investigation is needed to understand the effects of culling overhead, more accurate culling and possibly off-screen rendering.

Furthermore, extended studies on varying settings and inhomogeneous rendering node capacities are expected to further reveal the potential benefits of dynamic work-package based load balancing in parallel rendering.

6. Acknowledgements

This work was supported in part by the EU FP7 People Programme (Marie Curie Actions) under REA Grant Agreement n°290227 and a Hasler Stiftung grant (project number 12097). We thank Fatih Erol, who helped implementing our early prototypes. 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

[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. 3

[Amd67] AMDAHL G. M.: Validity of the single-processor approach to achieving large scale computing capabilities. In *Proceedings American Federation of Information Processing Societies Joint Computer Conference* (1967), vol. 30, pp. 483–485. 2

[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 of IEEE Virtual Reality* (2001), pp. 89–96. 2

[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. 2

[CKP12] CHO Y., KIM M., PARK K. S.: LOTUS: Composing a multi-user interactive tiled display virtual environment. *The Visual Computer 28*, 1 (2012), 99–109. 2

[CT08] CEDERMAN D., TSIGAS P.: On dynamic load balancing on graphics processors. In *Proceedings ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (2008), pp. 57–64. URL: http://dl.acm.org/citation.cfm?id=1413957.1413967. 3

[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*, 2 (March 2011), 320–332. 2

[EEP11] EROL F., EILEMANN S., PAJAROLA R.: Cross-segment load balancing in parallel rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization* (2011), pp. 41–50. 3

[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. 2

[Eye08] EYESCALE SOFTWARE GMBH: Load Equalizer. http://www.equalizergraphics.com/scalability/loadEqualizer.html, 2008. 8

[HA98] HEIRICH A., ARVO J.: A competitive analysis of load balancing strategies for parallel ray tracing. *The Journal of Supercomputing 12*, 1-2 (1998), 57–68. doi:10.1023/A:1007977326603. 3

[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. 2

[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. 3

[JDB*04] JONES K., DANZER C., BYRNES J., JACOBSON K., BOUCHAUD P., COURVOISIER D., EILEMANN S., ROBERT P.: *SGI®OpenGL Multipipe™SDK User's Guide*. Tech. Rep. 007-4239-004, Silicon Graphics, 2004. URL: http://techpubs.sgi.com/library/manuals/4000/007-4239-004/sgi_html/index.html. 2

[KR04] KORCH M., RAUBER T.: A comparison of task pools for dynamic load balancing of irregular algorithms. *Concurrency and Computation: Practice and Experience 16*, 1 (2004), 1–47. URL: http://dx.doi.org/10.1002/cpe.745, doi:10.1002/cpe.745. 3

[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. 2

[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. 3

[MWP01] MORELAND K., WYLIE B. N., PAVLAKOS C. J.: Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2001), IEEE, pp. 85–92. 2

[NHM11] NEAL B., HUNKIN P., MCGREGOR A.: Distributed OpenGL rendering in network bandwidth constrained environments. In *Proceedings Eurographics Conference on Parallel Graphics and Visualization* (2011), Kuhlen T., Pajarola R., Zhou K., (Eds.), Eurographics Association, pp. 21–29. 2

[SFLS00] 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. 3

[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. 3