Equalizer 2.0 – Convergence of a Parallel Rendering Framework

Stefan Eilemann, David Steiner and Renato Pajarola

Abstract—Developing complex, real world graphics applications which utilize multiple GPUs and computers for interactive 3D rendering tasks is a complex task, it requires expertise in distributed systems and parallel rendering in addition to the application domain itself. We present a mature parallel rendering framework which provides a large set of features, algorithms and system integration for a wide range of real-world research and industry applications. Based on the basic architecture and implementation of the Equalizer parallel rendering framework presented in [**?**], we show how a variety of generic algorithms are integrated in the framework to help application scalability and development in many different domains, highlight how concrete applications benefit from the diverse aspects and use cases of Equalizer, and show new experimental results for parallel rendering and data distribution.

Index Terms—Parallel Rendering, Scalable Visualization, Cluster Graphics, Immersive Environments, Display Walls

1 Introduction

Highlight integration of many features into a general framework vs one-off research implementation of one algorithm. Also point out the various and very diverse examples in Section 7, highlighting the variety of application and implementation scenarios that can benefit from Equalizer.

The rest of this paper is structured as follows: First we introduce new related work since the introduction of Equalizer. The main body of this paper presents new performance features, virtual reality algorithms, usability features to build complex applications, main novel features of the underlying Collage network library and a quick overview of the main Equalizer-based applications. A result section presents new experiments not previously published, followed by the discussion and conclusion.

2 Related Work

In [**?**], we presented the initial Equalizer parallel rendering framework, which summarized the work in parallel rendering up to 2009. An extensive Programming and User Guide provides in-depth documentation on using and programming Equalizer [**?**]. In the following we assume these two publications and their references as a starting point, and focus on the related work published since then.

The concept of transparent OpenGL interception popularized by WireGL and Chromium [**?**] has received little attention since 2009. While some commercial implementations such as TechViz and MechDyne Conduit continue to exist, on the research side only ClusterGL [**?**] has been presented. ClusterGL employs the same approach as Chromium, but delivers a significantly faster implementation of transparent OpenGL interception and distribution for parallel rendering. CGLX [**?**] tries to bring parallel execution transparently to OpenGL applications, by emulating the GLUT API and intercepting certain OpenGL calls. In contrast to frameworks like Chromium and ClusterGL which distribute OpenGL calls, CGLX follows the distributed application approach. This works transparently for trivial applications, but quickly requires the application developer to address the complexities of a distributed application, when mutable application state needs to be synchronized across processes. For realistic applications, writing parallel applications remains the only viable approach for scalable parallel rendering, as shown by the success of Paraview, Visit and Equalizer-based applications.

On the other hand, software for driving and interacting with tiled display walls has received significant attention, including Sage [**?**] and Sage 2 [**?**] in particular. Sage was built entirely around the concept of a shared framebuffer where all content windows are separate applications using pixel streaming but is no longer actively supported. Sage 2 is a complete, browser-centric reimplementation where each application is a web application distributed across browser instances. DisplayCluster [**?**], and its continuation Tide [**?**], also implement the shared framebuffer concept of Sage, but provide a few native content applications integrated into the display servers. These solutions implement a scalable display environment and are a target display platform for scalable 3D graphics applications.

Equalizer itself has received significant attention within the research community. Various algorithms to improve the parallel rendering performance have been proposed: compression and region of interest during compositing [**?**]; load-balancing resources for multi-display installations [**?**]; asynchronous compositing and NUMA optimizations [**?**]; as well as work queueing [**?**]. Additionally, complex large scale and out-of-core multiresolution rendering approaches have been parallelized and implemented with Equalizer [**?**], [**?**], demonstrating the feasibility of the framework to be used with complex rendering algorithms and 3D model representations.

Furthermore, various applications and frameworks have used Equalizer for new research in visualization. On the application side, RTT Deltagen, Bino, Livre and RTNeuron [**?**] are the most mature examples and are presented in Section 7. On the framework side, Omegalib [**?**], a framework used in the Cave2, made significant progress in integrating 2D collaborative workspaces like Sage 2 with 3D immersive content.

- *All authors are with the Visualization and MultiMedia Lab, Department of Informatics, University of Zürich.*
- *S. Eilemann is also with the Blue Brain Project, Ecole Polytechnique Federale de Lausanne.*

(a) (b) (c)

Fig. 1: Example Equalizer applications: (a) 192 Megapixel CAVE at KAUST running RTNeuron, (b) Immersive HMD with external tracked and untracked views running RTT DeltaGen for virtual car usability studies, and (c) Cave2 running a molecular visualization.

visualization for remote sensing data [**?**]
[**?**] tiled display wall virtual environment

3 Performance

This section introduces new decomposition modes, runtime adaptations and other optimizations to increase the rendering performance for a wide variety of algorithms.

3.1 New Decomposition Modes

The initial version of Equalizer implemented the basic sort-first (2D), sort-last (DB), stereo (EYE) and multilevel decompositions [**?**]. In the following we present the newly added decomposition modes and motivate their use case. Figure 2 provides an overview of the new modes. We describe how to setup compound descriptions in the framework in [**?**].

3.1.1 Time-Multiplex

Time-multiplexing (Figure 2a), also called AFR or DPlex, was first implemented in [**?**] for shared memory machines. It is however a better fit for distributed memory systems, since the separate memory space makes concurrent rendering of different frames easier to implement. While it increases the framerate linearly, it does not decrease the latency between user input and the corresponding output. Consequently, this decomposition mode is mostly useful for non-interactive movie generation. It is transparent to Equalizer applications, but does require the configuration latency to be equal or greater than the number of source channels. Furthermore, to work in a multi-threaded, multi-GPU configuration, the application needs to support running the rendering threads asynchronously, as outlined below in Section 3.3.4. The output frame rate of the destination channel is smoothened using a *frame rate equalizer* (Section 3.2.4).

3.1.2 Tiles and Chunks

Tile (Figure 2b) and chunk decompositions are a variant of sort-first and sort-last rendering, respectively. They decompose the scene into a predefined set of fixed-size image tiles or database ranges. These tasks are queued and processed by all source channels by polling a server-central queue. Prefetching ensures that the task communication overlaps with rendering. As shown in [**?**], these modes provide better performance due to being inherently load-balanced, as long as there is an insignificant overhead for the render task setup. This mode is transparent to Equalizer applications.

3.1.3 Pixel

Pixel compounds (Figure 2c) decompose the destination channel by interleaving rows or columns in image space. They are a variant of sort-first decomposition which works well for fill-limited applications which are not geometry bound. Source channels cannot reduce geometry load through view frustum culling, since each source channel has almost the same frustum (only shifted by some pixels), but applied to a reduced 2D viewport. However, the fragment load on all source channels is very similar due to the interleaved distribution of pixels. This functionality is transparent to Equalizer applications, and the default compositing implementation uses the OpenGL stencil buffer to blit pixels onto the destination channel.

3.1.4 Subpixel

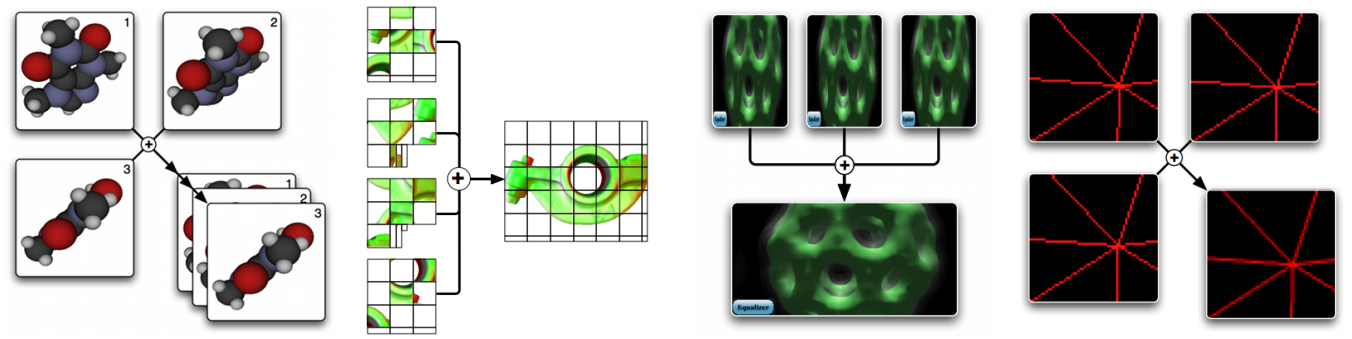
Subpixel compounds (Figure 2d) are similar to pixel compounds, but they decompose the work for a single pixel, for example when using multisampling or depth of field. Composition typically uses accumulation and averaging of all computed fragments for a pixel. This feature is not fully transparent to the application, since it needs to adapt (jitter or tilt) the frustum based on the iteration executed. Furthermore, subpixel compounds interact with idle image refinements, e.g., they can accelerate idle anti-aliasing of a scene when the camera and scene are not changed.

3.2 Equalizers

Equalizers are an addition to compound trees. They modify parameters of their respective subtree at runtime to dynamically optimize the resource usage, by each tuning one aspect of the decomposition. Due to their nature, they are transparent to application developers, but might have application-accessible parameters to tune their behavior. Resource equalization is the critical component for scalable parallel rendering, and therefore the eponym for the Equalizer project name. **FIXME: modified paragraph is ok; we have to ensure that the font differs between the Equalizer project and the Equalizer components consistently throughout the text**

3.2.1 Sort-First and Sort-Last Load Equalizer

Sort-first (Figure 3a) and sort-last load balancing is the most obvious optimization for these parallel rendering modes. Our load equalizer is fully transparent for application developers; that is, it uses a reactive approach based on past rendering times. This assumes a reasonable frame-to-frame coherency. Our implementation stores a 2D or 1D grid of the load, mapping the load of each



```
compound {
  channel "destination"
  framerate_equalizer {}
  compound {
    channel "source1"
    phase 0 period 3
    outputframe  { name "frame" }
  }
  compound {
    channel "source2"
    phase 1 period 3
    outputframe  { name "frame" }
  }
  compound {
    channel "source3"
    phase 2 period 3
    outputframe  { name "frame" }
  }
  inputframe { name "frame" }
}
```

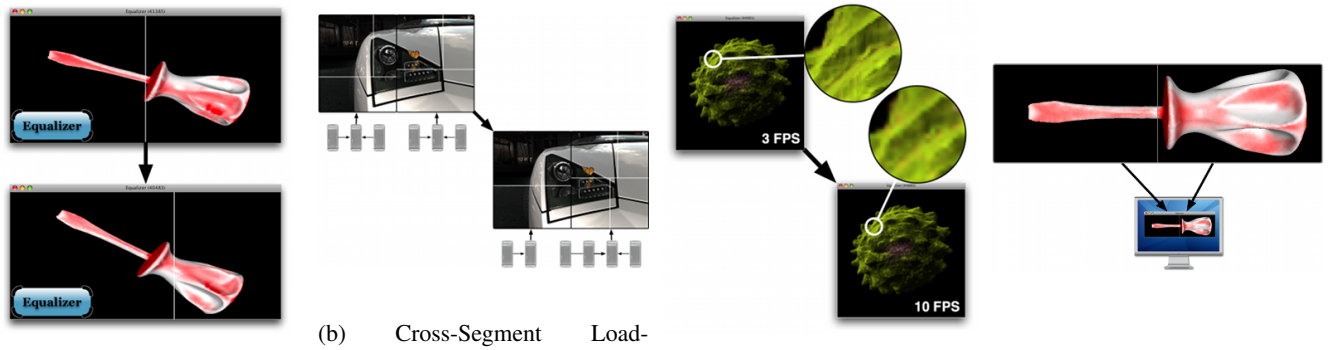
```
compound {
  channel "destination"
  outputtiles {
    name "queue"
    size [ 64 64 ]
  }
  compound {
    channel "destination"
    inputtiles { name "queue" }
  }
  compound {
    channel "source1"
    inputtiles { name "queue" }
    outputframe {}
  }
  compound {
    channel "source2"
    inputtiles { name "queue" }
    outputframe {}
  }
  compound {
    channel "source3"
    inputtiles { name "queue" }
    outputframe {}
  }
  inputframe { name "frame.source1" }
  inputframe { name "frame.source2" }
  inputframe { name "frame.source3" }
}
```

```
compound {
  channel "dest"
  compound {
    channel "dest"
    pixel [ 0 0 3 1 ]
    outputframe { type texture }
  }
  compound {
    channel "source1"
    pixel [ 1 0 3 1 ]
    outputframe {}
  }
  compound {
    channel "source2"
    pixel [ 2 0 3 1 ]
    outputframe {}
  }
  inputframe { name "frame.dest" }
  inputframe { name "frame.source1" }
  inputframe { name "frame.source2" }
}
```

```
compound {
  channel "dest"
  compound {
    channel "dest"
    subpixel [ 0 3 ]
    outputframe { type texture }
  }
  compound {
    channel "source1"
    subpixel [ 1 3 ]
    outputframe {}
  }
  compound {
    channel "source2"
    subpixel [ 1 3 ]
    outputframe {}
  }
  inputframe { name "frame.dest" }
  inputframe { name "frame.source1" }
  inputframe { name "frame.source2" }
}
```

(a) Time-Multiplex (b) Tiles (c) Pixel (d) Subpixel

Fig. 2: New Equalizer task decomposition modes and their compound descriptions for parallel rendering



(a) Load-Balancing (b) Cross-Segment Load-Balancing (c) Dynamic Frame Resolution (d) Monitoring

Fig. 3: Runtime modifications

channel. The load is stored in normalized 2D/1D coordinates using $\frac{time}{area}$ as the load, the contributing source channels are organized in a binary tree, and then the algorithm balances the two branches of each level by equalizing the integral over the area on each side.

We have implemented various tunable parameters allowing application developers to optimize the load balancing based on the characteristics of their rendering algorithm:

Damping reduces frame-to-frame oscillations. It is a normalized scalar defining how much of the computed delta from the previous position is applied. The equal load distribution within the region of interest assumed by the load equalizer is in reality not equal, causing the load balancing to overshoot.

Resistance eliminates small deltas in the load balancing step. This might help the application to cache some computations since the frustum does not change each frame.

Boundaries define the modulo factor in pixels onto which a load split may fall. Some rendering algorithms produce artefacts related to the OpenGL raster position, e.g., screen door transparency, which can be eliminated by aligning the boundary to the pixel repetition. Furthermore, some rendering

algorithms are sensitive to cache alignments, which can again be exploited by chosing the corresponding boundary.

3.2.2 Cross-Segment Load Balancing

Cross-segment load balancing (Figure 3b) addresses the optimal resource allocation of n rendering resources to m output channels (with $n \geq m$). The view equalizer works in conjunction with load equalizer balancing the individual output channels. It monitors the usage of shared source channels (across outputs) and activates them to balance the rendering time of all outputs. In [**?**], we provide a detailed description and evaluation of our algorithm.

3.2.3 Dynamic Frame Resolution

The DFR equalizer (Figure 3c) provides a functionality similar to dynamic video resizing [**?**]; that is, it maintains a constant framerate by adapting the rendering resolution of a fill-limited application. In Equalizer, this works by rendering into a source channel (typically on a FBO) separate to the destination channel, and then scaling the rendering during the transfer (typically through an on-GPU texture) to the destination channel. The DFR equalizer monitors the rendering performance and accordingly adapts the resolution of the source channel and zoom factor for the source to destination transfer. If the performance and source channel resolutions allows, this will not only subsample, but also supersample the destination channel to reduce aliasing artefacts.

3.2.4 Frame Rate Equalizer

The framerate equalizer smoothens the output frame rate of a destination channel by instructing the corresponding window to delay its buffer swap to a minimum time between swaps. This is regularly used for time-multiplexed decompostions, where source channels tend to drift and finish their rendering not evenly distributed over time. This equalizer is however fully independent of DPlex compounds, and may be used to smoothen irregular application rendering algorithms.

3.2.5 Monitoring

The monitor equalizer (Figure 3d) allows to reuse the rendering on another channel, typically for monitoring a larger setup on a control workstation. Output frames on the display channels are connected to input frames on a single monitoring channel. The monitor equalizer changes the scaling factor and offset between the output and input, so that the monitor channel has the same, but typically downscaled view, as the originating segments.

3.3 Optimizations

3.3.1 Region of Interest

The region of interest is the screen-space 2D bounding box enclosing the geometry rendered by a single resource. We have extended the core parallel rendering framework to use an application-provided ROI to optimize the load equalizer as well as image compositing performance. The load equalizer uses the ROI to refine its load grid to the regions containing data. The compositing code uses the ROI to minimize image readback and network transmission. In [**?**] and [**?**], we provide the details of the algorithm, and show that using ROI can quadruple the rendering performance, in particular for the costly compositing step in sort-last rendering.

3.3.2 Asynchronous Compositing

Asynchronous compositing pipelines rendering with compositing operations, by executing the image readback, network transfer and image assembly from threads running in parallel to the rendering threads. In [**?**], we provide the details of the implementation and experimental data showing an improvement of the rendering performance of over 25% for large node counts.

3.3.3 Download and Compression Plugins

Compression for the compositing step in parallel rendering is critical for performance. This not only applies to the well-researched network transfer step, but also for the transfer between GPU and CPU. Equalizer supports a variety of compression algorithms, from very fast RLE encoding, YUV subsampling on the GPU to JPEG compression. These algorithms are implemented as runtime-loaded plugins, allowing easy extension and customization to application-specific compression. In [**?**], we show this to be a critical step for interactive performance at scale.

3.3.4 Thread Synchronization Modes

Different applications have different degrees on how decoupled and thread-safe the rendering code is from the application logic. For full decoupling all mutable data has to have a copy in each render thread, which is not feasible in large data scenarios. To easily customize the synchronization of all threads on a single process, Equalizer implements three threading modes: Full synchronization, draw synchronization and asynchronous. Note that the execution between nodes is always asynchronous, for up to latency frames.

In full synchronization, all threads always execute the same frame; that is, the render threads are unlocked after Node::frameStart, and the node is blocked for all render threads to finish the frame before executing Node::frameFinish. This allows the render threads to read shared data from all their operations, but provides the slowest performance.

In draw synchronization, the node thread and all render threads are synchronized for all frameDraw operations; that is, Node::frameFinish is executed after the last channel is done drawing. This allows the render threads to read shared data during their draw operation, but not during compositing. Since compositing is typically independent of the rendered data, this is the default mode. This mode allows to overlap compositing with data synchronization on multi-GPU machines.

In asynchronous execution, all threads run asynchronously. Render threads may work on different frames at any given time. This mode is the fastest, and requires the application to have one instance of each mutable object in each render thread. It is required for using time-multiplex compounds in multithreaded rendering.

4 Virtual Reality

Virtual Reality is an important field for parallel rendering. It does however require special attention to support it as a first-class citizen in a generic parallel rendering framework. Equalizer has been used in many virtual reality installations, such as the Cave2 ([**?**]), the high-resolution C6 CAVE at the KAUST visualization laboratory and head-mounted displays (Figure 1). In the following we lay out the features needed to support these installations. All features presented were motivated by application use cases and have been validated with their respective users.

4.1 Head Tracking

Head tracking is the minimal feature needed to support immersive installations. Equalizer does support multiple, independent tracked views through the observer abstraction introduced in Section 5.1. Built-in VRPN support enables the direct, application-transparent configuration of a VRPN tracker device. Alternatively, an application can provide a 4×4 tracking matrix defining the transformation from the canvas coordinate system to the observer. Both CAVE-like tracking with fixed projection surfaces and HMD tracking modes are implemented.

4.2 Dynamic Focus Distance

To our knowledge, all parallel rendering systems have the focal plane coincide with the physical display surface. For better viewing comfort, we introduce a new dynamic focus mode, where the application defines the distance of the focal plane from the observer, based on the current view direction of the user. Initial experiments show that this is particularly effective for objects placed within the immersive space; that is, in front of all display segments.

4.3 Asymmetric Eye Position

Traditional head tracking computes the left and right eye positions by using a configurable interocular distance and the tracking matrix. However, human heads are not symmetric, and by measuring individual users a more precise frustum can be computed. Equalizer supports this through the optional configuration (file or programmatically) of individual 3D positions for the left and right eye.

4.4 Model Unit

This feature allows applications to specify a scaling factor between the model and the real world, to allow exploration of macroscopic or microscopic worlds in virtual reality. The unit is per view, allowing different scale factors within the same configuration. It scales both the specified projection surface as well as the eye position (and therefore separation) to achieve the necessary effect.

4.5 Runtime Stereo Switch

Applications can switch each view between mono and stereo rendering at runtime, and can run both monoscopic and stereoscopic views concurrently (Figure 1 (b)). While this is trivial for active stereo rendering, it does potentially involve the start and stop of resources and processes for passive stereo or stereo-dependent task decompositions, as described in Section 5.2.

5 USABILITY

In this section we present features motivated by real-world application use cases, i.e., new functionalities rather then performance improvements. We motivate the use case, explain the architecture and integration into our parallel rendering framework, and, where applicable, show the steps needed to use this functionality in applications.

5.1 Physical and Logical Visualization Setup

Real-world visualization setups are often complex, and having an abstract representation of the display system can simplify the configuration process. Real-world applications often have the need to be aware of spatial relationship of the display setup, for example to render 2D overlays or to configure multiple views on a tiled display wall.

We addressed this need through a new configuration section interspersed between the node/pipe/window/channel hardware resources and the compound trees configurating the resource usage for parallel rendering.

A typical installation consists of one projection canvas, which is one aggregated projection surface, e.g., a tiled display wall or a CAVE. Desktop windows are considered a canvas. Each canvas is made of one or more segments, which are the individual outputs connected to a display or projector. Segments can be planar or non-planar to each other, and can overlap or have gaps between each other. A segment is referencing a channel, which defines the output area of this segment, e.g., on a DVI connector connected to a projector.

A canvas can define a frustum, which will create default, planar sub-frusta for all of its segments. A segment can also define a frustum, which overrides the canvas frustum, e.g., for non-planar setups such as CAVEs or curved screens. These frusta describe a physically-correct display setup for a Virtual Reality installation. A canvas may have a software or hardware swap barrier, which will synchronize the rendering of all contributing GPUs. The software barrier executes a glFinish to ensure the GPU is ready to swap, a Collage barrier (Section 6.4) to synchronize all segments, the swap buffers call followed by a glFlush to ensure timely execution of the swap command. The hardware swap barrier is implemented using the WGL_NV_swap_group or GLX_NV_swap_group extensions.

On each canvas, the application can display one or more views. A view is a view on a model, in the sense used by the MVC pattern. The view class is used by Equalizer applications to define view-specific data for rendering, e.g., a scene, viewing mode or camera. The application process manages this data, and the render clients receive it for rendering.

A layout groups one or more views which logically belong together. A layout is applied to a canvas. The layout assignment can be changed at run-time by the application. The intersection between views and segments defines which output channels are available, and which frustum they should use for rendering. These output channels are then used as destination channels in a compound. They are automatically created during configuration.

A view may have a frustum description. The view's frustum overrides frusta specified at the canvas or segment level. This is used for non-physically correct rendering, e.g., to compare two models side-by-side on a tiled display wall. If the view does not specify a frustum, the corresponding destination channels will use the physically correct sub-frustum resulting from the view/segment intersection.

An observer looks at one or more views. It is described by the observer position in the world and its eye separation. Each observer will have its own stereo mode, focus distance and frame loop (framerate). This allows to have untracked views and multiple tracked views, e.g., two HMDs, in the same application.

5.2 Runtime Reconfiguration

Switching a layout, as described above, or switching the stereo rendering mode, may involve a different set of resources after the change, including the launch and exit of render client processes. Equalizer solves this through a reconfiguration step at the beginning of each rendering frame. Each resource (channel, window, pipe, node) has an activation count, which is updated when the layout or any other relevant rendering parameter is changed. When a resource is found whose activation count does not match its current start/stopped state, the resource is created or destroyed and configInit or configExit are called accordingly. In the current implementation, a normal configuration initialization or exit, as described in [**?**], uses the same code path with all used resources transitioning to a running or stopped state, accordingly. Since starting new resources typically requires object mapping and associated data distribution, it is a costly operation. To improve performance and robustness, Equalizer will flush the rendering pipeline whenever a runtime reconfiguration is detected, before executing the change. This implementation allows the late join or early exit, transparent to application developers.

5.3 Automatic Configuration

Automatic configuration implements the discovery of local and remote resources as well as the creation of typical configurations using the discovered resources at application launch time.

The discovery is implemented in a separate library, hwsd (HardWare Service Discovery), which uses a plugin-based approach to discover GPUs for GLX, AGL or WGL windowing systems, as well as network interfaces on Linux, Mac OS X and Windows. Furthermore, it detects the presence of VirtualGL to allow optimal configuration of remote visualization clusters. The resources can be discovered on the local workstation, and through the help of a simple daemon using the zeroconf protocol, on a set of remote nodes within a visualization cluster. A session identifier may be used to support multiple users on a single cluster.

The Equalizer server uses the hwsd library to discover local and remote resources when an hwsd session name instead of a `.eqc` configuration file is provided. A set of standard decomposition modes is configured, which can be selected through activating the corresponding layout.

This versatile mechanism allows non-experts to configure and profit from multi-GPU workstations and visualization clusters, as well as to provide system administrators with the tools to implement easy to use integration with cluster schedulers. This feature is transparent to Equalizer application developers.

5.4 Qt Windowing

Qt is a popular window system with many application developers. Unfortunately, it imposes a different threading model for window creation and event handling compared to Equalizer. In Equalizer, each GPU rendering thread is independently responsible for creating its windows, receiving the events and eventually dispatching them to the application process main thread. This design is motivated by the natural threading model of X11 and WGL, and allows simple sequential semantics between OpenGL rendering and event handling. In contrast, Qt requires all windows and QOpenGLContext to be created from the Qt main thread. An existing Qt window or context may be moved to a different thread, and events are signalled from the main thread.

Challenges in threading model, architecture

5.5 Tide Integration

Tide (Tiled interactive display environment) is an improved version of DisplayCluster [**?**], providing a touch-based, multi-window user interface for high-resolution tiled display walls. Remote applications receive input events and send pixel streams using the Deflect client library. Equalizer includes full support, enabling application-transparent integration with Tide. When a Tide server is configured, all output channels of a view stream in parallel to one window on the wall. In [**?**], we have shown interactive framerates for a 24 megapixel resolution over a WAN link. Events received are translated into Equalizer events and injected into the normal event flow, allowing seamless application integration.

5.6 Sequel

Sequel is a simplification layer for Equalizer. It is based on the realization that while fully expressive, the verbatim abstraction layer of nodes, pipes, windows and channels in Equalizer requires significant learning to fully understand and exploit. In reality, a higher abstraction of Application and Renderer is sufficient for most use cases. In Sequel, the application class drives the configuration, and one renderer instance is created for each (pipe) render thread. They also provide the natural place to store and distribute data. Last, but not least, ViewData provides a convenient way to manage multiple views by storing the camera, model or any other view-specific information.

6 The Collage Network Library

An important part of writing a parallel rendering application is the communication layer between the individual processes. Equalizer relies on the Collage network library for its internal operation. Collage furthermore provides powerful abstractions for writing Equalizer applications, which are introduced in this section.

6.1 Architecture

Collage provides networking functionality of different abstraction layers, gradually providing higher level functionality for the programmer. The main primitives in Collage are:

Connection A stream-oriented point-to-point communication line. The connections transmit raw data reliably between two endpoints for unicast connections, and between a set of endpoints for multicast connections. For unicast, process-local pipes, TCP and Infiniband RDMA are implemented. For multicast, a reliable, UDP-based protocol is discussed in Section 6.2.

DataI/OStream Abstracts the input and output of C++ data types from or to a set of connections by implementing output stream operators. Uses buffering to aggregate data for network transmission. Performs endian swapping during input if the endianness differs between the originating and local node.

Node and LocalNode The abstraction of a process in the cluster. Nodes communicate with each other using connections. A LocalNode listens on various connections and processes requests for a given process. Received data is wrapped in ICommands and dispatched to command handler methods. A Node is a proxy for a remote LocalNode. The Equalizer Client object is a LocalNode.

Object Provides object-oriented, versioned data distribution of C++ objects between nodes. Objects are registered or mapped on a LocalNode.

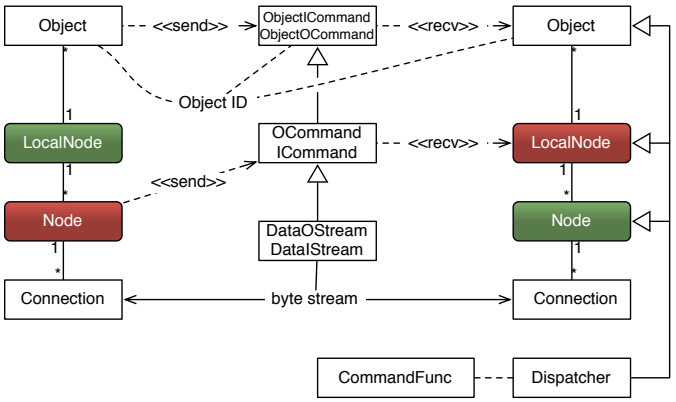


Fig. 4: Communication between two Collage objects

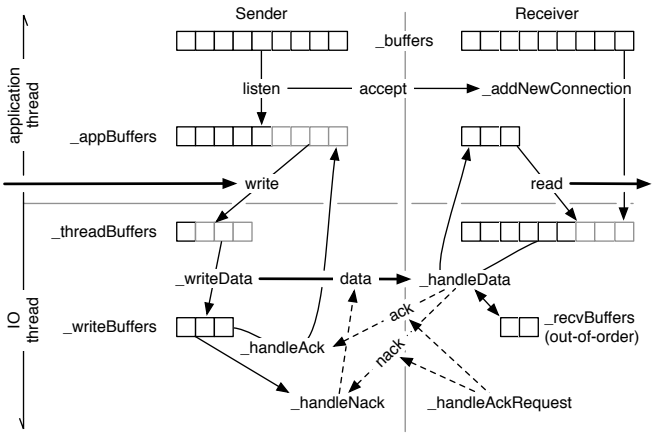


Fig. 5: Reliable Stream Protocol

Figure 4 illustrates this architecture.

6.2 Reliable Stream Protocol

RSP is an implementation of a reliable multicast protocol over unreliable UDP transport. RSP behaves similarly to TCP, it provides full reliability and ordering of the data, and slow receivers will eventually throttle the sender. This behavior is needed to guarantee delivery of data in all situations. Pragmatic generic multicast (PGM [**?**]) provides full ordering, but slow clients will disconnect from the multicast session instead of throttling the send rate.

RSP combines various established algorithms [**?**], [**?**] for multicast in an open source implementation capable of delivering wire speed transmission rates on high-speed LAN interfaces. In the following we will outline the RSP protocol and implementation as well as motivate the design decisions. Any defaults given below are for Linux or OS X, the Windows UDP stack requires different default values which can be found in the implementation.

Our RSP implementation uses a separate protocol thread for each RSP group, which handles all reads and writes on the multicast group. This thread implements the protocol handling and communicates with the application threads through thread-safe queues, as shown in Figure 5. These queues contain datagrams, which are prefixed by a type-specific header. Each connection has a configurable number of buffers, (1024 by default) of a configurable MTU (1470 bytes default) which are either free for use or in use for reading or writing.

A client opens a listening multicast RSP connection. Similar to TCP, this listener will create one connected RSP connection for each remote member in the multicast group; that is, on each node n RSP connection instances exist, 1 for the local listener and $n-1$ for all other nodes. A multicast group therefore looks like a fully connected set of TCP connections between n nodes for the user.

Each member of a multicast group has a group-unique 16 bit identifier, which is randomly selected, announced to the group, and if no collisions are notified by any other member, confirmed for this connection. This mechanism allows addressing of connections with a low protocol overhead. The datagrams used in this exchange contain no payload, and they contain two bytes type (HELLO, HELLO_REPLAY, HELLO_DENY, HELLO_CONFIRM), two bytes protocol version and two bytes connection identifier.

The _appBuffers queue, which is blocking, contains received UDP datagrams on incoming connections, or empty buffers for writing on outgoing connections. The _threadBuffers queue is lock-free and used by the protocol thread to store empty buffers on incoming connections, and pre-packaged UDP datagrams on outgoing connections. The application thread pushes buffers into this queue, consumed by the protocol thread. A datagram is up to 64 kilobytes (UDP limit, configurable) and contains a header (two bytes type, two bytes size, two bytes source identifier, two bytes sequence number) and its payload. The sequence number is used for positive (ack) and negative (nack) acknowledgments and reordering of packets.

The _writeBuffers std::deque is only used by the protocol thread. It contains already sent, but not yet acknowledged datagrams. Its datagrams are resent upon reception of a nack or recycled into _appBuffers when acknowledged by all receivers. Similarly, the _recvBuffers std::deque contains received out-of-order packets already received by an incoming connection. Upon each reception it is checked, and in-order datagrams are placed into _appBuffers for consumption by the application thread.

Handling a smooth packet flow is critical for performance. RSP uses active flow control to advance the byte stream buffered by the implementation. Each incoming connection actively acknowledges every n (17 by default) packets fully received. The incoming connections temporarily offset this acknowledgment by their connection identifier to avoid bursts of acks. Any missed datagram is actively nack'ed as soon as it is detected. Write connections continuously retransmit packets from nack datagrams, and advance their _writeBuffers upon reception of acks from all readers. Furthermore, the writer will explicitly request an ack or nack when it runs out of empty buffers or finishes its write queue. Nack datagrams may contain multiple ranges of missed datagrams, as UDP implementations often drop multiple packets at once.

Congestion control is necessary to optimize bandwidth usage. While TCP uses the well-known AIMD (additive increase, multiplicative decrease) algorithm, we have chosen a more aggressive congestion control algorithm of additive increase and decrease. This has proven experimentally to be more optimal: UDP is often rate-limited by switches; that is, packets are discarded regularly and not exceptionally. Only slowly backing of the current send rate helps to stay close to this limit. Furthermore, our RSP traffic is limited to the local subnet, making cooparation between multiple data stream less of an issue. Send rate limiting uses a bucket algorithm, where over time the bucket fills with send credits, from which sends are substracted. If there are no available credits, the sender sleeps until sufficient credits are available.

6.3 Distributed, Versioned Objects

Adapting an existing application for parallel rendering requires the synchronization of application data across the processes in the parallel rendering setup. Existing parallel rendering frameworks address this often poorly, at best they rely on MPI to distribute data. Real-world, interactive visualization applications are typically written in C++ and have complex data models and class hierarchies to represent their application state. As outlined in [**?**], the parallel rendering code in an Equalizer application only needs access to the data needed for rendering, as all application logic is centralized in the application main thread. We have encountered two main approaches to address this distribution: Using a shared filesystem for static data, or using data distribution for static and dynamic data.

Distributed objects are not required to build Equalizer applications. While most developers choose to use this abstraction for convenience, we have seen applications using other means for data distribution, e.g., MPI.

6.3.1 Programming Interface

Distributed objects in Collage provide powerful, object-oriented data distribution for C++ objects. They facilitate the implementation of data distribution in a cluster environment. Distributed objects are created by subclassing from co::Serializable or co::Object. The application programmer implements serialization and deserialization of the distributed data. Distributed objects can be static (immutable) or dynamic. Objects have a universally unique identifier (UUID) to address them cluster-wide. A master-slave model is used to establish mapping and data synchronization across processes. Typically, the application main loop registers a master instance and communicates the UUID to the render clients, which map their instance to the given identifier. The following object types are available:

Static The object is not versioned nor buffered. The instance data is serialized whenever a new slave instance is mapped. No additional data is stored.

Instance The object is versioned and buffered. The instance and delta data are identical; that is, only instance data is serialized. Previous instance data is saved to be able to map old versions.

Delta The object is versioned and buffered. The delta data is typically smaller than the instance data. The delta data is transmitted to slave instances for synchronization. Previous instance and delta data is saved to be able to map and sync old versions.

Unbuffered The object is versioned and unbuffered. No data is stored, and no previous versions can be mapped. No old versions can be mapped and no additional data is stored.

All distributed objects have to implement getInstanceData and applyInstanceData to serialize and deserialize the object's distributed data. These methods provide an output or input stream as a parameter, which abstracts the data transmission and can be used like a std::stream. The data streams implement efficient buffering and compression, and automatically select the best connection for data transport. Custom data type serializers can be implemented by providing the appropriate serialization functions. No pointers should be directly transmitted through the data streams. For pointers, the corresponding object is typically a distributed object as well, and its identifier and potentially version is transmitted in place of the memory address.

Dynamic objects are versioned, and have to override getChangeType to indicate how they want to have changes to be handled. Upon commit the delta data from the previous version is sent, if available using multicast, to all mapped slave instances. The data is queued on the remote node, and is applied when the application calls sync to synchronize the object to a new version. The sync method might block if a version has not yet been committed or is still in transmission. All versioned objects the following characteristics:

- The master instance of the object generates new versions for all slaves. These versions are continuous. It is possible to commit on slave instances, but special care has to be taken to handle possible conflicts.
- Slave instance versions can only be advanced; that is, sync(version) with a version smaller than the current version will fail.
- Newly mapped slave instances are mapped to the oldest available version by default, or to the version specified when calling mapObject.

Besides the instance data serialization methods used to map an object, delta and unbuffered objects may implement pack and unpack to serialize or deserialize the changes since the last version.

The Collage Serializable implements one convenient usage pattern for object data distribution. The co::Serializable data distribution is based on the concept of dirty bits, allowing inheritance with data distribution. Dirty bits form a 64-bit mask which marks the parts of the object to be distributed during the next commit. For serialization, the application developer implements serialize or deserialize, which are called with the bit mask specifying which data has to be transmitted or received. During a commit or sync, the current dirty bits are given, whereas during object mapping all dirty bits are passed to the serialization methods.

Blocking commits allow to limit the number of outstanding, queued versions on the slave nodes. A token-based protocol will block the commit on the master instance if to many unsynchronized versions exist.

6.3.2 Optimizations

The API presented in the previous section provides sufficient abstraction to implement various optimizations for faster mapping and synchronization of data: compression, chunking, caching, preloading and multicast. The results section evaluates these optimizations.

The most obvious one is compression. Recently, many new compression algorithms have been developed which exploit modern CPU architectures and deliver compression rates well above one gigabyte per second. Collage uses the Pression library [**?**], which provides an unified interface for a number of compression libraries, such as FastLZ [**?**], Snappy [**?**] and ZStandard [**?**]. It also contains a custom, virtually zero-cost RLE compressor. Pression parallelizes the compression and decompression using data decomposition. Note that this compression is generic, and implemented transparently for the application. Applications can still use data-specific compression and feed the compressed data to Collage, typically disabling the generic compression.

The data streaming interface also implements chunking, which aims to pipeline the serialization code with the network transmission. After a configurable number of bytes has been serialized to the internal buffer, it is transmitted and serialization continues. This is used both for the initial mapping data and for commit data.

Caching retains instance data of objects in a cache, and reuses this data to accelarate mapping of objects. The instance cache is either filled by "snooping" on multicast transmissions or by an explicit preloading when the master objects are registered. The preloading ("send on register") sends instance data of recently registered master objects while the correspinding node is idle to all connected nodes. These nodes simply enter the received data to their cache. Preloading uses multicast when available.

Due to the master-slave model of data distribution, multicast can be used to optimize the transmission time of data. If the contributing nodes share a multicast session, and more than one slave instance is mapped, Collage automatically uses the multicast connection to send the new version information.

6.4 Barriers, Queues and Object Maps

Collage implements a few generic distributed objects which are used by Equalizer and applicatons. A barrier is a distributed barrier primitive used for software swap barriers in Equalizer (Section 5.1). Its implementation follows a simple master-slave approach, which has shown to by sufficient for this use case. Queues are single producer, multiple consumer FIFOs. To hide network latencies, the consumers use prefetching of queue items into a local queue. They are used for tile and chunk compounds (Section 3.1.2).

The object map facilitates distribution and synchronization of a collection of distributed objects. Master versions can be registered on a central node, e.g., the application node in Equalizer. Consumers, e.g., Equalizer render clients, can selectively map the objects they are interested in. Committing the object map will commit all registered objects and sync their new version to the slaves. Syncing the map on the slaves will synchronize all mapped instances to the new version recorded in the object map. This effective design allows data distribution with minimal application logic. It is used by Sequel (Section 5.6) and other Collage applications.

7 Applications

In this section, we present some major applications built using Equalizer, and show how they interact with the framework to solve complex parallel rendering problems.

7.1 Livre

Livre (Large-scale Interactive Volume Rendering Engine) is a GPU ray-casting based parallel 4D volume renderer, implementing state-of-the-art view-dependent level-of-detail rendering (LOD) and out-of-core data management [**?**]. Hierarchical and out-of-core LOD data management is supported by an implicit volume octree, accessed asynchronously by the renderer from a data source on a shared file system. Different data sources providing an octree conform access to RAW or compressed as well as to implicitly generated volume data (e.g. such as from event simulations or surface meshes) can be used.

High-level state information, e.g., camera position and rendering settings, is shared in Livre through Collage objects between the parallel applications and rendering threads. Sort-first decomposition is efficiently supported through octree traversal and culling both for scalability as well as for driving large-scale tiled display walls.

7.2 RTT Deltagen

RTT Deltagen (now Dassault 3D Excite) is a commercial application for interactive, high quality rendering of CAD data. The RTT Scale module, delivering multi-GPU and distributed execution, is based on Equalizer and Collage, and has driven many of the aforementioned features.

RTT Scale uses a master-slave execution mode, were a single running Deltagen instance can go into "Scale mode" at any time by launching an Equalizer configuration. Consequently, the whole internal representation needed for rendering is based on a Collage-based data distribution. The rendering clients are separate, smaller applications which will map their scenes during startup. At runtime, any change performed in the main application is committed as a delta at the beginning of the next frame, following a design pattern similar to the Collage Serializable (Section 6.3.1). Multicast (Section 6.2) is used to keep data distribution times during session launch reasonable for larger cluster sizes (tens to hundreds of nodes).

RTT Scale is used for a wide variety of use cases. In virtual reality, the application is used for virtual prototyping and design reviews in front of high-resolution display walls or in a CAVE and for virtual prototyping of human-machine interactions in a CAVE or using HMDs. For scalability, sort-first and tile compounds are used to achieve fast, high-quality rendering, primarily for interactive raytracing, both based on CPUs and GPUs. For CPU-based raytracing, often Linux-based rendering clients are used with a Windows-based application node.

7.3 RTNeuron

[**?**]

7.4 Raster

RASTeR [**?**] is an out-of-core and view-dependent real-time multiresolution terrain rendering approach using a patch-based restricted quadtree triangulation. For load-balanced parallel rendering [**?**] it exploits fast hierarchical view-frustum culling of the level-of-detail (LOD) quadtree for sort-first decomposition, and uniform distribution of the visible LOD triangle patches for sort-last decomposition. The latter is enabled by a fast traversal of the patch-based restricted quadtree triangulation hierarchy which results in a list of selected LOD nodes, constituting a view-dependent cut or *front of activated nodes* through the LOD hierarchy. Assigning and distributing equally sized segments of this active LOD front to the concurrent rendering threads results in a near-optimal sort-last decomposition for each frame.

7.5 Bino

Bino is a stereoscopic 3D video player capable of running on very large display systems. Originally written for the immersive semi-cylindrical projection system at the University of Siegen, it has been used in many installations thanks to its flexibility of configuration. Bino decodes the video on each Equalizer rendering process, and only synchronizes the time step globally, therefore providing a scalable solution to video playback.

7.6 Omegalib

8 Experimental Results

This section presents new experiments, complementing the results of previous publications [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. The first

part summarizes rendering performance over all decomposition modes with a few representative workloads. The second part analyses data distribution performance, in particular how the various optimizations in Collage effect complex operations in realistic scenarios.

8.1 Decomposition Modes

Two graphs: Time to render a 4k, 256-step MSAA image of largest ply model and Livre using 1..16 nodes with all modes (2D, 2D LB, DB, DPlex (over AA steps), tiles, chunks, pixel, subpixel) (+DFR for Livre)

8.2 Object Distribution

The data distribution benchmarks have been performed on a cluster using dual processor Intel Xeon X5690 CPUs (3.47GHz), 24 GB main memory, 10GBit/s Ethernet and QDR Infiniband. Intel ICC 2017 has been used with CMake 3.2 release mode settings to compile the software stack.

To benchmark the data distribution we used two datasets: The david statue at 2mm resolution (Figure 6) and 3D volumes of a spike frequencies of an electrical simulation of three million neurons (Figure 7).

Fig. 6: David statue at 2mm resolution (145MB)

The ply file is converted into a kd-Tree for fast view frustum culling and rendering, and the resulting data structure is serialized in binary form for data transmission.

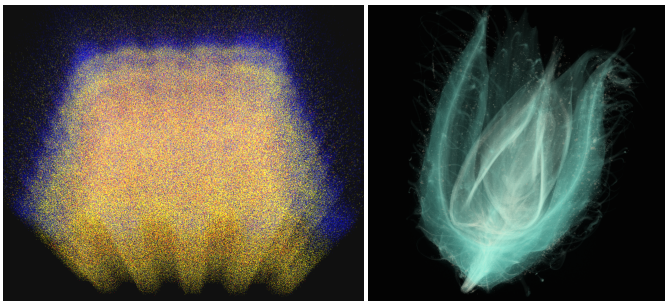


Fig. 7: Volumes used for object distribution benchmarks: spike frequencies of a three million neuron electrical simulation at 512x437x240 unsigned byte (51MB, left) and an MicroCT scan of a beechnut at 1024x1024x1546 unsigned short (3GB, right) resolution

The spike frequency volumes aggregate the number of spikes which happened within a given voxel over a given time. The absolute spike count is renormalized to an unsigned byte value during creation. Higher densities in the volume represent higher spiking activity in the voxel.

8.2.1 Data Compression Engines

A critical piece for data distribution performance are the characteristics of the data compression algorithm. Our microbenchmark compresses a set of binary files, precalculating the speed and compression ratio of the various engines. Figure 8 shows the compression and decompression speed in gigabyte per second as well as the size of the compressed data relative to the uncompressed data. The different ZSTD*x* engines use the ZStandard compression library at compression level *x*. The measurements were performed on a single, isolated node.

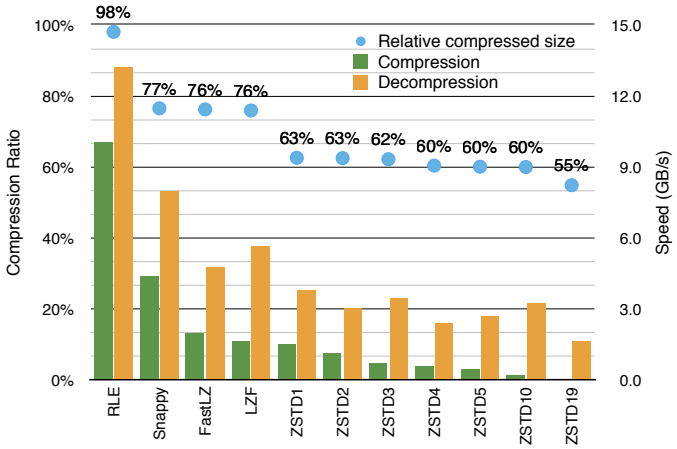


Fig. 8: Data compression for generic binary data

RLE compression has a very low overhead, but does not compress the data, since it only removes "blank space" in the data. The Snappy compression, used as default in Collage, achieves the same compression ratio as the LZ variants at a much higher speed. The ZStandard compressor has roughly the same speed as the LZ variants at the lowest compression level, but provides significantly better compression. At higher compression levels it can improve the compression ratio slightly, but at a high cost for the compression speed.

The compression ratio for the models used in the following section deviate from this averaged distribution. Figure 9 shows the compression ratios for the triangle and volume data.

The ply data is little compressible, the default compressor achieves a 10% reduction. This is due to a high entropy in the data and the dominant use of floating point values. Overall the profile is similar to the generic benchmark, at a smaller compression rate.

The volume data on the other hand is sparsely populated and using integer (byte and short) values, which is easier to compress. The naive RLE implementation already achieves a good compression, showing that the smaller volume contains at most 28% empty space and the bigger volume at most 43%. Snappy and ZStandard can reduce the spike data much further, reducing the data to a few megabytes. Surprisingly, the beechnut data set does not yield significant higher compression with the modern Snappy and ZStandard libraries.

8.2.2 Model Distribution and Update

In this section we present how data distribution and synchronization performs in real-world applications. We extracted the

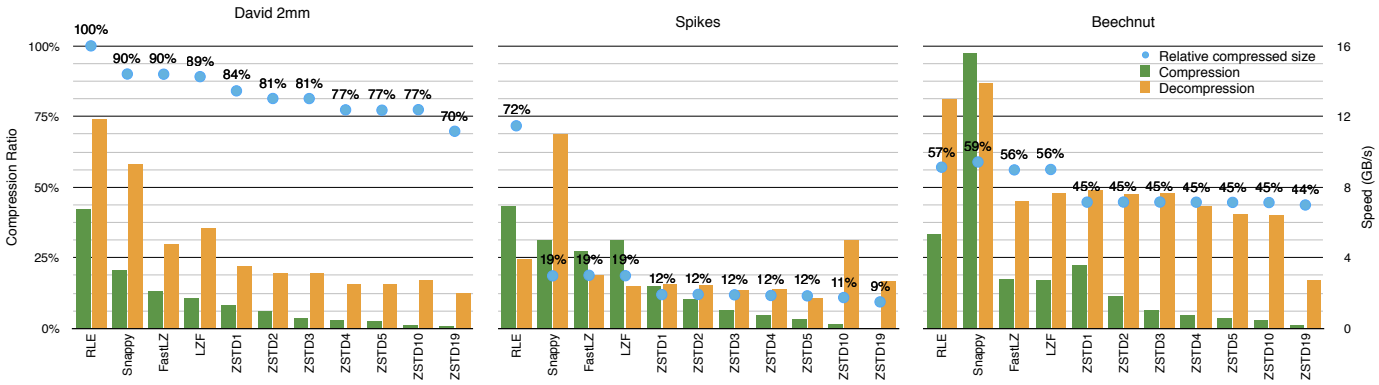


Fig. 9: Data compression for ply data (left, David statue 2mm) and raw volumes shown in Figure 7 (middle and right)

existing data distribution code in a PLY (eqPly) and a volume renderer (Livre) into a benchmark application to measure the time to initially map all the objects on the render client nodes, and to perform a commit-sync of the full data set after mapping has been established. All figures observe a noticable measurement jitter due to other services running on the cluster. The details of the benchmark algorithm can be found in the implementation [**?**]. We used the same data sets as in the previous section. After eight processes, a nodes start to run two processes per node, which share CPU, memory and network interface bandwidth.

Object mapping is measured using the following settings: **none** distributes the raw, uncompressed, and unbuffered data, **compression** uses the Snappy compressor to distribute unbuffered data, **buffered** reuses uncompressed, serialized data for mappings of other nodes, and **compression buffered** compresses this buffer. Unbuffered operations need to reserialize, and potentially recompress, the master object data for each slave node. Each slave instance needs to deserialize and decompress the data.

During data synchronization, the master instance commits the object data to all mapped slave instances simultaneously. This is a "push" operation, whereas the mapping is a slave-triggered "pull" operation with additional synchronization points. Slave nodes queue this data and consume it during synchronization. We test the time to commit and sync the data using different compression engines.

The David statue at 2mm resolution is organized in a kD-tree for rendering. Each node is a separate distributed object, which has two child node objects. A total of 1023 objects is distributed and synchronized. Figure 10 shows the distribution times for this data set. Due to limited compressibility of the data, the results are relative similar. Compressing the data repeatedly for each client leads to decreased performance, since the compression overhead cannot be amortized by the decreased transmission time. Buffering data slightly improves performance by reducing the CPU and copy overhead. Combining compression and buffering leads to the best performance, although only by about 10%.

During synchronization, data is pushed from the master process to all mapped slaves using a unicast connection to each slave. While the results in the Figure 10 (middle) are relatively close to each other, we can still observe how the tradeoff between compression ratio and speed influences overall performance. Better, slower compression algorithms lead to better overall better performance when amortized over many send operations.

The volume data sets are distributed in a single object, serializing the raw volume buffer. The spike volume data set has a significant compression ratio, which reflects in the results in Figure 11. Compression for this data is benefitial for transmitting data over a 10 Gb/s link even for a single slave process only (970 MB/s vs 570 MB/s raw data transfer). Buffering has little benefit since the serialization of volume data is trivial. Buffered compression makes a huge difference, since the compression cost can be amortized over n nodes, reaching raw data transmission rates of 3.7 GB/s with the default Snappy compressor and at best 4.4 GB/s with ZStandard at level 1.

The distribution of the beechnut data set also behaves as expected (Figure 12). Due to the larger object size, uncompressed transmission is slightly faster compared to the spike data set at 700 MB/s, and compressed transmission does not improve the mapping performance, likely due to increased memory pressure caused by the large data size. The comparison of the various compression engines is consistent with the benchmarks in Figure 9; RLE, Snappy and the LZ variants are very close to each other, and ZSTD1 can provide better performance after four nodes due to the better compression ratio.

Finally, we compare data distribution speed using different protocols. In this benchmark, data synchronization time of the spike volume data set is measured, as in Figure 11 (middle). Buffering is enabled, and compression is disabled to focus on the raw network performance. Figure 13 shows the performance using various protocols.

Unsurprisingly, TCP over the faster InfiniBand link outperforms the cheaper ten gigabit ethernet link by more than a factor of two. Unexpectedly, the native RDMA connection performs worse, even though it outperforms IPoIB in a simple peer-to-peer connection benchmark. This needs further investigation, but we suspect the abstraction of a byte stream connection chosen by Collage for historical reasons is not well suited for remote DMA semantics; that is, one needs to design the network API around zero-copy semantics with managed memory for modern high-speed transports. Both Infiniband connections show significant measurement jitter.

Our RSP multicast implementation performs as expected. Collage starts using multicast to commit new object versions when two or more clients are mapped, since the transmission to a single client is better done using unicast. RSP consistently outperforms unicast on the same physical interface and shows good scaling behavior (2.5 times slower on 16 vs 2 clients on ethernet, 1.8 times slower on InfiniBand). The scaling is significantly better when only one process per node is used (Figure 13, middle: 30% slower on ethernet, nearly flat on InfiniBand). The increased transmission

time with multiple clients is caused by a higher probability of packet loss, which increases significantly when using more than one process per node, as shown in Figure 13, right. This figure plots the number of retransmissions divided by the number of datagrams. Infiniband outperforms ethernet slightly, but is largely limited by the RSP implementation throughput of preparing and queueing the datagrams to and from the protocol thread, which we observed in a profiling analysis.

9 Discussion and Conclusion

RDMA needs further investigation.

Acknowledgments

We would like to thank and acknowledge the following institutions and projects for providing the 3D geometry and volume test data sets: the Digital Michelangelo Project, Stanford 3D Scanning Repository, Cyberware Inc., volvis.org and the Visual Human Project. This work was partially supported by the Swiss National Science Foundation Grant 200021-116329/1.**FIXME: SNF and Hasler as well as EU grants**

We would also like to thank all supporters and contributors of Equalizer, most notably RTT, the Blue Brain Project, the University of Siegen, EVL, Dardo, TBD.

Stefan Eilemann Stefan is the technical manager of the Visualization Team in the Blue Brain Project. His work involves working towards large-scale visualization for Exascale simulations; interactive integration of simulation, analysis and visualization as well as flexible frameworks for data sharing and dynamic allocation of heterogenous resources. He is the CEO and founder of Eyescale and also the lead developer of the Equalizer parallel rendering framework. He has worked in SGI's Advanced Graphics Division as the technical lead of OpenGL Multipipe SDK - a toolkit for scalable, parallel visualization software on shared memory supercomputers. He received his masters degree in Computer Science from EPFL in 2015, and his Engineering Diploma in Computer Science in 1998. He is currently working towards a PhD in Computer Science at the Visualization and MultiMedia Lab at the University of Zurich.

John Doe Biography text here.

Jane Doe Biography text here.

References

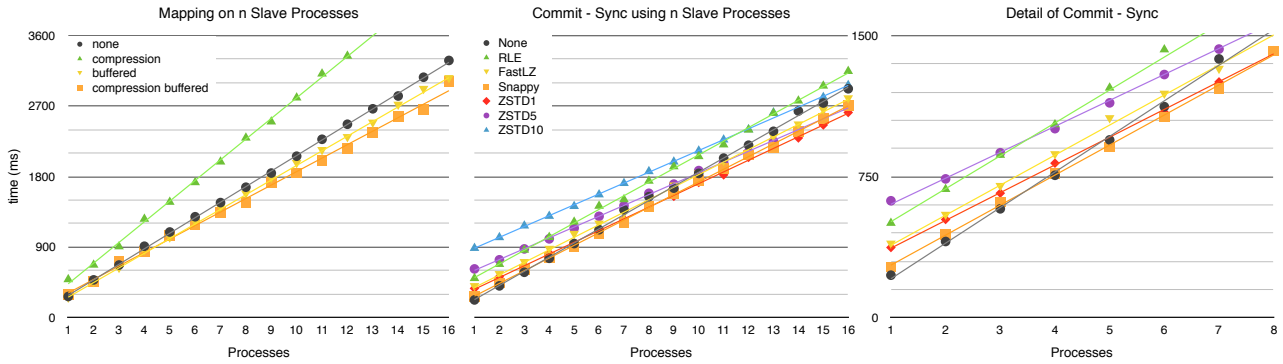


Fig. 10: Object mapping (left) and data synchronization time (middle, detail view right) for the David 2mm data set in Figure 6

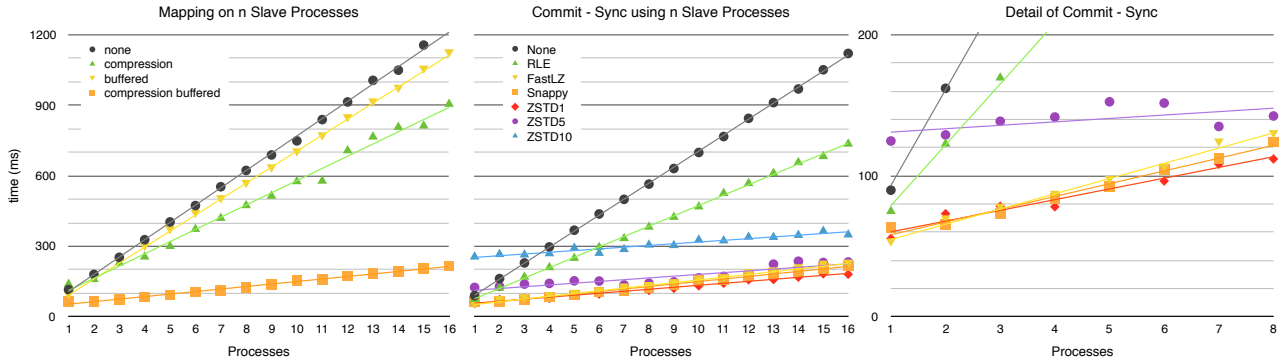


Fig. 11: Object mapping (left) and data synchronization time (middle, detail view right) for the spike data set in Figure 7, left

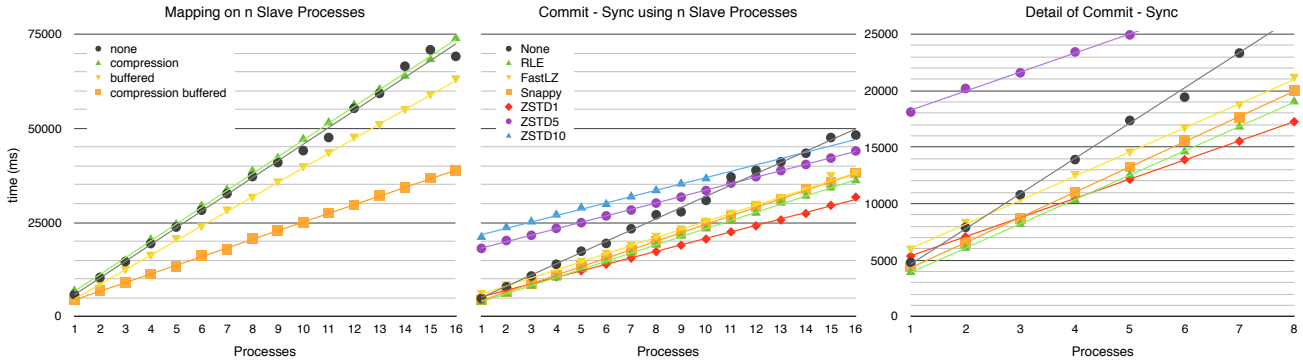


Fig. 12: Object mapping (left) and data synchronization time (middle, detail view right) for the beechnut data set in Figure 7, right

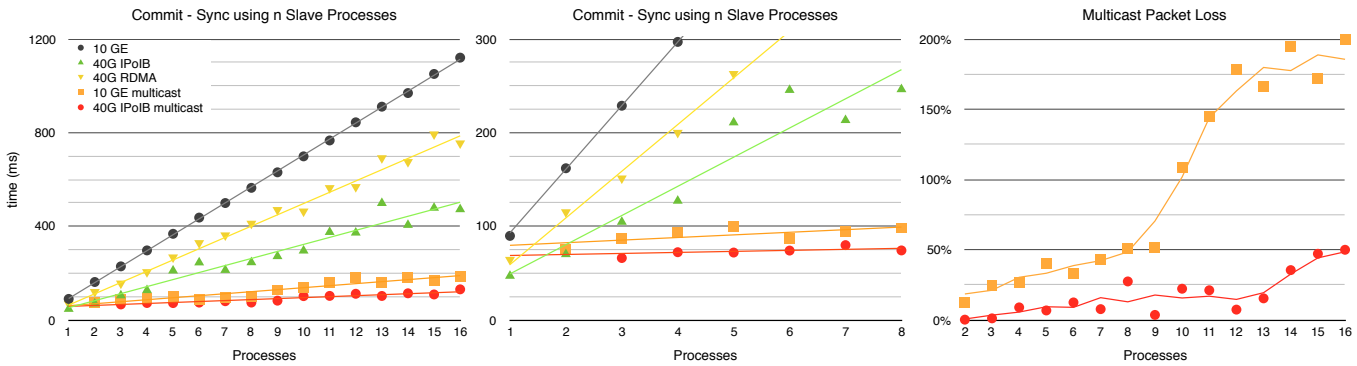


Fig. 13: Object synchronization using different network transports