Eurographics Symposium on Parallel Graphics and Visualization (2006)
Alan Heirich, Bruno Raffin, and Luis Paulo dos Santos (Editors)

Parallel Texture-Based Vector Field Visualization on Curved Surfaces Using GPU Cluster Computers

S. Bachthaler[1], M. Strengert[1], D. Weiskopf[2], and T. Ertl[1]

[1]Institute of Visualization and Interactive Systems, University of Stuttgart, Germany
[2]School of Computing Science, Simon Fraser University, Canada

Abstract

We adopt a technique for texture-based visualization of flow fields on curved surfaces for parallel computation on a GPU cluster. The underlying LIC method relies on image-space calculations and allows the user to visualize a full 3D vector field on arbitrary and changing hypersurfaces. By using parallelization, both the visualization speed and the maximum data set size are scaled with the number of cluster nodes. A sort-first strategy with image-space decomposition is employed to distribute the workload for the LIC computation, while a sort-last approach with an object-space partitioning of the vector field is used to increase the total amount of available GPU memory. We specifically address issues for parallel GPU-based vector field visualization, such as reduced locality of memory accesses caused by particle tracing, dynamic load balancing for changing camera parameters, and the combination of image-space and object-space decomposition in a hybrid approach. Performance measurements document the behavior of our implementation on a GPU cluster with AMD Opteron CPUs, NVIDIA GeForce 6800 Ultra GPUs, and Infiniband network connection.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Viewing algorithms I.3.3 [Three-Dimensional Graphics and Realism]: Color, shading, shadowing, and texture

1. Introduction

Flow visualization is an important topic in scientific visualization, addressing the display and visual analysis of data that may originate from numerical simulations—such as those of computational fluid dynamics—or from measurements during flow experiments. Line integral convolution (LIC) [CL93] and other related texture-based methods are successfully applied to flow fields on planar 2D domains. A key benefit of texture-based techniques is their capability to gradually modify the density of the visual representation. In particular, a representation that densely covers the domain with particle traces overcomes the problem of identifying appropriate seed points for traces. Although LIC can be applied to 3D flow, it typically suffers from problems of clutter and occlusion. Therefore, flow visualization on curved 2D hypersurfaces through the complete 3D data set is an interesting compromise between the completeness and flexibility of the visual representation on the one hand and the reduction of perceptual problems on the other hand.

Recently, there has been significant algorithmic progress in efficiently computing LIC on surfaces via image-space approaches [LJH03, vW03, WE04]. These methods also benefit from leveraging the processing power of GPUs (graphics processing units). In this paper, we adopt the method from [WE04] because it most accurately models the original LIC idea, providing good image quality and temporal coherence for animated visualizations. Unfortunately, the performance of surface LIC is strongly influenced by image resolution, which may lead to rendering rates well below one frame per second (fps) for high-resolution visualizations covering more than a megapixel. Another issue is the restriction of the amount of available texture memory, which limits the maximum size of the 3D flow data set that can be visualized.

In this paper, we address both the performance and memory issues by extending and adapting surface LIC to a GPU cluster, i.e., a cluster computer with GPU-equipped compute nodes. Image-space decomposition is used to scale the visualization speed with the number of GPU nodes, while object-space decomposition leads to a scaling of available GPU memory. Parallelizing this kind of GPU algorithm poses par-

ticular challenges that are addressed in this paper: communication of intermediate results between GPUs, main memory, and different compute nodes; a significantly reduced locality of memory accesses caused by particle traces that cover large spatial regions (i.e., particularly less locality than for parallel volume rendering); dynamic load balancing that takes into account a strongly view-dependent behavior of surface LIC; and the combination of image-space and object-space decomposition. In addition, we document the performance characteristics of our approach by including timings of our implementation.

2. Related Work

Texture-based flow visualization has been an important element of the research in scientific visualization. An overview of the large body of previous work is given in the survey article [LHD*04]. Spot noise [vW91] and line integral convolution (LIC) [CL93] are early examples for texture-based techniques. Recently, advances in texture-based flow visualization are often connected to the increasing performance and functionality of GPUs, which can be used to improve the speed of 2D flow visualization [JEH00, WHE01, vW02].

One approach to extend flow visualization methods from Euclidean space to a curved 2D manifold—such as a surface embedded in 3D space—uses a parametrization of that manifold. For example, LIC can be extended to curved surfaces by computing the convolution in the parameter space of a curvilinear grid [FC95]. One of the problems of such an approach is the need for finding a parametrization, which can be difficult and time-consuming. An alternative approach overcomes issues of parametrization-oriented methods by working in image space. For example, texture advection can be applied in such a way [LJH03, vW03]. In this paper, we follow an improved method that largely adopts an image-space approach and overcomes problems related to temporal coherence and visualization quality [WE04]. All these image-space methods are designed to work with GPUs to achieve a high visualization speed.

There is a large body of research on utilizing cluster systems to improve the performance of typical computer graphics methods like ray tracing, volume rendering, and polygon-based rendering. In general, parallel visualization systems can be classified as *sort-first*, *sort-middle*, or *sort-last* [MCEF94]. In addition, hybrid approaches combining features of different partitioning strategies have been gaining increasingly more attention, e.g., in the form of a hybrid sort-first and sort-last method for parallel polygon rendering [SFLS00] or a hybrid object-space and image-space distribution scheme for volume rendering [GS02].

There is only little previous work on parallel methods specifically related to vector field visualization. Early examples adopt multi-processor workstations, such as SGI's 4D/340 or Cray's T3D, to parallelize particle tracing [BMP*90, Lan94, Lan95]. For texture-based flow visualization, there exist parallel versions of LIC [CL95, ZSH97] that run on massively parallel CPU systems. Similarly, the *GeoFEM* [CFN02] system, which is designed for large shared memory symmetric multiprocessor architectures like the Earth Simulator, contains methods for parallel particle tracking and 3D LIC alike. More recently, Muraki et al. [MLM*03] described an approach using GPU-based cluster systems for rendering volumetric data sets with an extension for visualizing 3D LIC volumes constructed in a preprocessing step.

3. Image-Space LIC on Surfaces

In this section, we give a brief overview of the single-GPU flow visualization method [WE04] that is used in this paper. We adopt a Lagrangian approach to particle tracing as the basis for LIC. The path of a particle is determined by the ordinary differential equation

$$\frac{\mathrm{d}\mathbf{r}(t)}{\mathrm{d}t} = \mathbf{v}(\mathbf{r}(t),t) \quad , \tag{1}$$

where $\mathbf{r}(t)$ describes the position of the particle at time t and $\mathbf{v}(\mathbf{r},t)$ denotes the time-dependent vector field. The positions $\mathbf{r} \in \mathbb{R}^3$ are restricted to locations on the surface embedded in $\mathbb{R}^3$. For a tangential vector field, Eq. (1) leads to curves that stay on the surface.

So far, the vector and point quantities are given with respect to physical space (P-space). The basic idea of image-space methods is to perform the relevant computation in image space. In fact, the image-space operations are performed in normalized device space (D-space), which has extent $[0,1]^3$. D-space is ideal to compute LIC on a per-pixel basis with respect to the image plane in order achieve a largely output-sensitive algorithm and a uniform density on the image plane. On the other hand, there are some aspects that are better represented in P-space, in particular, the 3D noise input for LIC in order to guarantee frame-to-frame coherence under camera motion. The advantages of P-space and D-space representations are combined by computing particle paths in both spaces simultaneously, as illustrated in Figure 1.

Explicit numerical integration, such as a first-order explicit Euler scheme, works with P-space coordinates $\mathbf{r}_P \equiv \mathbf{r}$ and the original tangential vectors $\mathbf{v}_P \equiv \mathbf{v}$ to solve Eq. (1). After each integration step, the corresponding position in D-space is computed. The vector field is no longer given on a P-space but a D-space domain, i.e., we have different representations for the vector components and the associated point on the surface. The modified particle-tracing equation then is

$$\frac{\mathrm{d}\mathbf{r}_P(t)}{\mathrm{d}t} = \mathbf{v}_P(\mathbf{r}_D(t),t) \quad , \tag{2}$$

where $\mathbf{r}_D$ and $\mathbf{r}_P$ represent the same position with respect to D-space and P-space, respectively.

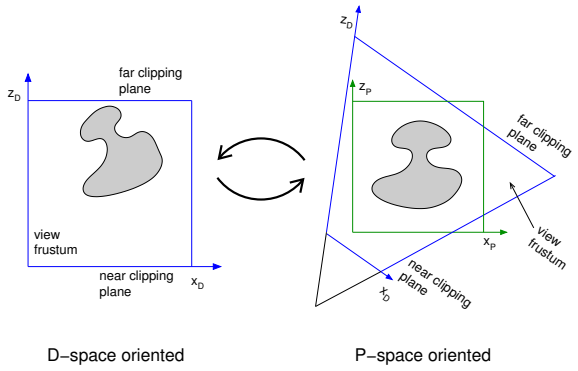

Figure 1: *Coupled D-space (left) and P-space (right) representations of the same scene. © 2004 Weiskopf and Ertl [WE04].*

The crucial step in making the integration process efficient is to reduce the 3D representation of the quantities to a 2D D-space representation when possible. Since flow fields are assumed to live on opaque surfaces, only the closest surface layer needs to be considered. Here, the depth component can be indirectly computed because the depth values of the surface on which the visualization is computed is known.

The algorithm consists of two major parts. In the first part, the 2D textures for starting positions $\mathbf{r}_P$ and the vector field $\mathbf{v}_P$ are initialized by rendering the mesh representation of the hypersurface. The closest depth layer is extracted by the z test. The P-space positions are set according to the surface's object coordinates. The vector field texture is filled by $\mathbf{v}_P$, which originates from slicing through a 3D texture that holds the vector field data set. Because the vector field is usually not tangential from construction, it has to be made tangential by removing the normal component, which is computed according to the normal vectors of the surface mesh. In the second part, Eq. (2) is solved by iterating over integration steps. This part works on the 2D x–y sub-domain of D-space, and it successively updates the coordinates $\mathbf{r}_P$ and $\mathbf{r}_D$ along the particle traces, while simultaneously accumulating contributions to the convolution integral.

The implementation of the complete visualization process can be split into three stages: the *projection stage*, which projects the surface geometry and the vector field onto the image plane, the *LIC stage*, which computes the line integral on the image plane, and the *blending stage*, which combines a LIC image with the rendered image of the surface geometry. All three stages are implemented by vertex and fragment programs to make use of the high processing speed of GPUs.

The projection stage produces three 2D textures as intermediate results: the projected vector field $\mathbf{v}_P$, the start point for particle tracing $\mathbf{r}_P$, and the rendered image of the illuminated surface. These three textures are filled in a single rendering pass by using multiple render targets (MRTs). The performance of the projection stage is comparable to the performance of rendering the surface with illumination being enabled—it just adds a few instructions that project the vector field and write out the initial coordinates for particle tracing.

In contrast, the LIC stage is computationally more expensive. This stage uses intermediate results from the projection part to compute the line integral. It solves the particle tracing Eq. (2) to advance positions along streamlines. Simultaneously, contributions to the line integral are accumulated along the streamline. The input noise is stored in a 3D texture. Potential aliasing due to perspective foreshortening is avoided by an adapted version of MIP mapping (see [WE04] for details). The result of this stage is written to a 2D texture that holds the gray-scale LIC image on the image plane. Particle tracing and integral accumulation are implemented in a shader loop that advances along the streamline. Typically, the number of iterations is between 40 and 300. Therefore, the overwhelming performance costs are associated with the LIC stage. Note that the LIC computation is only performed for pixels that are covered by the projected surface geometry, i.e., fragment processing is skipped for background pixels by means of the early z test. The z values for this masking are obtained from the projection stage. Masking leads to a rendering run time that is proportional to the number of visible pixels, i.e. output sensitivity is achieved to a large extent.

Finally, the blending stage combines the result of the LIC stage and the lit surface generated in the projection stage by blending and modulation. In this way, both the LIC texture and the surface geometry are visualized at the same time.

This flow visualization technique completely re-computes the LIC image for each frame. Therefore, the rendering performance is not affected by deforming or changing the surface geometry, or by moving the camera. In this way, this approach is suited for interactive applications in which visualization parameters can be rapidly changed by the user.

4. Parallel GPU-Based Visualization

In order to utilize the cluster environment in terms of rendering speed as well as memory scalability, we employ a hybrid sort-first sort-last rendering scheme.

Our software architecture consists of two major elements: a user application and distributed render clients. The user application acts as the frontend that presents the final image of the distributed rendering and handles user inputs. For rendering, basic graphics functionality is sufficient and no special hardware requirements need to be met, in particular not the ones necessary for visualizing the vector data. The frontend is connected to the PC cluster via TCP/IP allowing the user to visualize data from a remote location. Each node of the cluster system runs an instance of the render client that visualizes parts of the final image depending on the user-

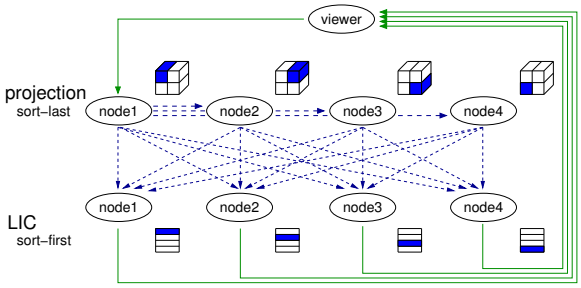

Figure 2: *Overview of the communication architecture. The viewer communicates with the cluster using TCP/IP (green solid lines). Cluster-internal data transfer is driven by MPI (blue dashed lines). In the first part of the algorithm object-space partitioning is applied to the projection of the vector field. Image-space partitioning is used for the LIC stage.*

specified parameters, the number of cluster nodes used, and the partitioning scheme.

The communication scheme for parallel visualization is illustrated in Figure 2. The following steps are involved: First, the viewer application sends a render request to a single cluster node. Such a render request contains all information necessary to render a single frame, e.g. camera parameters, lighting conditions, and LIC parameters. Second, the request is then broadcast to all the other remaining cluster nodes using MPI (message passing interface). We adopt this two-level communication in contrast of a direct broadcast of the user application in order to minimize the amount of data to be sent over a possibly narrow-banded TCP/IP interconnection. Third, every cluster node processes the request and renders an output image according to the given parameters. Image-space partitioning is based on stripes dividing the framebuffer into separate areas each of which is assigned to a different render node (see Section 5). Additionally, the projection of the vector field onto the hypersurface can optionally make use of object-space partitioning to exploit the scalability of texture memory in a GPU-based cluster environment (see Section 6). Finally, the content of the framebuffer within the assigned stripe is read back on each node and sent to the user application. To reduce the overall amount of data to be transferred over TCP/IP, we compress the data using LZO real-time compression before sending. Note that a two-level approach for communication would not reduce the amount of transferred data in case of sort-first partitioning schemes, but would introduce additional overhead. The user application decompresses the image tiles and composes them into a final image that is shown to the user.

5. Image-Space Decomposition

Our method to accelerate surface LIC makes use of a sort-first approach. The image plane is split into sub-images that are rendered on separate cluster nodes. Ideally, the amount of work per node is reduced to $1/n$, where n is the number of sub-images (i.e. number of cluster nodes).

5.1. Partitioning of Image Space

We partition the image space with a collection of horizontal stripes. This partitioning affects the projection and LIC stages alike.

For the projection stage, we do not separate the surface geometry into different pieces for parallel rendering because the size of the surface mesh does not pose a rendering bottleneck in our case. Typical visualization meshes consist of fewer than a million triangles. Therefore, each node holds a copy of the complete surface geometry. For the time being, we also assume that the vector field data is completely replicated on each node. (This restriction is overcome by object-space partitioning in Section 6.) In the projection stage, the viewport needs to be adjusted to produce intermediate results only for the image-space stripe that is associated with a respective render node. This is achieved by modifying the view frustum of the camera. The parts of the geometry that are not visible in the stripe are removed by view frustum clipping.

The LIC stage works directly in image space. A render node only processes those pixels that lie within its respective stripe. In other words, the LIC computation is parallelized with a sort-first approach because the computational domain is partitioned without any overlap. Similarly, the blending stage works directly on a per-pixel basis in the image stripe.

The construction of the final image is reduced to a simple tiling of intermediate image stripes. Here, the viewer application needs an offset in addition to the content of a respective stripe to place the received framebuffer content of a stripe at the proper position in the final image. To calculate this offset, only the heights of the stripes that are placed below the current stripe are needed. As an example, the lowermost stripe does not need any offset (this is because the framebuffer "begins" at the lower left corner). The stripe on top of this stripe does need an offset equal to the height of the first stripe, etc.

5.2. Continuous Border Transitions

A problem arises when the above stripe approach is used in combination with particle tracing in the LIC stage: A particle trace that starts within one stripe may leave that stripe and enter a neighboring stripe. In other words, particle tracing breaks the per-pixel locality that has to be assumed for a naive image decomposition. Once a particle trace leaves a stripe, it has no longer access to required vector field information. This leads to clearly visible border artifacts between two stripes, as illustrated in Figure 3. The flow is interrupted at the border and does not continue seamlessly into the next stripe.

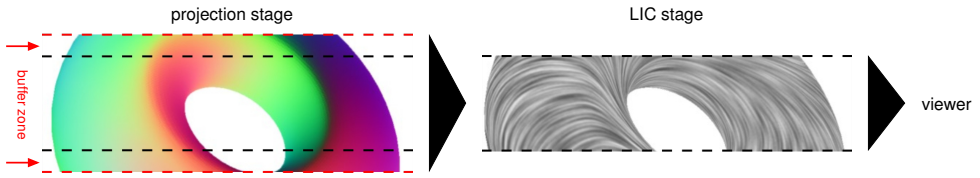

Figure 4: *The buffer zone around an image stripe, as constructed in the projection stage and subsequently used by the LIC stage.*

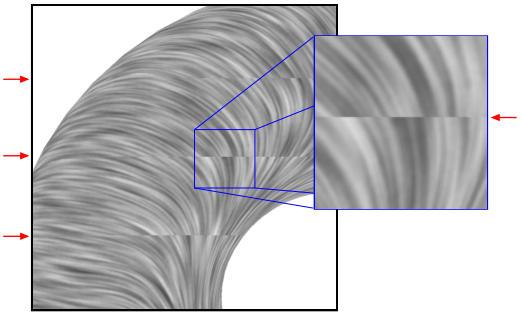

Figure 3: *Artifacts at stripe boundaries caused by missing vector field data. Red arrows highlight stripe borders.*

This problem can be overcome by increasing the spatial domain of the available vector field data: The projection stage has to produce stripes with an additional area at the upper or lower parts of the stripe. Of course, this overlapping area—or "buffer zone"—is only needed if a stripe has a neighbor at the corresponding border, i.e., the uppermost and lowermost stripes only need one buffer zone at the lower or upper borders, respectively. Figure 4 shows how the buffer zone is constructed in the projection stage and subsequently used by the LIC stage. The buffer zone is only needed during particle tracing and can be ignored for later process stages of the visualization process. In particular, the starting points for LIC traces (in the LIC stage), the blending stage, and the readback of intermediate results from the framebuffer are based on the original stripe area in order to avoid unnecessary computations.

The size of the buffer zone should be chosen cautiously because it can unnecessarily slow down the rendering process if set too big. Of course, if the buffer zone is too small, the previously mentioned error remains visible. The size of the buffer zone is determined by

$$s_{\text{buffer}} = \frac{n_{\text{convolution}}}{2} v_{\max} \Delta t \quad ,$$

where $n_{\text{convolution}}$ is the number of convolution steps (the size of the discretized LIC filter kernel), $v_{\max}$ is the maximum velocity magnitude in the data set, and Δt is the step size used for discretizing the LIC computation. The factor $1/2$ reflects the fact that a symmetric filter kernel is used, i.e. particle traces follow one half of the filter kernel in both directions. The value s_{buffer} is the maximum distance along a particle trace in image space and describes the worst case when a streamline is perpendicular to the stripe border. For consistency, the parameters $v_{\max}$ and Δt need to be specified with respect to image space as well. Note that $v_{\max}$ is readily available for many applications. For example, streamline-based LIC assumes that the vector magnitude is normalized to unit length. As another example, the representation of vector data in 8-bit texture formats gives a direct bound for the vector magnitude. The buffer zone is designed for the worst case scenario with a conservative estimate for the particle-trace length. More sophisticated estimates (e.g. by considering vector field direction) might lead to a reduced size of the buffer zone, however, at the cost of a more time-consuming computation of the estimate.

5.3. Load Balancing

In order to achieve an optimal overall performance, every node of the cluster should be assigned an equal share of the workload. So far, image-space partitioning relies on static stripes that divide the viewport in equally sized areas. However, the determining factor for performance is not the size of the area in image space, but the actual number of fragments of the surface geometry that need to be processed. It is obvious that a node with a stripe fully covered by the surface model is far slower than a node assigned to a completely empty region. To overcome this problem we adopt a dynamic adjustment of the height of the stripes depending on the associated workload. We use two alternative methods to determine the workload: a timing-based method that actually measures the workload, and a pixel-counting approach that provides an estimate for the computation time based on the number of pixels.

For the timing-based method, the time needed to finish rendering is continuously measured on every cluster node. These timings are then gathered for the current frame and used to adjust the stripe heights for rendering the subsequent frame: The stripe heights are modified relative to the speed differences between nodes. The underlying assumption is that the timings are a good estimate for the rendering times of the following frame, which is reasonable when there is temporal coherence for rendering. The timing-based ap-

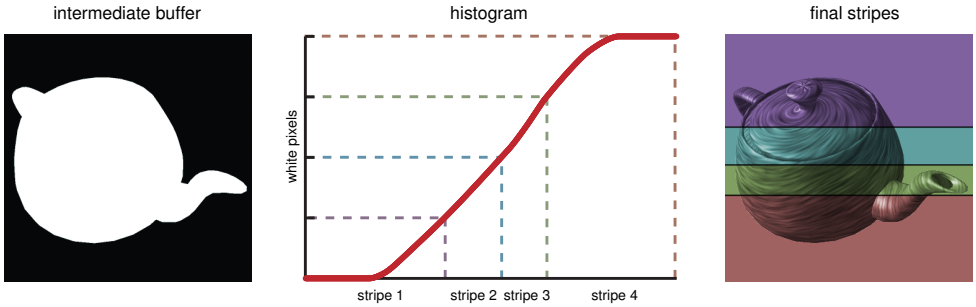

Figure 5: *Computing the estimate for the pixel-counting approach. Pixels of the downsampled rendering (left) are stored in an accumulated histogram (center). The stripe sizes (along the horizontal axis of the histogram) are determined by inverting the accumulated histogram for equally sized pixel intervals (along the vertical axis). The stripes are shown to the right.*

proach can be implemented with almost no overhead or additional processing. It just involves taking the start and end times for rendering, and transferring those times when the intermediate images are sent between nodes for final rendering. The main drawback is that the optimal size for the stripes is typically not achieved completely. This is to some extent due to inaccuracies of time measurements in connection with high frame rates. But more importantly, this issue is related to the partially violated assumption of perfect temporal coherence. If the content of the framebuffer changes rapidly, temporal coherence between consecutive frames diminishes and the load balancing is less effective.

The second approach uses a pixel-counting algorithm to avoid the aforementioned drawbacks. The idea is to obtain an estimate of the framebuffer content before rendering. With a good estimate, the workload can be adjusted in a way that every node gets the same number of fragments of the geometry model assigned for LIC processing. Since the workload is mainly dependent on the amount of fragments the vector fields gets projected on, this approach allows for an effective load balancing. Since computing the estimate has to be fast to minimize overhead, we render a downsampled image of the geometry model first. In order to assure the accuracy of the estimation a tradeoff between the inherent overhead and the used quality needs to be made. For all our tests and performance measurements a factor of one-fifth for each dimension was chosen. The geometry is rendered completely unshaded to further speed up processing. The result of this render process is read from the framebuffer to generate an accumulated histogram. For each row, the number of pixels covered by the geometry model is determined and summed up consecutively. Figure 5 shows the relationship between framebuffer content and the accumulated histogram. We divide the total number of pixels by the number of cluster nodes to obtain the ideal number of pixels per node. Using the inverse of the histogram, equally sized intervals (where the interval size corresponds to the number of pixels per node) are mapped to actual stripe heights in the framebuffer. The pixel-counting approach is computed with the current camera parameters and, thus, is not affected by changes of framebuffer contents. The main disadvantage is the slightly higher rendering overhead, especially for large surface meshes.

6. Object-Space Partitioning

Not only can a cluster be used to scale rendering speed, but also to scale available memory because each node provides some fixed amount of memory. The goal to achieve the best performance by using multiple processors is often on a par with the goal to visualize very large vector fields. To display huge vector fields, the considerably larger combined texture memory of the PC cluster is used.

The vector field has to be partitioned because the texture memory of a PC cluster is not available in one piece—it is distributed memory. We adopt a bricking approach known from parallel volume rendering: The vector field is divided into bricks, and each of these bricks corresponds to a subvolume of the vector field. A single cluster node works on its assigned brick. The brick size (or the number of nodes) should be chosen so that the vector field data of a brick fits in the texture memory of a single node's GPU.

Only the projection stage is affected by bricking because the 3D vector field is only used in that part of our surface LIC algorithm. The following modifications need to be incorporated. First, it has to be ensured that a cluster node generates a projected vector field only within its own brick. Six brick-aligned clip planes are used to cut the surface geometry away for regions outside the brick. Second, a compositing step needs to be included to reconstruct the full image-space based vector field. Compositing is distributed among nodes, with the same stripe-based organization that is used for the LIC stage. Therefore, every node has to send its content of the corresponding stripe to the cluster node that is responsible for this stripe. Here, we must pay attention that the nodes send stripes including the buffer zones. In this sort-last approach, the intermediate vector field images are composited in a back-to-front order. Since the surface geometry

Table 1: *Performance with varying amount of nodes, rendered on a 800×800 viewport. Numbers in brackets denote speedup.*

number of cluster nodes	1	2	3	4	5	6	7	8
FPS static stripes	2.23	3.86 (1.7)	2.82 (1.3)	4.01 (1.8)	4.12 (1.8)	5.07 (2.3)	5.06 (2.3)	6.29 (2.8)
FPS dynamic time-based	-	4.00 (1.8)	5.62 (2.5)	6.67 (3.0)	7.91 (3.5)	9.10 (4.1)	10.12 (4.5)	10.89 (4.9)
FPS dynamic pixel-based	-	3.83 (1.7)	5.21 (2.3)	6.29 (2.8)	7.25 (3.3)	8.14 (3.7)	8.95 (4.0)	9.67 (4.3)

is opaque, alpha blending is not needed. In fact, similarly to painter's algorithm, incoming non-background pixels just overwrite existing pixels. The result of this compositing is a complete projection of the vector field for that stripe.

The LIC stage is not affected by these modifications because it uses only the result of the projection part. Therefore, the object-space partitioning of the data set can be directly combined with the image-space decomposition for the LIC computations.

7. Results and Discussion

All measurements were conducted on a GPU-based cluster with eight render nodes. Each node runs two AMD Opteron processors at 2.18 GHz and is provided with 4 GB of system memory. For rendering, all nodes have an NVIDIA GeForce 6800 Ultra with 256 MB of texture memory installed. The cluster's internal communication is driven by MPI over an Infiniband interconnection that provides low latency times and data transfer rates of up to 800 MB per second. In our test environment, the PC running the viewer application is connected to the cluster using a Gigabit Ethernet network.

To demonstrate the scaling behavior of our system we first show the results obtained with an increasing number of nodes used for rendering on a 800×800 sized viewport using a 128^3 sized vector field. During this test series no object-space partitioning was carried out in order to provide comparability and to avoid effects caused by a compositing stage or a corresponding communication scheme. The surface is generated by rendering the GLUT teapot that rotates around the main axis while the measurement was taken (see color plate Figure I). With this constant change of the rendered image, we try to simulate reproducible user interaction and allow for measuring the effects of the dynamic stripe adaption under realistic conditions. Table 1 documents our results for three test series, either using a static distribution with equally sized areas in image-space or using one of the two dynamic load balancing techniques from Section 5.3.

The speedup obtained in the static case using all eight cluster nodes is only a factor of 2.82, compared to the single node setup. This small speedup is mainly caused by a highly imbalanced distribution of the workload throughout the cluster. With increasing number of nodes, the top and bottom stripes receive a rapidly decreasing number of fragments of the surface model, while the number of fragments for the center stripes decreases only slowly. Adding dynamic load balancing significantly improves the performance of the complete system. For both dynamic load balancing techniques, an 8-node configuration achieves a speedup of over 4.3, with an average frame rate of approximately 10 fps. In all measurements, the timing-based method performed better than the pixel-counting approach, which is due to the higher overhead for computing the pixel-counting estimate. However, for other tests with less temporal coherence between frames, we already experienced that the performance gap between the timing-based and pixel-counting approaches closed. If more parameters change in-between frames, such as the geometry of the surface, we expect to achieve a performance advantage with the pixel-counting approach.

Adding the object-space partitioning scheme to the projection stage allows us to increase the size of the visualized vector field at the cost of additional communication and compositing. For performance measurements, we used a 512^3 vector field with an overall data amount of 1.28 GB. Distributing the data using all eight cluster nodes leads to nearly 200 MB of texture data assigned to each cluster node, which is close to the texture limit of the used GPUs. Us-

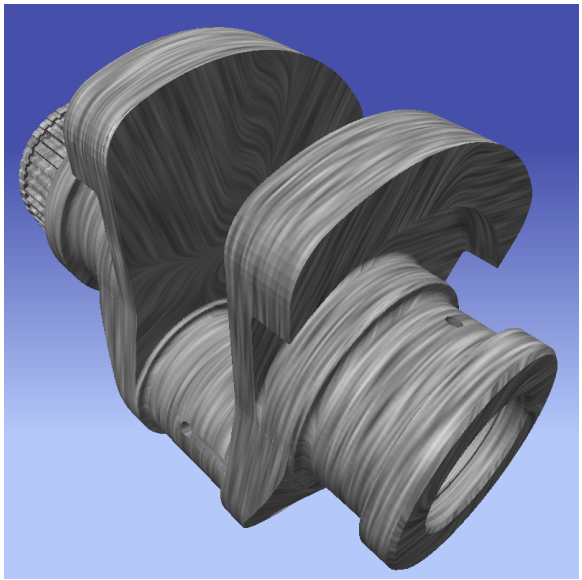

Figure 6: *Resulting image of distributed rendering with eight GPU nodes. The geometric object was modeled by Sam Drake and tesselated by Amy Gooch and Peter-Pike Sloan.*

ing the same parameters and geometry as stated above, we achieved a rendering rate of 3.08 fps for static image-space load balancing and 3.79 fps for both dynamic load balancing methods (with eight nodes). For comparison, the 128^3 sized data set renders at 3.11 fps and 3.90 fps, respectively. As described, using object-space partitioning requires an additional compositing step for the projected vector field, which is also the main reason for the measured drop in performance. Taking this overhead into account we achieve an almost optimal scaling behaviour for the GPU memory.

Figure 6 shows the result of flow visualization on a more complex surface mesh with 25000 polygons and a 512^3 sized vector field. When rendering on a 800×800 viewport with eight nodes, we obtained 2.22 fps.

8. Conclusions and Future Work

We have presented a parallelized algorithm for surface LIC on a GPU-cluster architecture. The key elements are a sort-first strategy with image-space decomposition to distribute the workload for expensive LIC computations, and a sort-last approach with an object-space partitioning of the vector field to increase the available GPU memory. We have demonstrated that the sort-first and sort-last methods can be combined to a hybrid technique in order to benefit from both of their advantages. We have also addressed issues such as dynamic load balancing or the reduced locality of particle tracing, which can be solved by introducing buffer zones. Our performance measurements have shown that our implementation provides an acceptable performance scaling behavior for moderately sized GPU clusters. An equally important benefit is that object-space decomposition enables us to visualize large vector field data sets interactively; and memory scaling comes at a negligible performance cost.

For future work, parallel rendering could be extended to the projection of the surface geometry itself in order to visualize extremely large surface meshes.

References

[BMP*90] BANCROFT G. V., MERRITT F. J., PLESSEL T. C., KELAITA P. G., MCCABE R. K., GLOBUS A.: FAST: a multi-processed environment for visualization of computational fluid dynamics. In *Proc. IEEE Visualization* (1990), pp. 14–27.

[CFN02] CHEN L., FUJISHIRO I., NAKAJIMA K.: Parallel performance optimization of large-scale unstructured data visualization for the Earth Simulator. In *Proc. Eurographics Workshop on Parallel Graphics and Visualization (EGPGV)* (2002), pp. 133–140.

[CL93] CABRAL B., LEEDOM L. C.: Imaging vector fields using line integral convolution. In *Proc. ACM SIGGRAPH* (1993), pp. 263–270.

[CL95] CABRAL B., LEEDOM L. C.: Highly parallel vector visualization using line integral convolution. In *Proc. Seventh SIAM Conference on Parallel Processing for Scientific Computing (PPSC)* (1995), pp. 802–807.

[FC95] FORSSELL L. K., COHEN S. D.: Using line integral convolution for flow visualization: Curvilinear grids, variable-speed animation, and unsteady flows. *IEEE Transactions on Visualization and Computer Graphics 1*, 2 (1995), 133–141.

[GS02] GARCIA A., SHEN H.-W.: An interleaved parallel volume renderer with PC-clusters. In *Proc. Eurographics Workshop on Parallel Graphics and Visualization (EGPGV)* (2002), pp. 51–59.

[JEH00] JOBARD B., ERLEBACHER G., HUSSAINI M. Y.: Hardware-accelerated texture advection for unsteady flow visualization. In *Proc. IEEE Visualization* (2000), pp. 155–162.

[Lan94] LANE D. A.: UFAT: a particle tracer for time-dependent flow fields. In *Proc. IEEE Visualization* (1994), pp. 257–264.

[Lan95] LANE D. A.: Parallelizing a practicle tracer for flow visualization. In *Proc. Seventh SIAM Conference on Parallel Processing for Scientific Computing (PPSC)* (1995), pp. 784–789.

[LHD*04] LARAMEE R. S., HAUSER H., DOLEISCH H., VROLIJK B., POST F. H., WEISKOPF D.: The state of the art in flow visualization: Dense and texture-based techniques. *Computer Graphics Forum 23*, 2 (2004), 143–161.

[LJH03] LARAMEE R. S., JOBARD B., HAUSER H.: Image space based visualization of unsteady flow on surfaces. In *Proc. IEEE Visualization* (2003), pp. 131–138.

[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.

[MLM*03] MURAKI S., LUM E. B., MA K.-L., OGATA M., LIU X.: A PC cluster system for simultaneous interactive volumetric modeling and visualization. In *Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)* (2003), pp. 95–102.

[SFLS00] SAMANTA R., FUNKHOUSER T., LI K., SINGH J. P.: Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *Proc. Eurographics / ACM SIGGRAPH Workshop on Graphics Hardware* (2000), pp. 97–108.

[vW91] VAN WIJK J. J.: Spot noise – texture synthesis for data visualization. *Computer Graphics (Proc. ACM SIGGRAPH 91) 25* (1991), 309–318.

[vW02] VAN WIJK J. J.: Image based flow visualization. *ACM Transactions on Graphics (Proc. ACM SIGGRAPH 2002) 21*, 3 (2002), 745–754.

[vW03] VAN WIJK J. J.: Image based flow visualization for curved surfaces. In *Proc. IEEE Visualization* (2003), pp. 123–130.

[WE04] WEISKOPF D., ERTL T.: A hybrid physical/device-space approach for spatio-temporally coherent interactive texture advection on curved surfaces. In *Proc. Graphics Interface* (2004), pp. 263–270.

[WHE01] WEISKOPF D., HOPF M., ERTL T.: Hardware-accelerated visualization of time-varying 2D and 3D vector fields by texture advection via programmable per-pixel operations. In *Proc. Vision, Modeling, and Visualization (VMV)* (2001), pp. 439–446.

[ZSH97] ZÖCKLER M., STALLING D., HEGE H.-C.: Parallel line integral convolution. *Parallel Computing 23*, 7 (1997), 975–989.

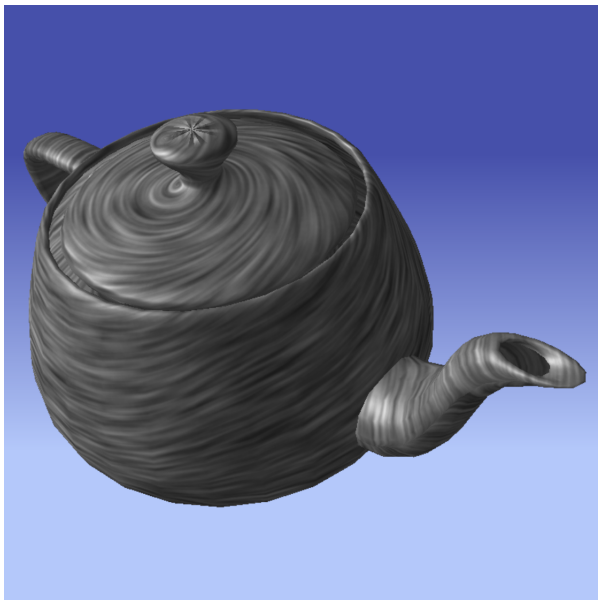

Figure I: *Image taken from the the performance measurement series.*

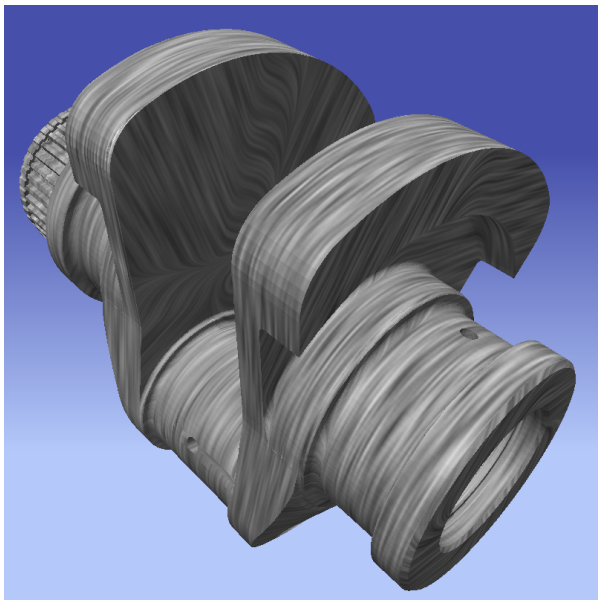

Figure II: *Resulting image of distributed rendering with eight GPU nodes.*