Multiblock Data Parallel Surface LIC

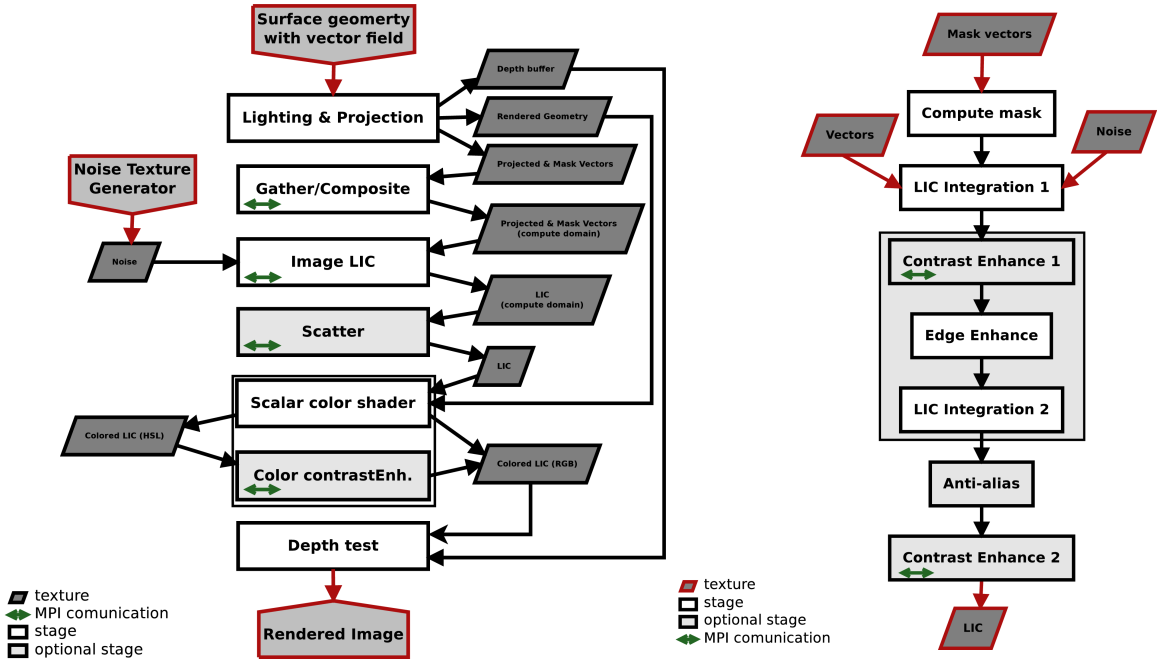
Burlen Loring

September 2, 2013

Abstract

1 Features for interactive data exploration

1.1 Pipeline



(a) (b)

The pipeline employed by the vtkSurfaceLICPainter, including the new contrast enhancement stages, is shown in the following schematic:

```
noise
  |
  [ PROJ LIC2D COLOR (CCE) ]
  |                        |
vectors                surface LIC
```

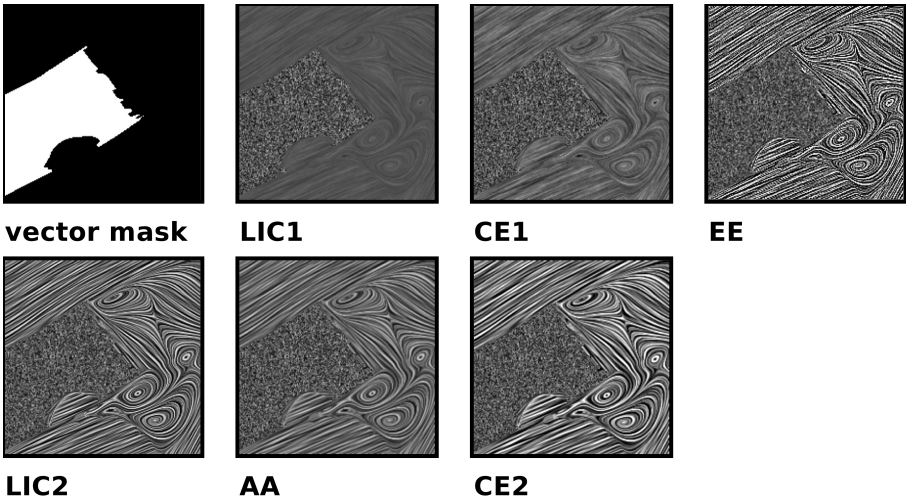


Figure 1: Line integral 2D stages

Where, the PROJ stage renders geometry, projecting vectors onto the surface, and the the screen; the LIC2D stage computes the LIC driving the vtkLineIntegralConvolution2D class; the COLOR stage combines LIC with scalar colors shading the surface geometry; and the CCE stage applies the new color contrast enhancement. Optional stages are enclosed in ().

In order to provide better interaction and faster response as parameters are varied during interactive exploration, each stage renders to a frame-buffer which is cached. This enables un-necessary stages to be skipped during interaction. For example, if an integrator parameter is modified then the geometry rendering and projection stage can be skipped. For large data this optimization can result in orders of magnitude speed up during interaction.

The pipeline employed in vtkLineIntegralConvolution2D, including the new contrast enhancement stages, is shown in the following schematic:

```
noise texture
         |
         [ LIC ( ( CE ) HPF LIC ) ( AA ) ( CE ) ]
         |                                       |
vector texture                              LIC'd image
```

Where LIC is the line integral convolution stage, HPF is a high-pass filter stage, CE is a contrast enhancement stage, and AA is an anti-aliasing stage. The optional stages are enclosed in (). Nested optional stages depend on the execution of their parent stage.

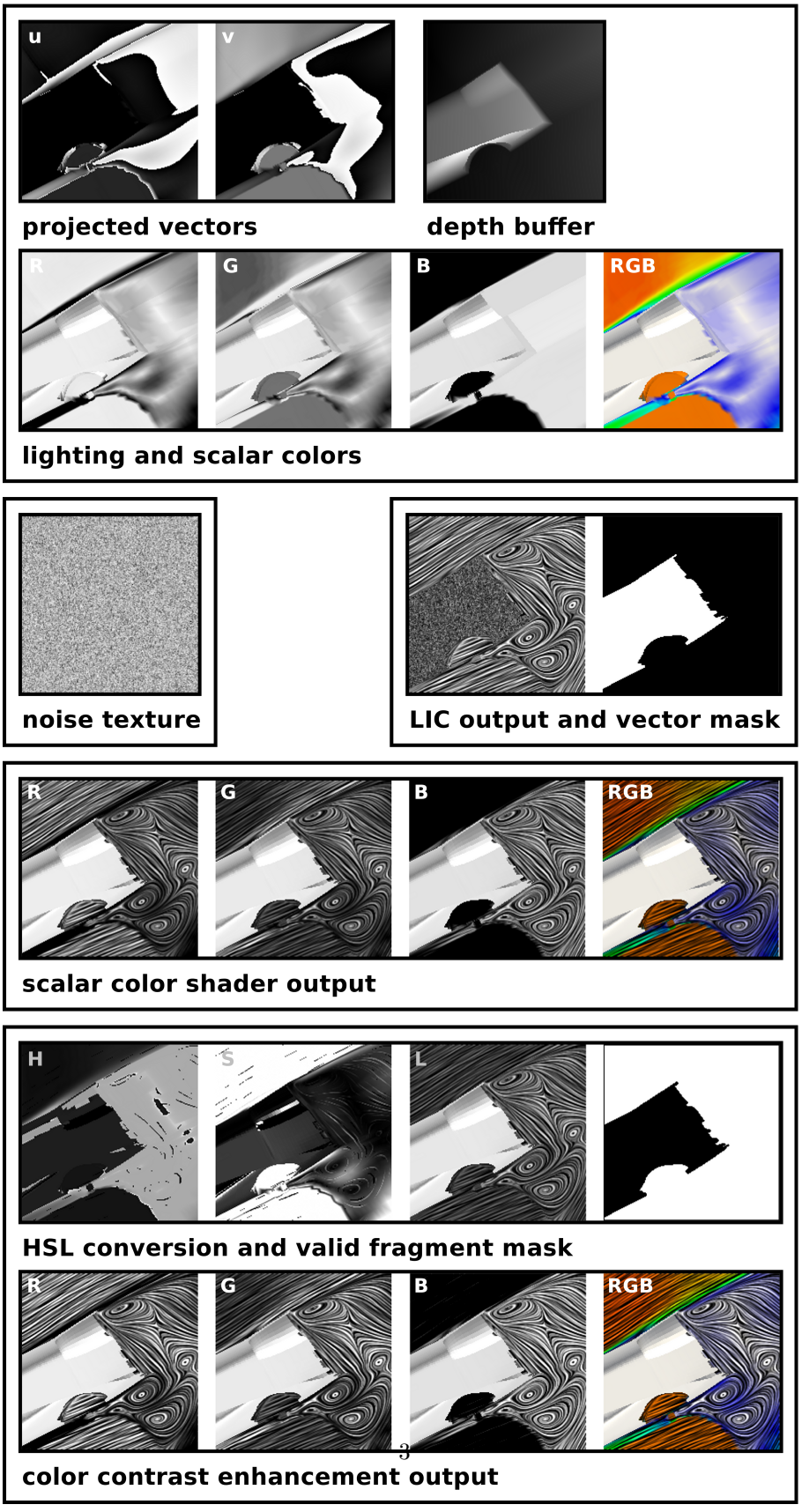


Figure 2: Surface LIC Painter stages

1.2 Noise generator

The streak characteristics and look of the final LIC'ed image vary widely based on the input dataset, resolution of the final image, and choice of surface geometry. One important factor in determining the streaking characteristics and overall look of the final image are the characteristics of the input noise texture. The size of the noise pixels, the size of the noise texture, the contrast and dynamic range of the noise texture, all play a role in the look of the final image. Using the same noise texture produces markedly different results on different datasets and even on the same dataset at different screen resolutions. It's unlikely that a single noise texture will work well in all cases.

In order to provide more control over the streaking characteristics and general look of the final LIC image I have added customizable noise texture generator. The noise texture can be modified based on the following parameters:

GenerateNoiseTexture Select between the default static noise texture or generate a custom noise texture using the following parameters.

NoiseType Select between Gaussian, uniformly distributed noise, or Perlin noise.

NoiseTextureSize Select the number of pixels in a side of the square noise texture.

NoiseGrainSize Select the number of pixels each side of the square noise value spans.

MinNoiseValue Set the lowest color in the noise texture, the default, 0, is black.

MaxNoisevalue Set the highest color in the noise texture, the default, 1, is white.

NumberOfNoiseLevels Set the number of noise colors.

ImpulseNoiseProbability Impulse noise noise can be generated with a probability less than 1.

ImpulseNoiseBackgroundColor The color to use for un-touched pixels.

NoiseGeneratorSeed Modify the seed value of the random number generators.

Changing the noise texture gives one greater control over the look of the final image. By default the original static 200x200 noise texture (see VTKData/Data/Data/noise.png) is used.

1.3 Scalar color shading

The current approach for shading the LIC'ed image with scalar colors blends the scalar colors with the LIC according to

$$c_{ij} = L_{ij} * I + S_{ij} * (1 - I) \tag{1}$$

where the indices i, j identify a specific fragment, c is final RGB color, L is LIC gray scale value, S is the scalar RGB color, and I is a constant ranging from 0 to 1, with a default of 0.8. Decreasing I to obtain brighter colors diminishes the intensity of the LIC, and vise versa. When colors are bright the LIC is difficult to see. Currently, the best results are obtained by sacrificing on both fronts.

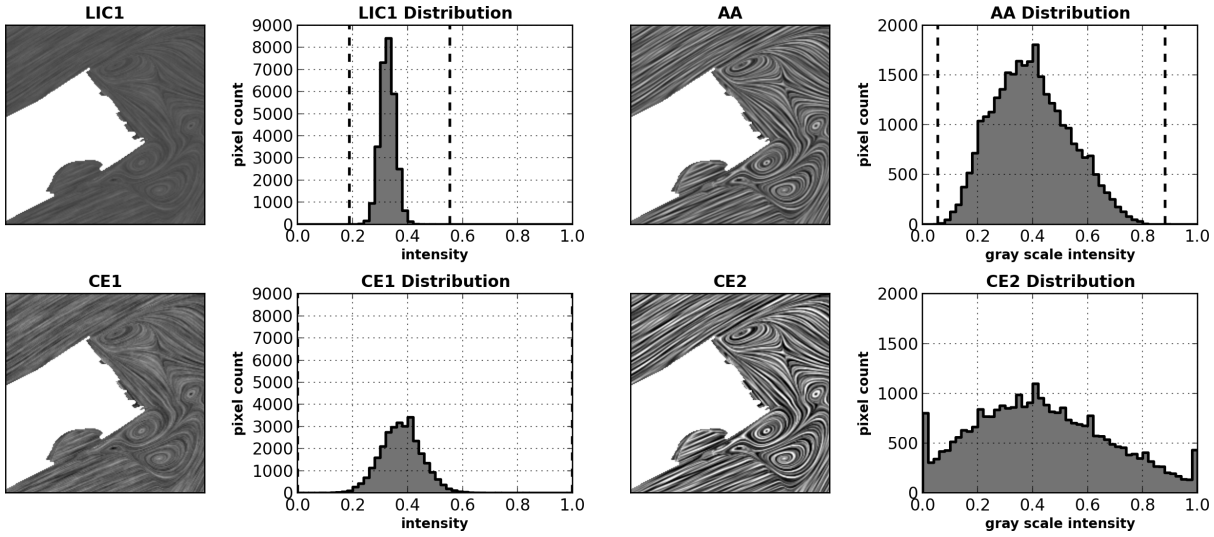
The purpose of the new scalar color shader is to provide both bright and intense scalar colors without diminishing the visibility of the LIC itself. The shader is implemented by the following equation:

$$c_{ij} = (L_{ij} + f) * S_{ij} \tag{2}$$

where the indices i, j identify a specific fragment, c is final RGB color, L is LIC gray scale intensity, S is the scalar RGB color, and f is a biasing parameter, typically 0, that may be used for fine tuning.

When $f = 0$, the typical case, colors are transfered directly to the final image where the LIC is 1, and a linearly scaled transfer of scalar colors where LIC gray scale color is less than one down to 0, where the final color is black. The bias parameter f may be set to small positiver or negative values between -1 and 1 to increase or decrease LIC values uniformly resulting in brighter or darker images. When $f \neq 0$ final fragment colors, c, are clamped such that $0 \leq c \leq 1$.

1.4 Contrast enahancement



(a) (b)

The convolution process tends to decrease both contrast and dynamic range, often producing dark and dull results. Figure **??**.a illustrates this issue. The narrowing of dynamic range and diminishing of contrast are consequences of the convolution operation which convolves a noise texture with a vector field. These reductions are made worse by the use of Gaussian noise because by definition centrally concentrates its values about a mid gray tone.

In order to counteract this, optional contrast enhancement stages have been added. The new stages increase the dynamic range and contrast without modifying the patterns that emerge from the LIC process or modifying the scalar colors. VTK's surface LIC implementation is provided by two classes, the vtkSurfaceLICPainter and vtkLineIntegralConvolution2D. The vtkSurfaceLICPainter, projects vectors, sets up noise texture, and passes control to the vtkLineIntegralConvolution2D to generate a LIC, after which it shades scalar colors combined with the generated LIC onto the surface geometry. Contrast enhancement stages have been added to both classes.

The contrast enhancement stages are implemented by histogram stretching of the gray scale colors in the LIC'ed image as follows:

$$c_{ij} = \frac{c_{ij} - m}{M - m} \tag{3}$$

where, the indices i, j identify a specific fragment, c is the fragment's gray scale color, m is the gray scale color value to map to 0, M is the gray scale color value to map to 1. When the contrast enhancement stage is applied on the input of the high-pass filter stage, m and M are always set to the minimum and maximum gray scale color of all fragments. In the final contrast enhancement stage m and M take on minimum and maximum gray scale colors by default but may be individually adjusted by the following

set of equations:

$$m = min(C) + F_m * (max(C) - min(C)) \tag{4}$$

$$M = max(C) - F_M * (max(C) - min(C)) \tag{5}$$

where, $C = \{c_{00}, c_{01}, ..., c_{nm}\}$, are all of the gray scale fragments in the LIC image and F_m and F_M are adjustment factors that take on values between 0 and 1. When F_m and F_M are 0 minimum and maximum are gray scale values are used. This is the default. Adjusting F_m and F_M above zero controls the saturation of normalization. This is useful, for example, if the brightness of pixels near the border dominate because these are convolved less because we can't integrate outside of the dataset.

Occasionally, often depending on the contrast and dynamic range and graininess of the noise texture, jagged or pixelated patterns may emerge in the LIC. These can be reduced by enabling the optional anti-aliasing pass.

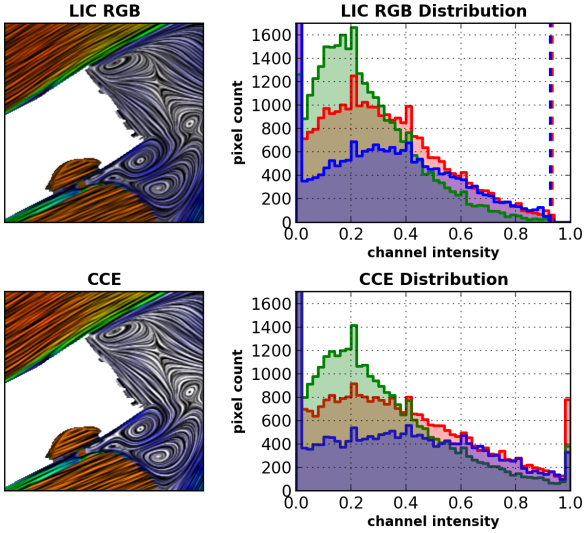


Figure 3: CE Curves

The color contrast enhancement stage is implemented with histogram stretching on the fragments lightness in the HSL color space.

$$L_{ij} = \frac{L_{ij} - m}{M - m} \tag{6}$$

where, the indices i, j identify a specific fragment, L is the fragment's lightness in HSL space, m is the lightness to map to 0, M is the lightness to map to 1. m and M take on minimum and maximum lightness over all fragments by default but may be individually adjusted by the following set of equations:

$$m = min(L) + F_m * (max(L) - min(L)) \tag{7}$$

$$M = max(L) - F_M * (max(L) - min(L)) \tag{8}$$

where, L are fragment lightness values and F_m and F_M are the adjustment factors that take on values between 0 and 1. When F_m and F_M are 0 minimum and maximum are lightness values are used. This is the default. Adjusting F_m and F_M above zero provides fine-tuning control over the saturation.

1.5 Integrator Normalization

Normalizing vectors during integration can highlight critical points in the flow. However, it can alter the relationships of flow features making weak features prominent and strong features less so. When normalized(the default) the input vector field is normalized during integration, and each integration occurs over the same arc-length. When not set each integration occurs over an arc length proportional to the field magnitude as is customary in traditional numerical methods. See, "Imaging Vector Fields Using Line Integral Convolution" for an example where normalization is used. See, "Image Space Based Visualization of Unsteady Flow on Surfaces" for an example of where no normalization is used. Both approaches are valid and useful, the new feature allows for the selection of one or the other.

2 Fragment Masking

Currently the criteria for masking fragments is that all vector components must be identically zero. This doesn't work for many numerical simulations where stagnant flow is not identically zero due to numerical rounding. For example, stagnate flow might be where $|V| < 1e^{-6}$. Also, the lack of control over the color characteristics of the masked fragments result in masked fragments looking drastically different than the LIC'ed fragments. This is illustrated in figure **??**.d where all fragments on the outer walls have been masked , the masked fragments are much brighter than the the LIC'ed fragments.

New fragment masking implementation makes use of a user specified threshold value below which fragments are masked. The masking test may be applied either to the original vectors or the surface projected vectors. By applying the test to the original vectors the masked fragments will match scalar colors when coloring by $|V|$.

Fragments where $|V| < t$ are masked. The fragment masking implementation provides control over the intensity and color of the masked fragments via the following equation:

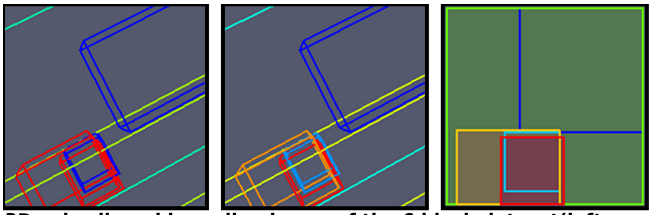
$$c_{ij} = M * I + S_{ij} * (1 - I) \tag{9}$$

where the indices i, j identify a specific fragment, c is final RGB color, M is the RGB mask color, S is the scalar RGB color, and I is the mask color intensity. This allows one control over the masking process so that masked fragments may be: highlighted (by setting a unique mask color and mask intensity ¿ 0), made invisible with and without passing the un-convolved noise texture (by setting mask intensity 0), or made to show the scalar color at a similar level of intensity as the LIC (mask intensity ¿ 0).

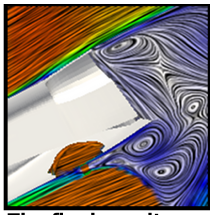
2.1 Optimizations for interactivity

3 Parallelization

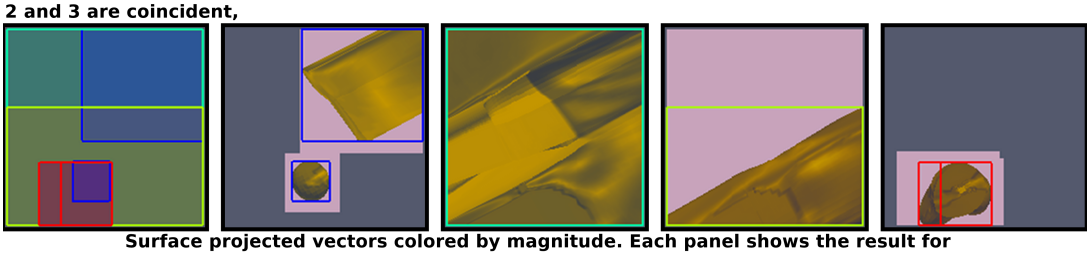
3.1 Compositing Algotithms



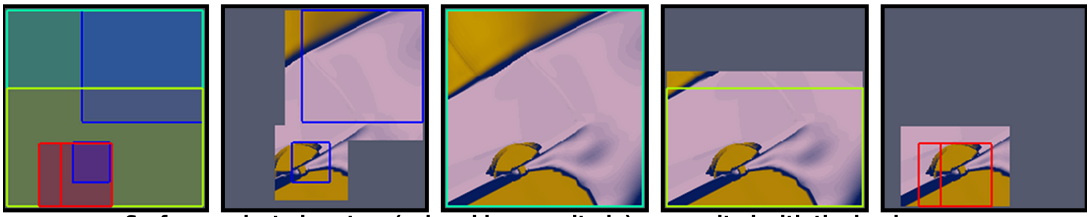
3D axis aligned bounding boxes of the 6 block dataset(left colored by MPI rank and center colored by block id) rendered on 4 MPI ranks, and coresponding 2D screen space pixel extents (right colored by block id). Note that the 2D projections of block 2 and 3 are coincident,



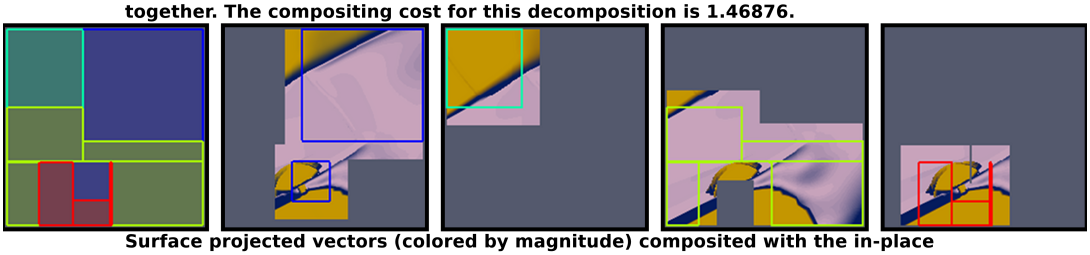
The final result.



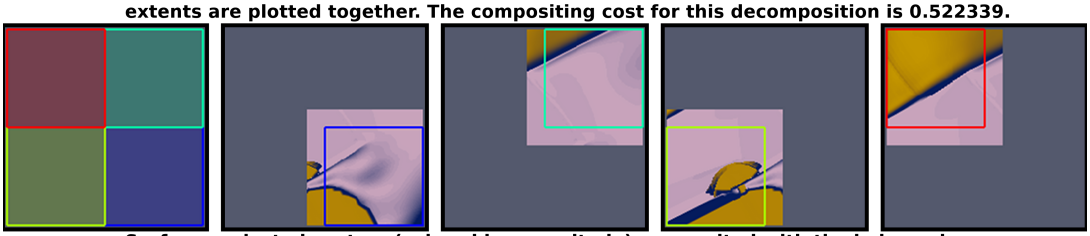
Surface projected vectors colored by magnitude. Each panel shows the result for one of 4 MPI ranks. Tightened pixel extents colored by MPI rank are superposed to show the savings attained by using tightened bounds. On the far left screen space extents for all ranks are plotted.



Surface projected vectors (colored by magnitude) composited with the in-place method. The screen space extents are superposed to show the domain where LIC is computed. The areas outside of the extents are gaurd pixels necessary for correct parallel computation. On the far left screen space extents for all ranks are plotted together. The compositing cost for this decomposition is 1.46876.



Surface projected vectors (colored by magnitude) composited with the in-place disjoint method. The screen space extents are superposed showing the domain where the LIC is computed. The pixels outside of the extents are gaurd pixels necesssary for correct parallel computaiton. On the far left the screen space extents are plotted together. The compositing cost for this decomposition is 0.522339.



Surface projected vectors (colored by magnitude) composited with the balanced method. The screen space extents are superposed showing the domain where the LIC is computed. The pixels outside of the extents are gaurd pixels necesssary for correct parallel computaiton. On the far left the screen space extents are plotted together. The compositing cost for this decomposition is 0.522339.

Figure 4: Compositing algorithms

3.2 Compositor Scaling

Over the past few days I've been analyzing the new parallel surface lic algorithms scalability and performance and I thought you might find the early results helpful as you look at the code. These runs were made with pvpython+pvserver on NERSC's Cray XC30 Edison using Mesa 9.2.0's OS Mesa llvmpipe driver with 4 rendering threads per rank and 8 ranks per node. There's also some documentation of the compositors in this talk (see the last few slides) http://www.hpcvis.com/vis/talks/rdav-ah-2013/rdav-ah-2013.pdf

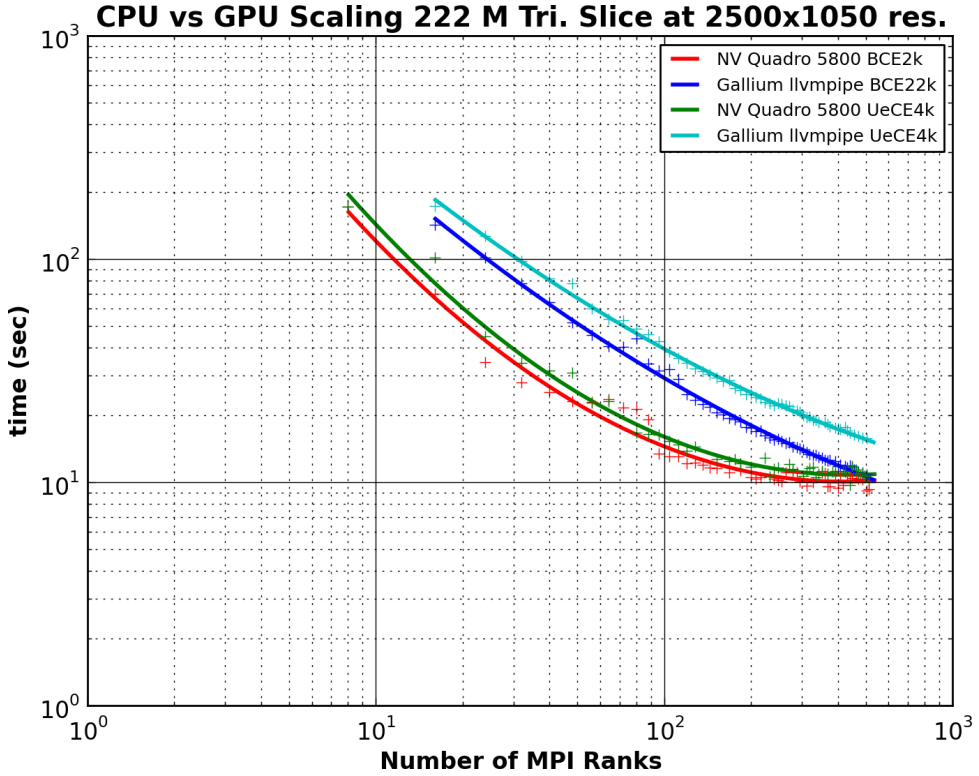


Figure 5: Test of CE scaling on two vector fields

This figure shows scaling from 16 to 512 ranks on a massive 16384x8192 VPIC dataset. The figure shows runs on two vector fields using the settings for a high quality image with and without the new contrast enhancement feature. The figure shows that although the new CE feature makes use of MPI_Allreduce this does not negatively impact scaling. The differences in performance are attributable to the difference in number of integration steps. Typically I've found that to get the look I want I end up using more integration steps when CE is enabled. A sample image from this run is here: http://www.hpcvis.com/vis/vtk-surface-lic-parallelization/kh-new-jaguar-lic-b-woce.png

This figure show scaling from 16 to 512 ranks on a large 1204^3 MHD turbulence dataset and a comparison of two of the compositing algorithms. The INPLACE option is what you and I had initially discussed on Skype, while the INPLACE_DISJOINT option ensures that each fragment is computed only once by uniquely assigning ownership of regions of the screen to ranks while leaving the data in-place ie. a rank doesn't take ownership of a fragments it doesn't have data for. For the INPLACE_DISJOINT approach the compositing communication cost is the same or in some cases less than the INPLACE approach although it's split into two phases(gather and scatter). The figure shows that the INPLACE_DISJOINT

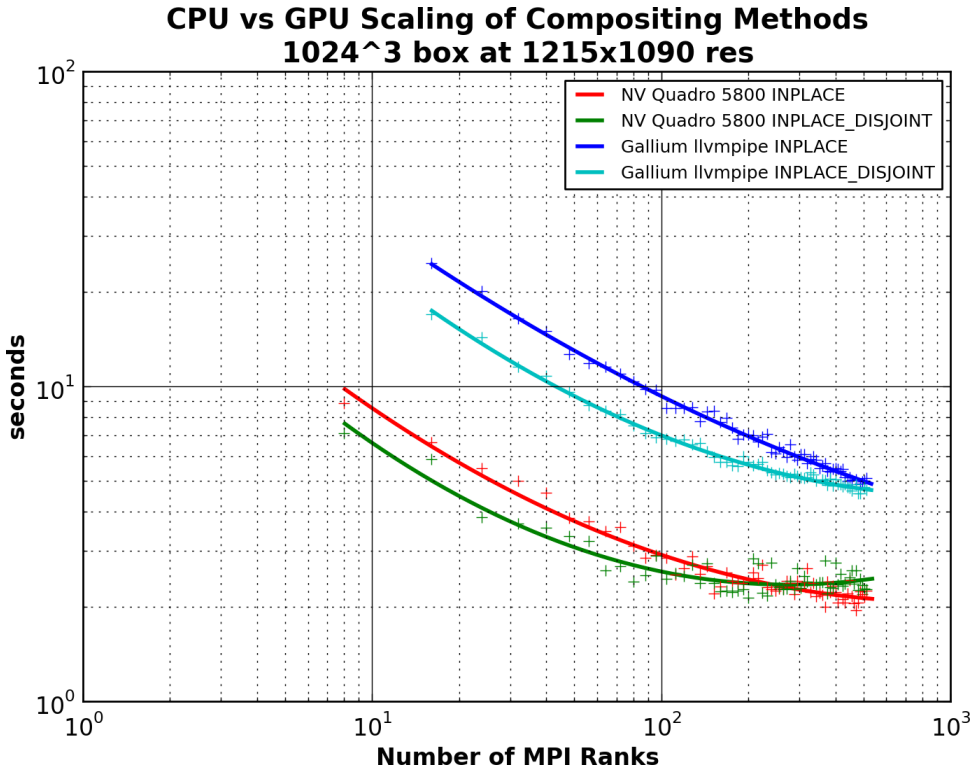


Figure 6: Compositor scaling on MHD turbulence dataset

method offers a nice increase in performance. The speed up is larger in the smaller runs because in those runs the screen regions by rank are larger, and this presents a greater opportunity to reduce the number of fragments processed per rank. The following teo plots show this better. The image produced by these runs is here: http://www.hpcvis.com/vis/vtk-surface-lic-parallelization/pr1-lic.png

These two figures are Gant plots of each rank's activity during the 2 node 16 rank compositing INPLACE and INPLACE_DISJOINT scaling runs. We're looking at the 2-node case because it shows the greatest speed up for the INPLACE_DISJOINT compositor and it will be easier to see the performance differences between the two. For both runs most of the time is spent in RenderGeometry (vector projection and light vertex shaders) and Integrate1/Integrate2 phases. CE phases tend to look long because of the MPI_Allreduce ranks get synchronized by the slowest rank so most of the time spent there is waiting. The RenderGeometry and GatherVectors(compositing and guard pixel exchange) times are nearly identical for both compositors. This means that speed up attained by the INPLACE_DISJOINT compositor is a result of reduced integration time while computing the LIC. With INPLACE_DISJOINT approach each rank has fewer fragments to process and there's a greater opportunity to cull empty fragments as the screen space regions are decomposed. These plots also show that the overhead of decomposing(makiing the screenspace disjoint amongst ranks) is small enough to be negligible and that although the INPLACE_DISJOINT moves less data around during compositing and guard pixel exchange it doesn't contribute a lot to the speed of the approach.

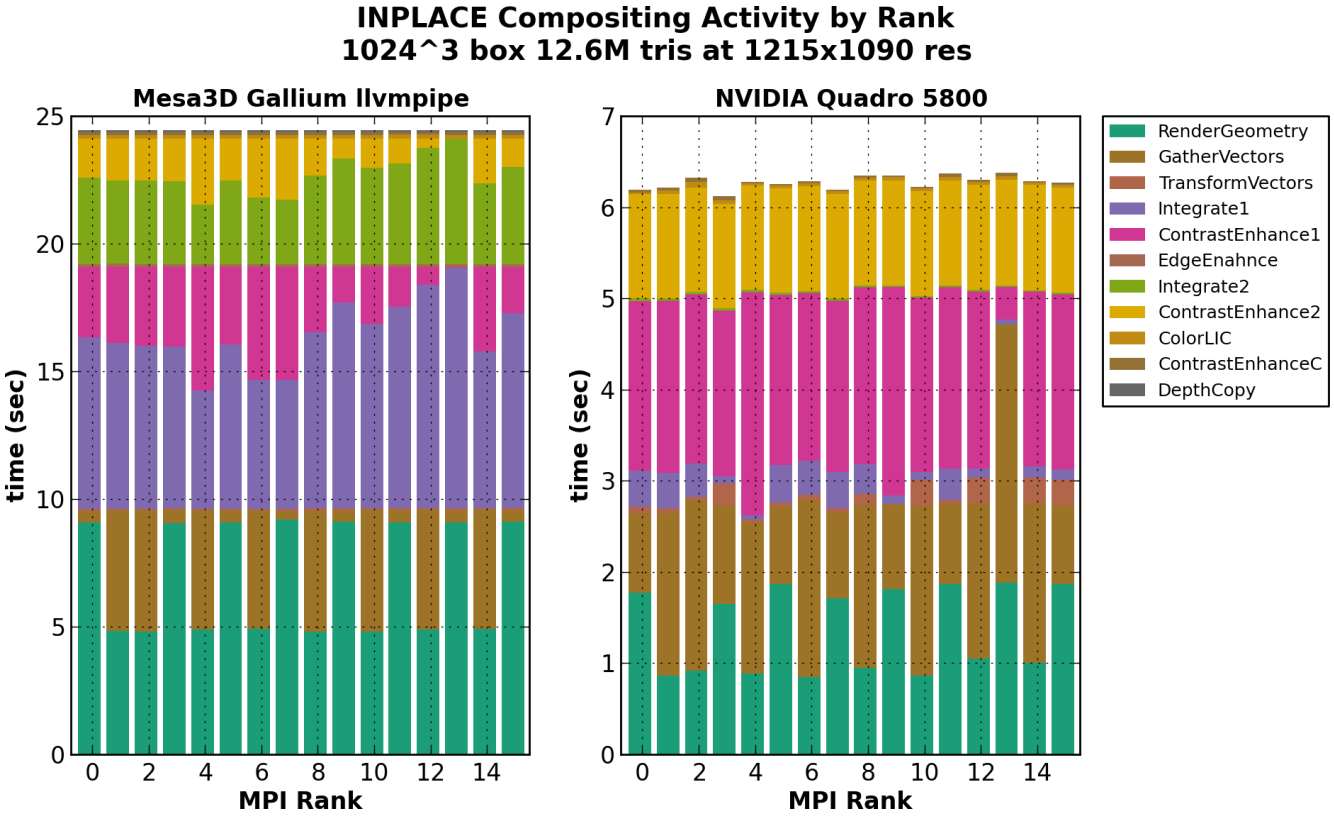


Figure 7: INPLACE Compositing

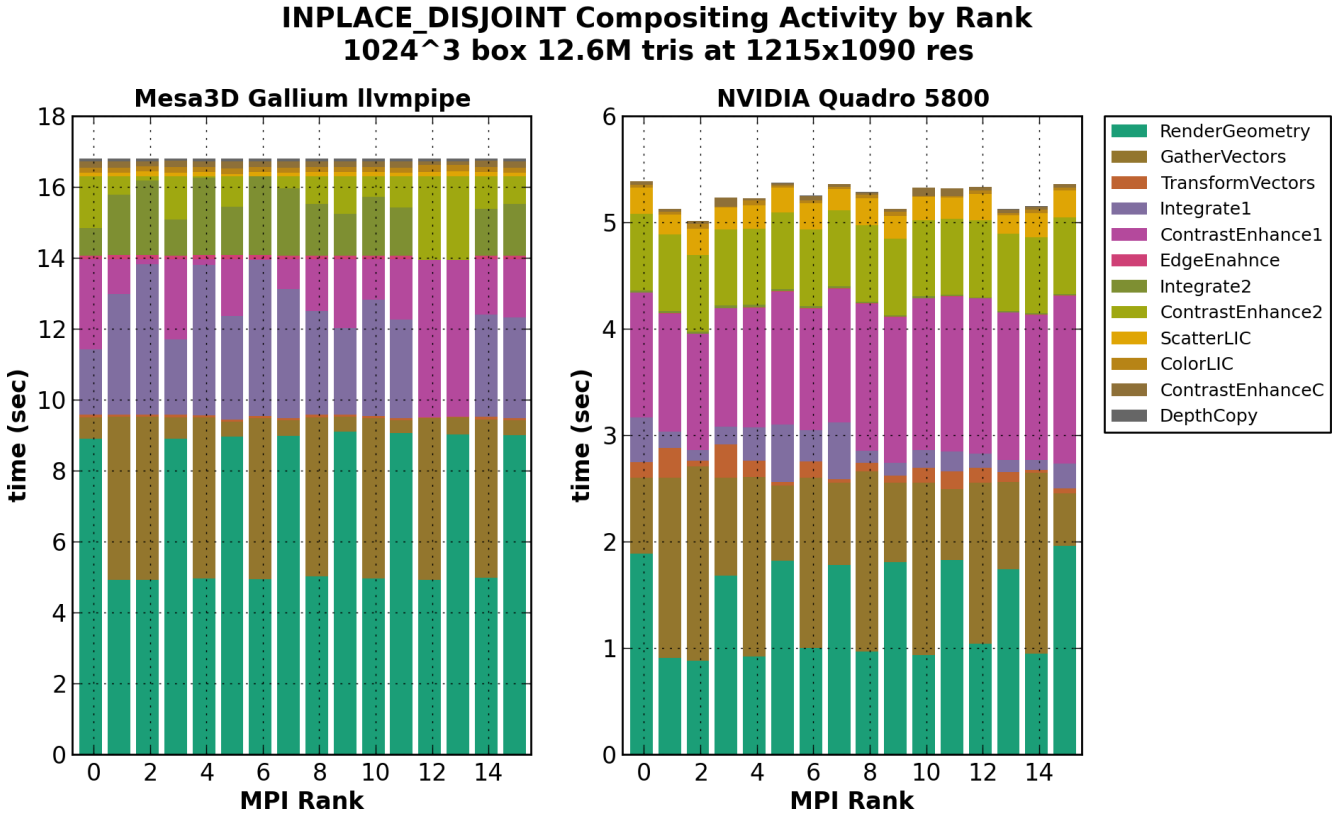


Figure 8: DISJOINT Compositor