PCL and ParaView—Connecting the Dots

Pat Marion, Roland Kwitt, Brad Davis
Kitware Inc.
Chapel Hill, NC, USA

Michael Gschwandtner
Department of Computer Science
University of Salzburg, Austria

Abstract

We introduce a novel open-source framework for analyzing and exploring point cloud datasets and algorithms. This is done by integrating the Point Cloud Library (PCL) within ParaView, a parallel scientific visualization tool. In particular, we demonstrate that by wrapping PCL algorithms as VTK[1] filters, we can leverage PCL's functionality in an interactive, easy-to-use manner within ParaView. The proposed approach enables rapid algorithm development in a coherent framework without the need to write custom visualization code. We illustrate the advantages of the framework with usage examples such as segmentation, data annotation and Python integration. Additionally, we build upon ParaView's inherent parallelization capabilities and present two strong scaling experiments that demonstrate near-linear scaling performance gains in a multi-processor setup.

1. Motivation

Over the last decade, a variety of sensors have emerged that allow 3-D sensing of the environment, even at moderate cost (*e.g.*, Microsoft Kinect). Many of these sensors produce enormous amounts of data which leads to algorithmic as well as computational challenges. In the DARPA Urban Challenge for instance, many teams used 3-D LIDAR sensors like the Velodyne HDL-64E to generate, in real-time, massive 3-D point clouds of the environment which are then input to obstacle detection, or SLAM algorithms. The *Point Cloud Library (PCL)* [7] addresses many of the challenges that arise in processing such data amounts by providing an extensive set of state-of-the-art algorithms for registration, recognition, or feature extraction, just to mention a few. However, an essential part of developing algorithms for point cloud data processing is a powerful, versatile, and interactive visualization tool. This not only allows for exploration of the data in the first place, but also facilitates interactive algorithm exploration—studying parameter spaces and creating, reorganizing, and assessing processing pipelines.

While PCL provides examples of basic point cloud viewers, we are still limited to a static analysis of the data, unless we want to develop custom, interactive, visualizations. While static visual analysis of an algorithm's output is valuable on its own, the ability to interactively modify algorithm parameters and even rearrange the processing pipeline is potentially beneficial, especially in the context of rapid prototyping. While this functionality could be implemented in a custom solution, we argue that time spent writing visualization routines is probably better spent advancing the algorithm part. In addition to that, the re-usability of custom visualization code is usually low. Many solutions tend not to be generic enough to be used in another project. As a consequence, this inevitably leads to re-implementations and even more time spent on non-algorithmic problems.

To address these issues, we propose combining the algorithmic power of PCL with ParaView [1, 9], a well-established scientific visualization tool that is designed to 1) operate on massive data sets [4] and 2) can be run as a parallel visualization solution (*cf*. [2]). While several commercial and non-commercial visualization solutions exist (*e.g.*, *LViz*, *MARS*, *CloudCompare*, or *MeshLab*), the conceptual similarity to PCL renders ParaView a natural choice that allows a straightforward and seamless integration.

ParaView is an open-source, multi-platform data analysis and visualization framework. The *Visualization Toolkit (VTK)* [8] provides the data model and filtering pipeline at ParaView's core. Users can quickly build visualizations to analyze their data using qualitative and quantitative techniques. ParaView was developed to analyze extremely large datasets using distributed memory computing resources, and is runnable on systems ranging from single-core laptops to supercomputers [4].

Among the features that make ParaView a desirable platform for PCL integration are its ability to manipulate different data types together in the same scene, *e.g.*, 2-D and 3-D image data, structured and unstructured grids, meshes, and point clouds, as well as tabular data and graphs. Advanced rendering capabilities such as volume rendering are

[1] Visualization Toolkit (VTK) [8]

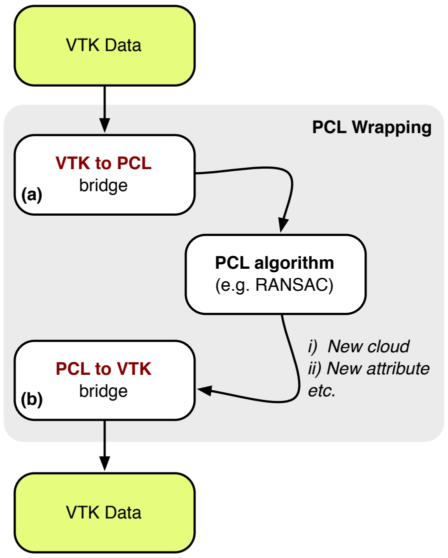


Figure 1. Wrapping of PCL algorithms as VTK filters. Depending on the type of output data of the PCL algorithm, we create new point clouds, add attributes (*e.g.*, outlier field), etc.

available along with a flexible color map and transfer function editor. ParaView provides a 3-D render view, 2-D chart and plot views, and a spreadsheet view. ParaView has support for time varying datasets, and includes filters capable of handling time series data. The animation framework further provides a simple means of producing movies with camera sweeps and varying filter parameters.

ParaView's Python interface ties together these features; filters, views, and I/O routines are fully scriptable. The Python integration includes support for numpy, allowing for rapid prototyping with efficient array computations.

2. Combining PCL with ParaView

The central workflow in ParaView is the filter pipeline. Data flow starts with a source filter, commonly a reader that loads data from the filesystem. Next, a data processing pipeline is constructed by chaining together filters. The output and intermediate stages of the pipeline can be visualized in 3-D render views, displayed in tabular form, or plotted in a chart view (*e.g.*, as a histogram). By combining PCL with ParaView, a pipeline can execute filters from both PCL and VTK together.

PCL algorithms are made available to ParaView through a PCL plugin for ParaView. We have developed a method of wrapping PCL algorithms within VTK filters which are then accessible in ParaView via a plugin. The plugin currently provides several PCL filters from the standard PCL distribution, and establishes a process whereby additional PCL filters can also be wrapped. In addition, this allows developers of new PCL algorithms to wrap their own work for use within ParaView and provides a convenient means for sharing their work.

Data is passed between VTK filters using the VTK data object. A VTK data object is a container that holds vertices, cells, and attributes. When a VTK filter wraps a PCL filter, it internally converts VTK's input point cloud to the PCL data type and then executes the PCL filter, as depicted in Fig. 1(a). The default PCL point cloud data type (using four-component SSE memory alignment) is not compatible with VTK's three-component point data type, so the conversion requires a temporary copy of the data to be resident in memory while the PCL algorithm runs. *Zero-copy* conversions would be possible using custom PCL point types.

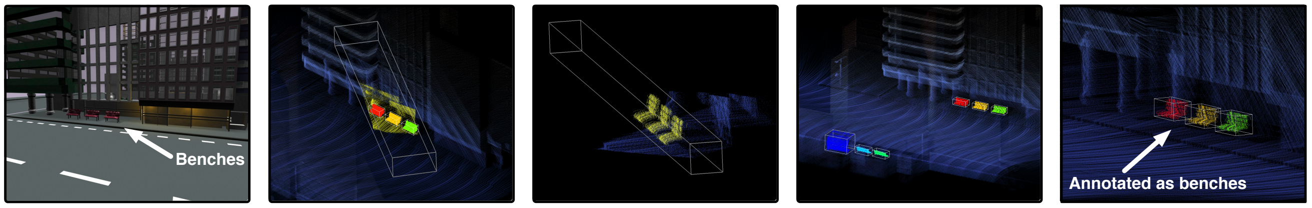
Upon completion of the PCL algorithm, the output is converted to a suitable VTK type, see Fig. 1(b). The conversion strategy of the PCL filter output depends on the nature of the algorithm. For example, the *Voxel Grid* filter outputs a new point cloud, while the *Normal Estimation* filter outputs a vector at each point of the input point cloud. The result of the *Euclidean Clustering* filter is a set of point index arrays, one array per cluster.

In the first case, when a new point cloud is produced, the VTK filter wrapper converts the new PCL point cloud into a VTK point dataset and returns the result. In the second and third cases, the VTK filter wrapper passes through the input point cloud and appends the resulting attribute arrays. In the case of normal estimation, the new attribute array will be a three-component array consisting of the (x, y, z) normal for each point in the cloud. For Euclidean clustering, an integer array is used to indicate point-wise cluster id.

VTK's data object has the ability to add a growing number of attribute arrays, consisting of any primitive data type and number of components. ParaView can visualize the point cloud attributes with colors mapped through a lookup table, or by rendering points with color values supplied directly from an RGB attribute array. Additionally, new attribute arrays can be created by combining attributes arrays with the *Calculator* filter.

When wrapping PCL filters such as the RANSAC plane fit algorithm, rather than extracting the inlier points to produce a new point cloud, we choose to pass through the entire input point cloud and add a new attribute array labeling points as model inliers or outliers. This way, the filter result can be visualized with unique colors mapped to the inlier and outlier points, and the job of extracting the inlier or outlier points can be deferred to another filter such as ParaView's *Threshold Points* filter.

The PCL plugin also includes reader and writer filters that provide I/O capabilities for PCL's native PCD file format. The reader is capable of loading ASCII and binary PCD files containing (x, y, z) coordinates, intensities, RGB colors, and normals, and likewise the writer produces PCD files containing point clouds with similar attributes.



(a) Park benches (b) Frustum selection (c) Refined selection (d) Clustered & Boxed (e) Annotated

Figure 2. Illustration of the annotation example pipeline: (b)-(c) Iterative frustum refinement to isolate points; (d) Euclidean clustering and bounding box extraction; (e) Object annotation/labeling.

3. Usage Examples

In this section, we present three usage examples that illustrate the advantages of combining PCL with ParView for 1) point cloud annotation, 2) rapid prototyping of an object segmentation pipeline and 3) parallelization of computationally intensive tasks.

3.1. Data material

We use two different types of point cloud datasets in this section. The first dataset is a *table scene*, see Fig. 3(a), captured using PCL's OpenNI grabber with Microsoft's Kinect camera. The point cloud contains (x, y, z) plus RGB data. The second dataset is a point cloud of a *city scene*, see Fig. 2(a), generated by *simulating* a Velodyne HDL-64E S2 LIDAR scanner. The sensor is mounted on a car driving through an artificial city (created using Blender[2]). The simulation is done in BlenSor [5][3], an open source project that simulates many of the physical characteristics (including typical sensor noise) of different LIDAR scanners. While the *table scene* contains $\approx$ 240000 data points, the *city scene* contains $\approx$ 10.8 million data points.

3.2. Point cloud annotation

In our first use-case, we demonstrate ParaView as an interactive data exploration tool by using it for extracting and annotating objects in point clouds.

A common challenge when developing segmentation or recognition algorithms for point cloud data is the lack of a-priori available ground-truth data that is required for training discriminant classifiers, or the quantitative evaluation of algorithm performance. In our example, the objective is to manually extract bounding boxes from the park benches shown in Fig. 2(a). This data could later be used to train or evaluate the performance of a *park bench detector*.

A pragmatic, yet time-consuming, approach could be to use a programmatic concatenation of several pass-thru filters, which subsequently *isolate* the object(s) of interest. Finding the right settings for each coordinate axis, however, can be quite challenging. For example, the park benches in Fig. 2(a) are 1) directly located under the edge of a large building and 2) in vicinity to some pillars; adjusting the coordinate axis parameters of pass-thru filters is tricky in that case.

It is substantially simpler to create these pass-thru filters using ParaView's interactive frustum selection, thus incrementally extracting the points corresponding to the park benches. This is done by finding a suitable view and selecting a rectangular region around the area of interest. Repeating that process multiple times allows us to isolate the points corresponding to the park benches easily with a small number of mouse clicks.

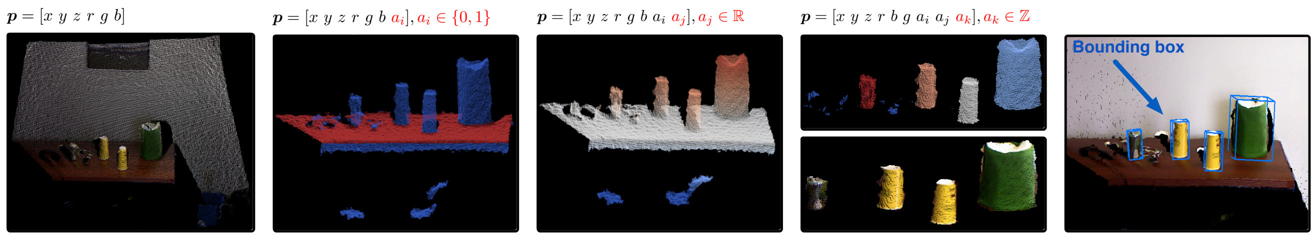
The resulting manual selections can be used as input to ParaView processing pipelines. For example, in Figs. 2(c)-2(e) we show how to individually annotate and measure the bench dimensions. With the bench points isolated, we run the wrapped PCL *Euclidean Clustering* filter to augment the points with a scalar representing the cluster groupings. Next, we fit oriented bounding boxes to the individual bench clusters using ParaView's *Oriented Bounding Box* filter.

We note that the process of iteratively isolating points can be used for data exploration as well. For example, ParaView facilitates interactive computation of point attribute histograms, which can be particularly useful when studying discriminative features for object recognition. Further, data can be accessed in spreadsheet views, *e.g.*, to extract GPS information, or exported in CSV file format for further processing by another tool (*e.g.*, MATLAB).

3.3. Object segmentation

Segmentation of point clouds has recently gained considerable research interest (*e.g.*, [6]). As our second usage example, we present a simple segmentation problem, based on the *table scene*. This example illustrates how a typical segmentation workflow can be designed and explored interactively in ParaView. Given the *table scene* shown in Fig. 3(a), the objective is to 1) segment all objects on the table, 2) measure the height of each object point relative to the table surface and 3) add that information as an attribute array to the point cloud. We use PCL's RANSAC plane fitting

[2]http://www.blender.org
[3]http://www.blensor.org


$\boldsymbol{p} = [x\ y\ z\ r\ g\ b]$
$\boldsymbol{p} = [x\ y\ z\ r\ g\ b\ a_i], a_i \in \{0,1\}$
$\boldsymbol{p} = [x\ y\ z\ r\ g\ b\ a_i\ a_j], a_j \in \mathbb{R}$
$\boldsymbol{p} = [x\ y\ z\ r\ b\ g\ a_i\ a_j\ a_k], a_k \in \mathbb{Z}$
Bounding box

(a) Kinect data (b) RANSAC plane fitting (c) Height from plane (d) Clustered (e) Boxed

Figure 3. Illustration of the pipeline and corresponding outputs of the segmentation example of Sect. 3.3. The figure further shows that points $\boldsymbol{p}$ are augmented by attributes a_i, a_j and a_k as we move forward in the pipeline (without having to duplicate the clouds).

implementation as well as PCL's Euclidean clustering, both wrapped as VTK filters, as the algorithmic core of the example.

Our first step is to remove all the points that lie on the wall behind the table. We can use the RANSAC segmentation algorithm with a plane model to find inlier points. The algorithm, wrapped as the *Fit Plane* filter, passes through its input point cloud and adds a new (binary) attribute $a_i \in \{0, 1\}$ to each point where a value of '1' signifies a model inlier and '0' signifies an outlier. The point cloud can be visualized in ParaView with colors assigned to points based on the value a_i. The *distance-to-model* threshold parameter for RANSAC fitting can be conveniently adjusted in the property panel of ParaView. Through visual inspection, we can determine that a distance-to-model threshold of 10 centimeters will robustly label all of the wall points as inliers, including points that lie on the surface of a picture frame that juts out from the wall by several centimeters. Tuning this parameter would be less efficient without an interactive filter execution environment.

To actually remove the wall from the point cloud, we apply ParaView's *Threshold Points* filter. This filter is a pass-thru filter that takes a point attribute and a threshold range, similar to PCL's own pass-thru filter. We use this filter to extract all points satisfying $a_i = 0$, *i.e.* the outlier points that do not belong to the plane model representing the wall.

The next most prominent planar surface is the tabletop, thus another application of the *Fit Plane* filter results in segmentation of the table points. This time, we'll use ParaView's *Programmable Filter* to implement an extended version of the filter. Technically, the *Programmable Filter* takes a Python code snippet and executes it. The Python code has access to the input data object, and can access the points and point attributes as numpy arrays. The VTK data array is compatible with numpy arrays so that the data may be accessed directly without any additional copying step. In Python snippet 1, we have implemented a *Programmable Filter* that executes 1) the RANSAC plane fitting, then 2) queries the filter for the origin and normal of the best fitting plane model and 3) computes the signed distance to the plane for each point in the input cloud using numpy array routines. Eventually, the input point cloud is passed through to the output using VTK's shallow copy semantics and the signed distance-to-plane scalar $a_j \in \mathbb{R}$ is added to each point as a new point attribute. ParaView renders the point cloud colored by that attribute, as shown in Fig. 3(c).

Python example 1 Distance to RANSAC plane model

```python
# Run PCL RANSAC using plane model with 1 cm tolerance
f = vtk.vtkPCLFitPlane()
f.SetDistanceThreshold(0.01)
f.SetInput(input)
f.Update()
origin = f.GetPlaneOrigin()
normal = f.GetPlaneNormal()

# Compute signed distance to the plane
dist = numpy.dot(input.points - origin, normal)

# Flip the sign if needed (dot normal with Y axis)
if numpy.dot(normal, [0,1,0]) > 0:
    dist *= -1.0

# Pass thru the RANSAC filter output and append
# the distance to plane attribute to the points
output.ShallowCopy(f.GetOutput())
output.pointdata.append(dist, 'dist_to_plane')
```

Applying the *Threshold Points* filter based on the signed distance-to-plane attribute a_j allows for extraction of points within a certain distance range, *i.e.* $a_j \in [0.01\ 0.3]$, relative to the table surface.

Since objects are now well isolated, we execute PCL's Euclidean clustering algorithm, wrapped as a VTK filter, to augment the points by a cluster identifier attribute $a_k \in \{0, \ldots, N\}$, where 0 is assigned to points that do not meet the clustering criteria, and values 1 through N identify points belonging to the different clusters. The output of the clustering step is visualized in ParaView with points colored by their cluster index as shown in Fig. 3(d). In a final step, we exploit the *Oriented Bounding Box* filter to compute an oriented bounding box for each distinct cluster.

Python snippet 2 shows how the entire processing pipeline described in this section can be applied to a point cloud in a batch-processing manner.

Python example 2 Reusability

```
# Fit wall points and remove them
PCLFitPlane(DistanceThreshold=0.1)
ThresholdPoints(Scalars='ransac_index',
                ThresholdRange=[0,0])

# Fit plane to table and measure point height
ProgrammableFilter(Script=readFile(
                  'dist_to_plane_filter.py'))

# Extract points within 30 cm of the tabletop
ThresholdPoints(Scalars='dist_to_plane',
                ThresholdRange=[0.01, 0.3])

# Cluster remaining points
PCLEuclideanCluster(ClusterTolerance=0.3, MinSize=400)
```

3.4. Parallelization

In our final usage example, we demonstrate how we can leverage ParaView's parallelization capabilities to reduce execution time of computationally intensive filters. ParaView is natively a parallel visualization tool. Though it may be used as a serial application, it is designed as a client/server application where the serial client connects to an MPI enabled, multi-process server. In the client/server configuration, all readers and filters are instantiated on the server. A complete instance of the filter pipeline exists on each server process, and each process executes the pipeline on a piece of the whole dataset. ParaView supports parallel rendering, where the composited image is delivered to the client, or client side rendering, where the geometry is collected and delivered to the client.

We note that certain PCL algorithms include support for OpenMP vectorization that scales up to the number of processors on a *single* machine. In contrast, ParaView's process-parallel approach, using MPI, has been shown to be effective from multi-core single machines up to supercomputer scales. This also has the advantage that existing serial algorithms can potentially benefit from parallelization without being refactored. Both paralelliatzion strategies have benefits and even a hybrid approach is sometimes warranted.

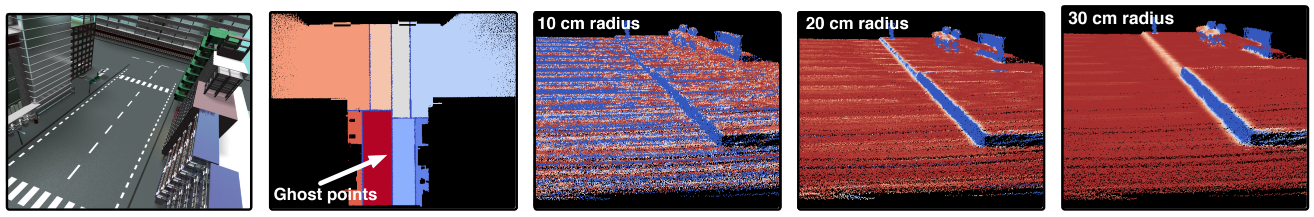
To illustrate parallelization, we chose PCL's surface normal estimation algorithm as an example. In our case, the algorithm is wrapped as a VTK *Surface Normal Filter* that computes a normal estimate at each point of the whole *city scene*, part of which is shown in Fig. 4(a). For visualization, we compute (using numpy) the angle between the surface normals and the z-axis and color the points by the value of that attribute. Fig. 4(c) shows a zoomed-in view of the curbs next to the street, for different normal estimation neighborhood radii, demonstrating that the added attribute is effective for visualizing surface planarity.

To facilitate parallelization of algorithms that require neighborhood information, ParaView supports the notion of *ghost points*. Ghost points are shared, or duplicated points located at partition boundaries. They are owned by one processes and duplicated on one or more other processes. Without ghost points, algorithms such as outlier removal or normal estimation, would inevitably yield incorrect results for points at partition boundaries. The *pv-meshless* plugin[4] for ParaView provides a point cloud partitioning filter that uses the Zoltan library [3] to perform data redistribution and balancing, and also performs ghost point generation with a specified boundary radius. Fig. 4(b) shows the *city scene*, colored by process IDs using the partitioning obtained from *pv-meshless*, as well as the generated ghost points.

As a reference for the parallelization experiments, non-parallel execution of surface normal estimation with a 10 centimeter neighborhood search radius takes $\approx$ 190 seconds on our Intel Xeon Dual Quad-Core test system, running Linux 2.6.32. We partitioned the point cloud with a 10 centimeter ghost point boundary which is the minimum overlap that we need to have to avoid boundary issues with a 10 centimeter search radius. Fig. 5 shows the execution time of the *Surface Normal Filter* as a function of the number of processors on a single machine. Execution times are averaged over ten runs of the experiment. As we can see, performance gains are close to what we expect in the optimal case with a speedup of $\approx 6\times$ using all eight processors. Fully tracking down the source of the overhead that leads to the gap to optimal scaling is tricky and can have multiple roots: Among those, we highlight that as the number of partitions increases, the shared ghost points increase the total overall number of points that are filtered. Using eight processors leads to $\approx 10\%$ increase in the total number of points for instance. In addition, the number of ghost points depends on the point density along the partition boundaries which can differ significantly in case of LIDAR data. Apart from that, Fig. 5 does not include I/O time. While parallel I/O is generally possible within ParaView, we did not explore that direction in this example. However, once data is partitioned (*e.g.*, using *pv-meshless*), it can be saved in a format compatible with one of ParaView's existing parallel I/O readers.

For comparison, Fig. 5 additionally shows a similar execution time vs. number-of-processors plot for running VTK's *Extract Geometry Filter* on the point cloud. The filter is configured to extract all points in the *city scene* that lie within any of the six bench bounding boxes, see Fig. 2(d), that were produced in the cloud annotation example of Sect. 3.2. The algorithm implemented by the *Extract Geometry* filter works on a single point at a time and does *not* require point neighborhood information. Thus it qualifies for immediate parallelization, without any ghost point requirements. In fact, we ran the same partitioning, except setting the boundary width to zero. From Fig. 5, we see

[4]https://hpcforge.org/projects/pv-meshless



(a) City scene (b) Partitioning (c) Varying search radius for normal estimation

Figure 4. Illustration of (b) Point cloud partitioning for eight processors with ghost points at partition boundaries; (c) Zoomed-in view of the street curbs with points colored by the angle between the surface normals and the z-axis at various normal estimation search radii.

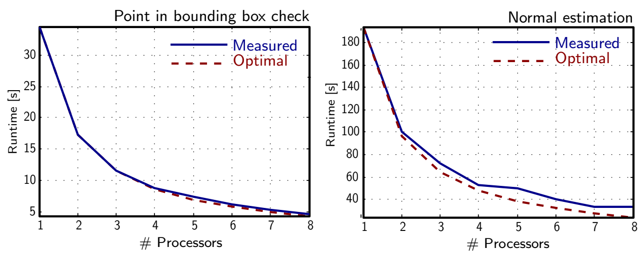


Figure 5. Execution time of strong scaling experiments demonstrating 1) point inside bounding box extraction and 2) surface normal computation.

that the measured execution time is even closer to the optimal case for this experiment which can be explained, to a large part, by the missing ghost points.

While many PCL algorithms produce valid results using ghost points, others require more sophisticated implementations to work in parallel. An example of that is Euclidean clustering: while cluster results are correct within each process piece, clusters spanning partition boundaries need to be joined using inter-process communication. Although, we did not implement an example of that, ParaView provides high-level communication routines built upon MPI that are available and make such implementations more straightforward.

4. Discussion

In this work, we have shown that the combination of ParaView and PCL leads to a unified framework for interactive data exploration, visualization as well as parallel data processing. The close integration with Python and numpy further facilitates rapid prototyping, while maintaining computational performance. We believe that this combination provides a natural integration of the strengths of ParaView with the strengths of PCL and lowers the barrier for the development of novel point cloud processing algorithms.

While the proposed approach is effective in targeting the issues we mentioned in the introduction, it is not designed for real-time processing of point clouds.

We further highlight that in addition to ParaView's filtering pipeline, which we exploited in this work, ParaView provides a framework for developing experimental rendering algorithms using GPU shaders. On our test system, we have successfully visualized point clouds of up to 20 million points with interactive frame rates, however, visualizing even larger clouds might require the use of GPU shaders at some point.

Finally, we note that the PCL plugin for ParaView, implementing the filters presented in this work and others, is available to the community in binary and source code form, to be downloaded at: `http://www.paraview.org/Wiki/ParaView/PCL_Plugin`

References

[1] J. Ahrens, B. Geveci, and C. Law. ParaView: An end-user tool for large data visualization. In C. Hansen and C. Johnson, editors, *The Visualization Handbook*, pages 717–732. Elsevier Academic Press, 2005. 1

[2] A. Cedilnik, B. Geveci, K. Moreland, J. Ahrens, and J. Favre. Remote large data visualization in the Paraview framework. In *EGPGV*, 2006. 1

[3] K. Devine, E. Boman, E. Heaphy, B. Hendrickson, and C. Vaughn. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002. 5

[4] N. Fabian, K. Moreland, D. Thompson, A. Bauer, P. Marion, B. Geveci, M. Rasquin, and K. Jansen. The paraview coprocessing library: A scalable, general purpose in situ visualization library. In *LDAV*, 2011. 1

[5] M. Gschwandtner, R. Kwitt, and A. Uhl. Blensor: Blender sensor simulation toolbox. In *ISVC*, 2011. 3

[6] F. Moosmann, O. Pink, and C. Stiller. Segmentation of 3D Lidar data in non-flat urban environments using a local convexity criterion. In *IV*, 2008. 3

[7] R. B. Rusu and S. Cousins. 3D is here: Point Cloud Library (PCL). In *ICRA*, 2011. 1

[8] W. Schroeder, K. Martin, and B. Lorensen. *Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Kitware Inc., 4th edition, 2008. 1

[9] A. Squillacote. *The Paraview Guide*. Kitware Inc., 3rd edition, 2008. 1