Scalable processing of massive geodata in the cloud: generating a level-of-detail structure optimized for web visualization

Michel Krämer[1,2], Ralf Gutbell[1], Hendrik M. Würz[1,2], and Jannis Weil[2]

[1] Fraunhofer Institute for Computer Graphics Research IGD, Darmstadt, Germany
[2] Technical University of Darmstadt, Germany

Abstract. We present a cloud-based approach to transform arbitrarily large terrain data to a hierarchical level-of-detail structure that is optimized for web visualization. Our approach is based on a divide-and-conquer strategy. The input data is split into tiles that are distributed to individual workers in the cloud. These workers apply a Delaunay triangulation with a maximum number of points and a maximum geometric error. They merge the results and triangulate them again to generate less detailed tiles. The process repeats until a hierarchical tree of different levels of detail has been created. This tree can be used to stream the data to the web browser. We have implemented this approach in the frameworks Apache Spark and GeoTrellis. Our paper includes an evaluation of our approach and the implementation. We focus on scalability and runtime but also investigate bottlenecks, possible reasons for them, as well as options for mitigation. The results of our evaluation show that our approach and implementation are scalable and that we are able to process massive terrain data.

Keywords: Distributed systems, Algorithms, Cloud computing, Geographic Information

1 Introduction

Terrain data is becoming more and more important for applications such as land monitoring, environmental management, hydrological modeling, or even tourism and city marketing. Earth observation satellites collect up to several TB of terrain data per day [20,33]. Public and industrial stakeholders have great interest in visualizing and comparing up-to-date datasets for various purposes.

Due to the increasing availability and resolution, processing and visualizing terrain data has, however, become a challenging task that requires scalable systems and algorithms. Modern cloud infrastructures offer virtually unlimited compute power, and 3D visualizations in the web enable large datasets to be shared among different collaborating parties through an accessible and lightweight medium.

The most common approach to create web-based visualizations of 3D terrain data is to generate image pyramids representing multiple levels of detail [5]. In this paper, we present an approach to preprocess terrain data in the cloud and to transform it into a hierarchical level-of-detail structure consisting of triangulated meshes. In particular, we show that we can apply a divide-and-conquer strategy to very large datasets in order to parallelize and distribute the processing. We perform a Delaunay triangulation [11,8] to individual tiles with a specified maximum number of points as well as a maximum geometric error. We implement the data processing with Apache Spark [2] and GeoTrellis [4], two frameworks for the distributed processing of large datasets. GeoTrellis is specifically designed for georeferenced raster data and contains many useful spatial operations. We visualize the results in Cesium [7], a web-based platform for creating virtual globes in 3D. Based on this, we evaluate scalability and performance of our approach and discuss possible bottlenecks.

The remainder of this paper is structured as follows. We first discuss related work (Section 2). We then present our approach (Section 3) and the implementation (Section 4). After this, we perform an evaluation and discuss the results (Section 5). We finish the paper with conclusions and directions for future work (Section 6).

2 Related work

Processing geospatial data in the cloud has become more and more important to the research community in the last decade. This section is divided into three subsections that present approaches with regard to general cloud processing (Section 2.1), distributed geo processing algorithms in the cloud (Section 2.2), and attempts to parallelize the Delaunay triangulation (Section 2.3).

2.1 General cloud processing approaches

General purpose architectures have been developed that are able to deploy different processing algorithms. Two major approaches are batch and stream processing.

Batch processing works well for existing datasets that have been acquired at a certain time and should later be transformed. Scientific workflow management systems such as Pegasus [10] or Kepler [25] support this style of processing. The same applies to the architecture presented in our earlier work [18,17] or the MapReduce programming paradigm [9].

For a constant stream of incoming data, *stream processing* was developed. In *stream processing*, as implemented by Apache Spark Streaming [3] or Storm [1], data is processed immediately while it is being acquired. The result dataset is updated incrementally. As a general downside, *stream processing* introduces some overhead. To mitigate this, novel concepts such as *micro-batching*, the *Lambda architecture* [26], or the *Kappa architecture* [19] have emerged.

Isenburg et al. have shown the high potential of streaming for the triangulation of spatially large areas with respect to memory and time consumption [16]. In contrast to our approach, their method lacks the ability to introduce an error bound metric to the triangulation and the ability to parallelize the triangulation processes.

Our approach is a typical batch process expecting the data to be present at the beginning.

2.2 Geo-specific cloud approaches

In addition to the general-purpose architectures mentioned in Section 2.1, there are approaches specialized for the processing of geospatial data. They are strongly coupled to the underlying cloud infrastructure. For example, Qazi et al. describe a software architecture to model domestic wastewater treatment solutions in Ireland [31]. Their solution depends on Amazon Web Services on which they install the commercial tool ArcGIS Server via special Amazon Machine Images (AMIs) provided by Esri. Warren et al. process over a petabyte of data acquired by the US Landsat and MODIS programs over the past 40 years [37]. Their processing pipeline connects 10 steps including uncompressing raw image data, classification of points, cutting tiles, performing coordinate transformations, and storing the results to the Google Cloud Storage. Their static process is highly optimized for the Google platform. Li et al. use the Microsoft Azure infrastructure to process high-volume datasets of satellite imagery [22]. They leverage a cluster of 150 virtual machine instances making the process 90 times faster in comparison to a conventional application on a high-end desktop machine.

In contrast to these works, our approach does not depend on a specific cloud infrastructure. We only require GeoTrellis, which can be installed on arbitrary (virtual) machines, even in a cluster or grid. We think that GeoTrellis (and the underlying Spark framework) are a good choice for the processing of large geospatial data. This is supported by the work of Liu et al. who present an approach to detect changes in large LiDAR point clouds [24]. They summarize that Spark is suitable to process data that exceeds the capacities of typical GIS workstations, which matches our evaluation results with regard to data scalability.

2.3 Parallelized Delaunay triangulation

Other earlier works focus on parallelizing the Delaunay triangulation. Spielman et al. presented a modification of the delaunay triangulation that enables parallel mesh updates [34]. In each iteration, they choose n points and insert them into the mesh in a parallelized manner. Hu et al. go even further and map the triangulation to a GPU-based implementation [15]. For this, they load the model in the video memory and compute updates on a vertex stream in the geometry shaders. With this approach, they are able to generate real-time view-dependent meshes, as long as the models are small enough. Concerning the scalable creation of TINs, Goodrich presents a method to create a convex hull with a Delaunay triangulation performed on bulk-synchronous parallel computers [13]. A more recent approach by Nath et al. guarantees $O(n\, log\, n)$ runtime to create TIN DEMs with a modified Delaunay algorithm tailored for the massive parallel communication model [27].

In contrast, our approach distributes terrain tiles to multiple virtual machine instances in the cloud in order to divide the time-consuming triangulation process to available compute resources. On each instance, the triangulation itself is a non-parallel iterative process.

3 Approach

Our approach is based on a strong mapping between the target data structure and how the processing is distributed in the cloud. We look into how geospatial data is usually organized for web-based visualization and optimize our processing and distribution strategy accordingly.

3.1 Hierarchical level-of-detail structure for terrain

Ulrich has shown that hierarchical Level of Detail (LoD) structures allow arbitrarily sized data to be visualized in the web [36]. Formats such as I3S [12] and 3D Tiles [6] follow this approach and are optimized for 3D objects but not terrain. As mentioned above, we use Cesium (Version 1.64) for the web visualization. This framework supports the Quantized Mesh (QM) format [5], which is a similar data format optimized for terrain. In QM, the globe is divided into quadratic subsections (tiles) with different granularity according to a given zoom level. These tiles are organized in a hierarchical quadtree as specified by the Tile Map Service (TMS) using the *global-geodetic* profile [30]. This tiling scheme is similar to the one used in the Web Map Tiling Service (WMTS) [28].

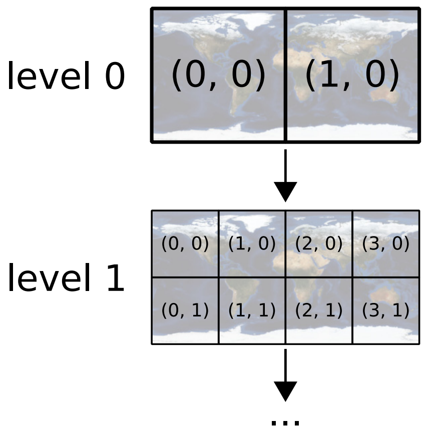

Fig. 1. Layout scheme for zoom levels 0 and 1. With increasing levels, the resolution increases too. Image of the earth by [35].

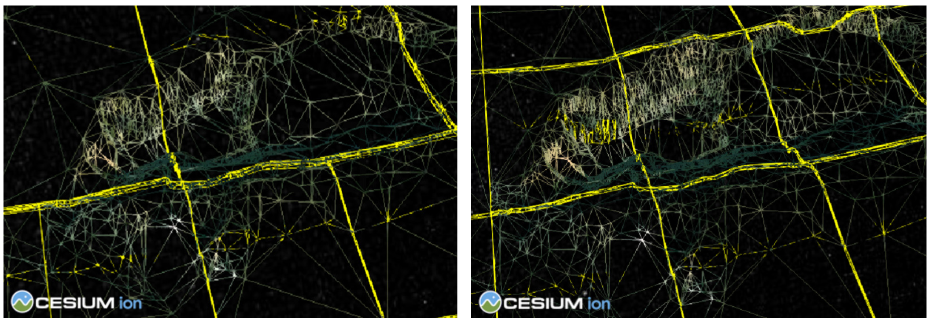

Fig. 2. Depending on the camera position, Cesium displays different levels of detail. In the left picture, a lower resolution is loaded than in the right picture.

QM initially divides the earth into two tiles for zoom level 0. For the next level, each tile from the previous level is divided into four subtiles. In this process, the number of points in the stored mesh increases. In level 0, the error (i.e. the difference) between the provided mesh and the real earth surface is very high, while level 17 provides enough detail to model objects with a size of a few meters. The dividing process of the tiles is repeated for subsequent zoom levels, resulting in a hierarchical subdivision of the surface of the earth (see Figure 1). The QM terrain format additionally specifies how the mesh data for the individual tiles should be saved.

When the globe's model is displayed, Cesium determines which tiles should be loaded based on the current view. This is not restricted to a single zoom level. Instead, Cesium mixes tiles from different levels as depicted in Figure 2.

3.2 Divide-and-conquer strategy for scalable processing

Let us assume we want to generate the terrain meshes for a raster dataset $\mathcal{D}$ on each level within the range from a bottom zoom level $b \in \mathbb{N}$ to a top zoom level $t \in \mathbb{N}$ with $b > t$. In order to achieve scalability, we apply a divide-and-conquer strategy and split the input data into tiles that can be processed individually on separate nodes in the cloud. The tile layout is determined by our target format, which is, as described above, a hierarchical quadtree.

The process is illustrated in Figure 3. First, we need to find out which pixels from $\mathcal{D}$ correspond to which tiles in our output data structure. For this, we split and resample $\mathcal{D}$ into raster tiles according to the bottom level b of our layout scheme. This includes loading and repartitioning the source data (see Section 4.1) as well as applying the layout scheme (Step ①). The repartitioned tiles then have the same extent as the terrain mesh tiles expected by Cesium for the bottom level b and contain the height data as resampled pixel values.

In the next step ②, we convert each of these 2.5D raster tiles to a set of 3D points and triangulate them as described in Section 4.3. This results in mesh tiles containing the height information of their corresponding extent in the

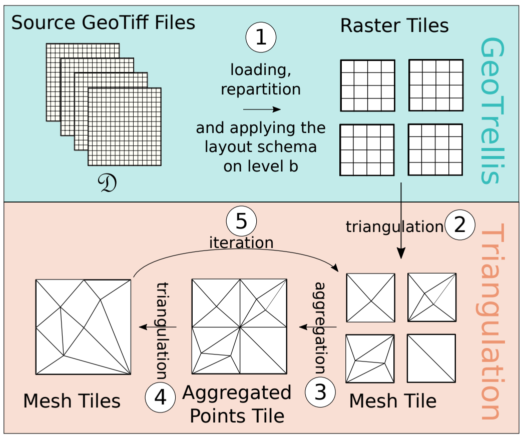

Fig. 3. Basic concept of the conversion process from raster input files to meshes for all required levels of detail.

form of geometry. The following steps of the conversion process get the height information from this mesh instead of the original raster data (bottom row in Figure 3).

When all the mesh tiles for zoom level b are generated, we aggregate this mesh data according to the layout of zoom level $b - 1$ ③. In our case, each tile in level $b - 1$ has four subtiles from level b to be merged together as described in Section 4.4.

To achieve this, we first extract the point data from the meshes. Afterwards, we start a new triangulation process on these points with a higher maximum error ④. This results in a new mesh with lower resolution for each tile of zoom level $b - 1$. This procedure is repeated until every zoom level up to the minimum level t is converted completely ⑤.

4 Implementation

Our main idea is to map the hierarchical conversion problem to data structures compatible with Apache Spark [2] (Version 2.20), which then provides the ability to triangulate the height data in a distributed way and utilize the resources of the cloud infrastructure. For the triangulation, we explain how we handle the raster data with GeoTrellis [4] (Version 1.2.1) and how the merging for less detailed levels can be done.

Apache Spark is a framework for distributed computing. It consists of multiple components with different responsibilities:

– **Master** Manages all available workers with their executors and distributes them on request of a driver.
– **Worker** A server instance consisting of multiple executors.

- **Executor** A working unit using multiple cores. It will perform the necessary calculations for a task.
- **Driver** The program where the tasks are created. The driver asks the master for executors and sends tasks to them. This allows multiple drivers to request calculation capabilities from the same master.

Apache Spark enables us to build a scalable platform for distributed calculations. It hides the network activities and executor allocation, so we can focus on the program logic.

GeoTrellis is a framework for the processing of georeferenced raster data. For this, it provides data types, I/O functionality, and raster mapping operations.

We use GeoTrellis to read information from multiple input files, to manage the underlying coordinate reference system, and to produce normalized tiles. A tile is a rectangle with customizable dimensions covering a specific area in the source data. For example, it is possible to split the input data into n tiles, each of them consisting of $width \times height$ pixels. Based on our layout scheme in Figure 1, we use $width = height = 256$. This means a single tile mesh will be generated based on a raster with 256×256 pixels. GeoTrellis can map, filter, and manipulate these tiles.

GeoTrellis integrates with Apache Spark. This allows us to combine the benefits of a managed distributed computation with the support of geospatial data.

4.1 Loading and repartitioning source raster data

We use GeoTrellis to split the input terrain $\mathcal{D}$ into separate tiles (Figure 3, Step ①). GeoTrellis creates a so-called Resilient Distributed Dataset (RDD) that contains the individual tiles and that can directly be used by Spark. To guarantee that each RDD tile has a size of 256×256 pixels we make use of the resampling methods offered by GeoTrellis—i.e. Nearest Neighbor (NN) and Bilinear sampling (see Section 5.2).

The whole mapping process is executed on a Spark cluster. The terrain meshes generated on the individual Spark instances are saved on the hard drive.

4.2 Calculating the required bottom zoom level

Algorithm 1 determines an appropriate bottom zoom level b for a given input tile resolution. The algorithm calculates the width and height of one source GeoTIFF file as the delta of its top-left and bottom-right WGS84 coordinates assuming that all GeoTIFF files have the same resolution. It divides these values by the file size in pixels and multiplies them by the grid size, which results in the area that is covered by the tile in the final output. The algorithm increases the zoom level incrementally and checks whether it is finer than the required granularity. It returns the first level that is sufficient to cover all data included in the source raster.

Algorithm 1 Calculate required zoom level

```
 1: Input: One GeoTIFF $t$ from the dataset,
 2:        The Gridsize $s$ of one output cell in pixels
 3: procedure REQUIREDZOOM(GeoTIFF $t$, Gridsize $s$)
 4:     ▷ One final cell should at least cover this area
 5:     $requiredWidth = t.extent.width/t.width * s$
 6:     $requiredHeight = t.extent.height/t.height * s$
 7:     $requiredSize = (requiredWidth, requiredHeight)$
 8:
 9:     $center = t.middle$
10:     $zoom = 0$
11:
12:     while $cellAt(zoom, center).extent > requiredSize$ do
13:         $zoom = zoom + 1$
14:     end while
15:
16:     return $zoom$                         ▷ First level with no data loss
17: end procedure
```

4.3 Raster triangulation

In Steps ② and ④ of our conversion process (Figure 3), we apply a Delaunay triangulation to sets of points extracted from the input raster data and the generated mesh tiles. Our implementation of this triangulation algorithm follows the iterative approach presented by de Berg et al. [8] but uses a different strategy to select the points to be added to the mesh (see Algorithm 2).

First, we have to specify *corner points* C, which are the four points in the four corners of each tile. They will definitely be included in the resulting mesh in order to make sure that it covers the full size of the tile and that holes in the final rendering are avoided. The algorithm starts with a mesh that consists of two triangles that are based on the four points in C. In the process, more points are iteratively integrated into the mesh as vertices (line 13). For this, the point with the maximum distance to the mesh is extracted in line 12. This approach is based on the idea that adding the point with the maximum distance results in a high increase of the quality of the resulting mesh (see also [21]). For each point, its height is compared to HEIGHTAT(m, p) in Line 12. This function calculates the height of the mesh at the position of Point p. The point with the maximum distance ($argmax$) to the mesh is added to it.[3]

This is done until one of the given termination conditions is met. The parameter `maxPoints` defines the maximum number of points to use for the resulting mesh. As soon as this number is reached, no more triangles will be added to the mesh. The second termination condition is given by `maxError`. The algorithm will terminate if the maximum of all distances from the original points to

[3] Note that adding a point to the mesh may require edge flipping to ensure the triangles still meet the Delaunay condition. Details on this are beyond the scope of this paper. We refer to the original algorithm by de Berg et al. [8].

Algorithm 2 Triangulation of one tile

```
 1: Input: Set of Points $P$ which should be triangulated,
 2:        Corner Points $C$ which are included definitely,
 3:        $maxError$ allows a quality based termination,
 4:        $maxPoints$ to limit the resulting mesh size
 5: Output: Mesh $m$ approximating the given points
 6: procedure TRIANGULATE($P$, $C$, $maxError$, $maxPoints$)
 7:   ▷ The initial Mesh covers all corner points
 8:   Mesh $m = new$ Mesh($C$)
 9:   $error = $ CALCULATEERROR($m, P$)
10:
11:   while $error > maxError$ and $m.points.length < maxPoints$ do
12:     $point = argmax_{Point\ p \in P} |p.height - \text{HEIGHTAT}(m, p)|$
13:     $m.addPoint(point)$
14:     $error = $ CALCULATEERROR($m, P$)
15:   end while
16:
17:   return $m$
18: end procedure

19: ▷ Calculate the max distance from the Mesh m to a point in $P$
20: function CALCULATEERROR(Mesh $m$, Points $P$)
21:   return $\max_{Point\ p \in P} |p.height - \text{HEIGHTAT}(m, p)|$
22: end function
```

the mesh (as calculated in the function CALCULATEERROR(m, P)) is less than `maxError`. In this case, adding a new point would exceed the desired quality of `maxError`.

An error of 100 meters between the mesh and the real height is sufficient if the whole globe should be displayed, but the more you zoom in, the lower the value should be. We calculate `maxError` dynamically based on the following empirical formula:

$$maxError = \frac{150000}{2^{zoomLevel}}$$

On zoom level 0, `maxError` is 150 kilometers. This value is halved by every increase of the zoom level.

4.4 Merging mesh tiles

In the aggregation step (Figure 3, Step ③), we merge four tiles of level b to generate input data for triangulating the next, less detailed level $b - 1$ in such a way that it is not necessary to resample the whole dataset $\mathcal{D}$ again.

Based on our layout scheme, each tile has a size of 256×256 pixels. Merging four tiles results in a new tile of size $[0, 511]^2$. We resample the tile to $[0, 255]^2$ again to achieve a uniform tile format.

After the merging step, we can perform the triangulation described in Algorithm 2 again to get a new mesh (Step ④). The algorithm requires four *corner*

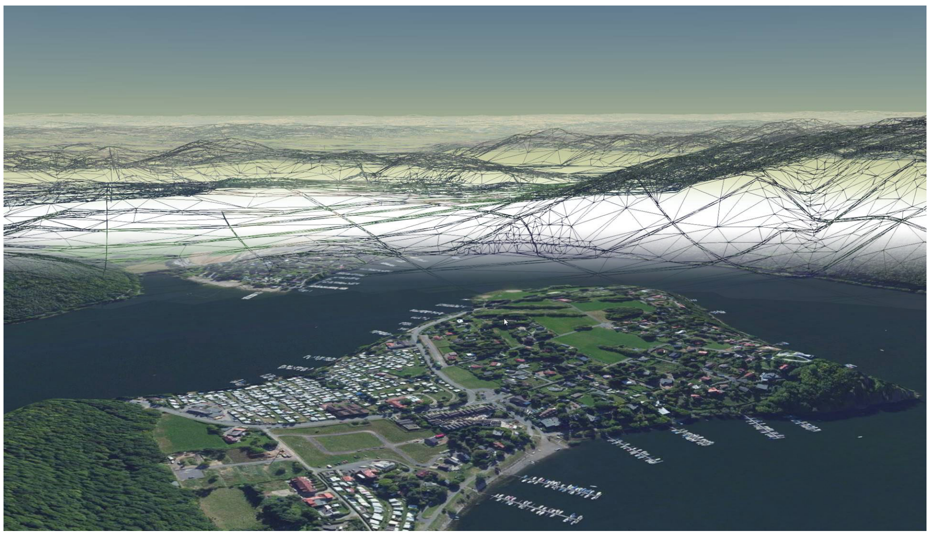

Fig. 4. A screenshot of our triangulated test dataset: terrain model with textures in the front and a wireframe in the back.

points to start with (see Section 4.3). They can be extracted from the corner points of the level below.

This process repeats until all levels have been generated (Step ⑤). Figure 4 shows a screenshot of the final result: the triangulated test dataset visualized in the web browser with Cesium.

5 Evaluation

In this section, we present the results from evaluating our approach and implementation based on a test dataset containing 973 terrain tiles (stored in the GeoTIFF file format) and covering the whole German Federal State of Hesse. Each of these tiles has a resolution of 5000×5000 pixels with one pixel per square meter, which results in a total area of $24\,325\,km^2$. The total data size is $84\,GB$. The copyright of the dataset is held by the Hessian State Office for Land Management and Geo-Information (HVBG). Publishing the data is not allowed, but it can be acquired through the online portal of the organization [14]. As mentioned in Section 4, we use a grid size of 256 pixels for the layout scheme. According to Algorithm 1, our bottom zoom level for an input resolution of 5000×5000 is 17.

In the following, we present results from measuring the runtime required to process the bottom level using different numbers of Spark executors (Section 5.1). We also evaluate two resampling techniques from GeoTrellis that affect the runtime of the overall process and discuss why this is the case (Section 5.2). Finally, we discuss benefits and drawbacks of using GeoTrellis (Section 5.3).

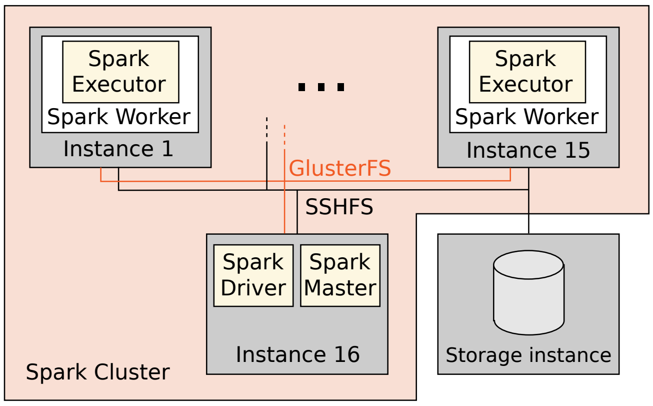

Fig. 5. The setup of our Spark cluster. We use up to 15 executors and GlusterFS as a distributed file system between them.

5.1 Scalability of the complete conversion process

To test the scalability, we set up a Spark cluster (see Figure 5) with up to 15 executors. They were running on an OpenStack [29] cluster with Ubuntu 18.04 as the operating system, two CPU cores and $8\,GB$ of RAM each. We set a memory limit of $4\,GB$ for Spark and disabled the swap to avoid any effects on the measurements. The Spark master and the driver were running together on a separate instance. All input data was stored on an additional instance and shared by an SSHFS mount [23] with the executors and the driver. The executors themselves stored their shuffle data in the distributed file system GlusterFS [32] (Version 5.6) that was spanned across all instances. This means the data had to be sent over the network each time for reading and writing. We used this setup for two reasons: First, our network connection was much faster than the HDD access. This means the additive network traffic did not affect our measurements too much. Second, the amount of shuffle data on each instance could change based on the task distribution. By using a distributed file system, we were able to calculate the required amount of memory more accurately and avoid running out of disk space during runtime.

To measure the degree of scalability of our approach, we increased the number of executors step by step from 1 to 15 and triangulated the complete test dataset each time. We used bilinear resampling (see Section 5.2) because it is faster and generates smoother meshes. Afterwards, we triangulated level 17 up to level 6. The measurement results are shown in Figure 6. Each bar represents the required processing time. This includes the initial loading and repartitioning of the source data (*Setup* step in blue) as well as the triangulation of the different levels (other parts of the bar).

We compared the runtime in relation to the amount of executors used. Given t_n represents the total runtime t for n executors, we calculated t_n/n to get the average runtime e_n for each of the n executors. In a perfect setup with linear

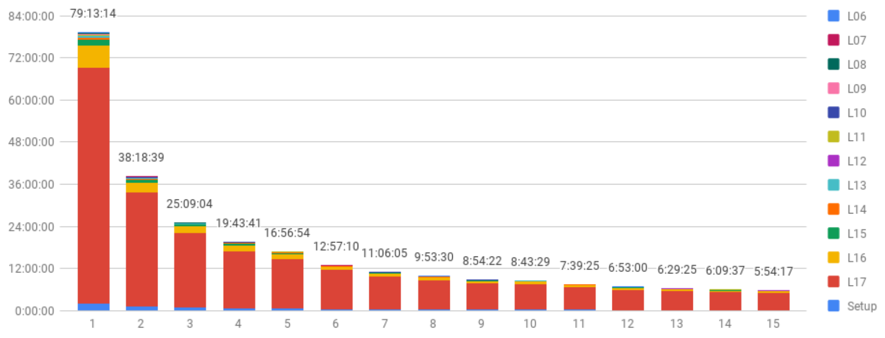

Fig. 6. Total runtime based on the amount of executors. The time for each level is visualized in different colors. Time format hh:mm:ss

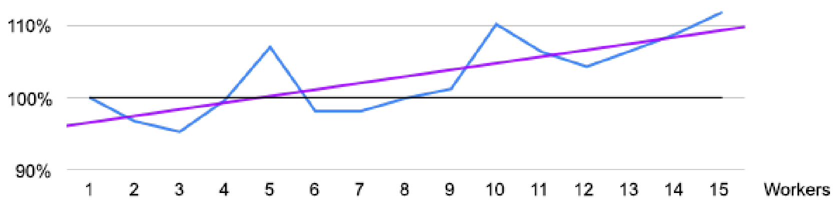

Fig. 7. The blue line represents the scaling factor of our approach. Lower values are better. A factor of 100% means linear scaling. The purple line marks the trend.

scaling, t_1 would equal e_1, e_2, ..., and e_{15}. We calculated the scaling factor of our approach $f_n = e_n/t_1$ and plotted the results in Figure 7.

The required runtime decreases almost linearly. In our scenario we lost approximately ten percent when scaling from one worker to 15. This is still a substantial speed up.

As shown in Figure 6, the bottom level (red part of the bar) requires most of the time compared to the other levels. This results from the high amount of points that are used as input for the triangulation in Algorithm 2. For each of these points, the distance to the current mesh has to be calculated. This is expensive if there are many points. Later levels can profit from the reduction of points in the first step.

A closer look at the system metrics shows that the CPU is the limiting factor during the triangulation of the leaf level 17. It is used to nearly 100% (see Figure 8), which is why better CPU performance would especially improve this step. The triangulation from level 16 to 6 has only a minor influence on the total time and the CPU is not even fully utilized. This results from the increased scheduling and communication overhead of many jobs being processed very fast.

The green line represents the Spark driver, which does not contribute to the calculations. This is why its CPU usage is very low during the triangulation.

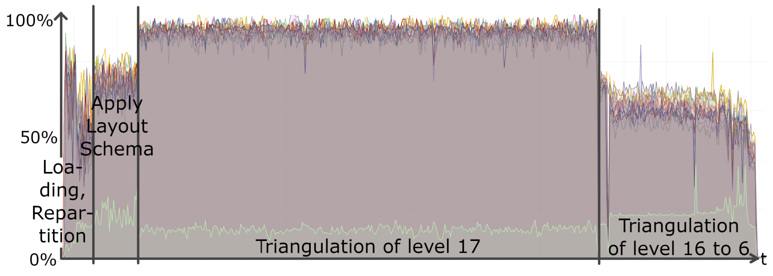

Fig. 8. CPU load while processing the data with three instances.

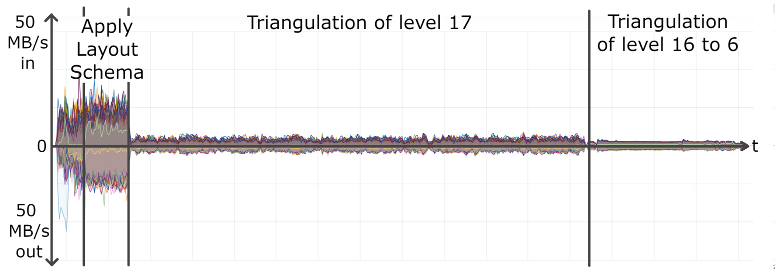

Fig. 9. Network usage while processing the data with 15 executors. Inbound traffic above the time-axis, outbound traffic below.

The setup phase (blue part in Figure 6) includes loading and repartitioning of the data as well as applying the layout scheme on it. During the loading phase the whole dataset is read from the disk. As mentioned before, the input data is shared using an SSHFS connection. You can see the data transmission in the peak of the blue line below the time axis in Figure 9. Afterwards, when the layout scheme is applied, all generated data has to be written back to disk. This time, the data becomes distributed over all executors using GlusterFS. As a result, you can see the high network utilization (inbound as well as outbound traffic) in Figure 9 and the sharp increase in disk usage in Figure 10.

After this period, the original data is no longer used and the traffic during triangulation is only caused by Spark accessing the shuffle files on GlusterFS.

It has to be noted that the network connection is not the bottleneck during the setup phase. The instances are connected with a bandwidth of $25\,GBit/s$, which would allow much more data to be transmitted. Instead, the used HDDs on the instances are the limiting factor. They cannot store the generated data fast enough. An upgrade to SSDs might result in a speed-up during the setup phase.

In our test, we used a memory limit of $4\,GB$ per executor. This is a high value and allows Spark to keep a lot of data in memory. In Figure 11, we can see that memory usage continuously grows until a point in the middle of triangulating

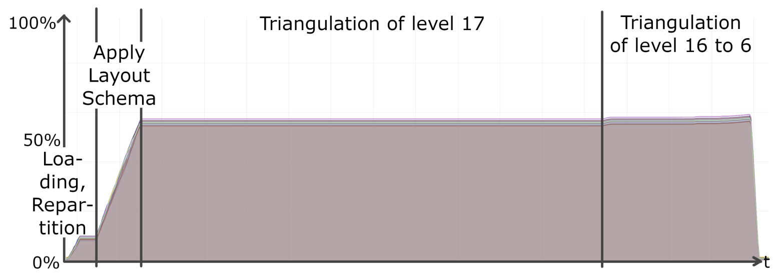

Fig. 10. Disk usage while processing the data with 15 executors.

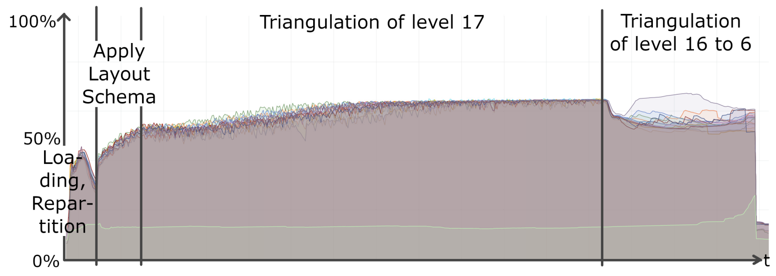

Fig. 11. Memory usage while processing the data with 15 executors.

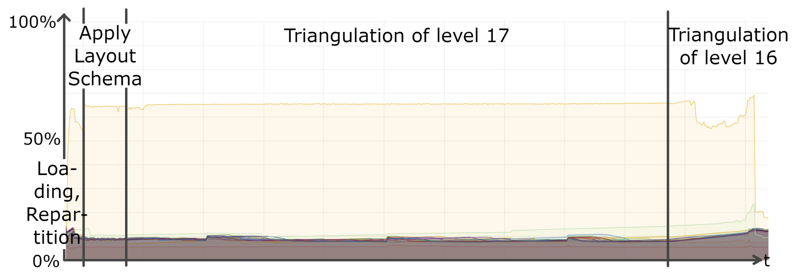

Fig. 12. Memory usage while processing the data with one executor.

level 17. When we look at only one executor (Figure 12), this limit is reached already during the loading and repartition phase. As Spark is not allowed to consume more memory than configured, it has to clean its storage iteratively. This might result in a slowdown of configurations with less executors compared to the ones with more because Spark can not keep data in memory. Theoretically, to analyze this influence in detail, a single executor with unlimited memory would have to be tested.

To summarize, we can conclude that our approach and implementation are suitable for the scalable processing of large terrain data. As shown in Figure 7, the overall runtime of the process scales almost linearly with the number of Spark executors. The bottlenecks described in this section are connected to available resources (main memory, CPU power, and disk bandwidth) as well as the fact that GeoTrellis needs to generate shuffle data.

5.2 Resampling techniques

As discussed in Section 4.1, GeoTrellis resamples the input data to align it to the layout scheme on the bottom zoom level. We inspected the Bilinear and Nearest Neighbor (NN) resampling methods for the initial conversion step (see Figure 3, Step ①) and how they influence the runtime of our triangulation. The resulting meshes can be seen in Figure 13. For us, it was especially interesting to compare the conversion time on deeper zoom levels, as this is the main bottleneck of the whole conversion process. The following results originate from the terrain generation based on a single 5000×5000 pixels GeoTIFF. As mentioned above, according to Algorithm 1, zoom level 17 is sufficiently fine enough to represent our input data. However, in the following, we compare it with the even finer level 18 to specifically demonstrate the differences in the resampling methods.

Comparing the runtime (see Figure 14), one can see that NN and bilinear sampling produce similar results for level 17. On zoom level 18, the conversion times diverge. NN is much slower than bilinear filtering. However, there is a downside of using bilinear sampling for the tile generation with GeoTrellis: The amount of shuffle data in Spark increases a lot (see Figure 15).

When converting our whole dataset at level 17, the total amount of shuffle data for bilinear sampling is $440\,GB$ compared to $165\,GB$ when using NN, but the runtime still decreases. In our case, this implies that bilinear sampling is

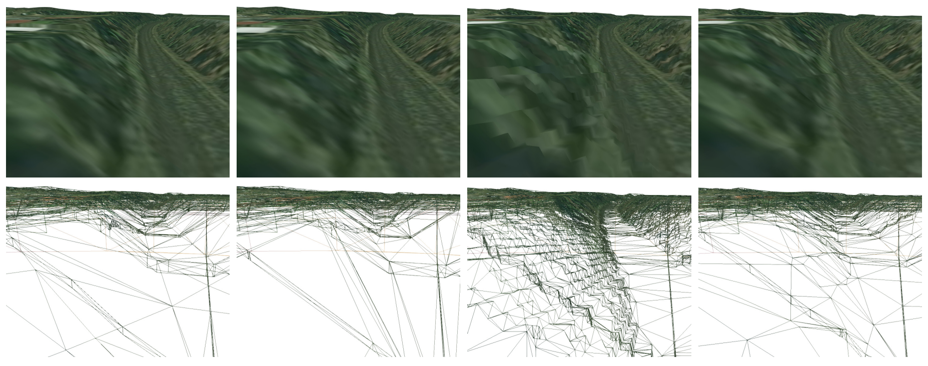

Fig. 13. Generated terrain meshes for zoom level 17 and 18 with different resampling methods

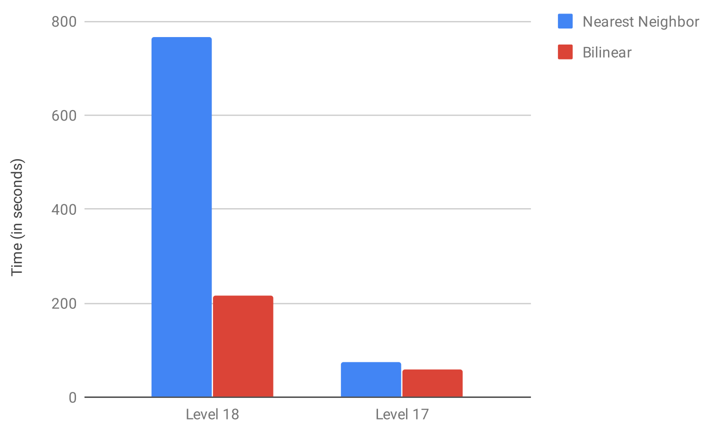

Fig. 14. Conversion times of the resampling of gridded height data.

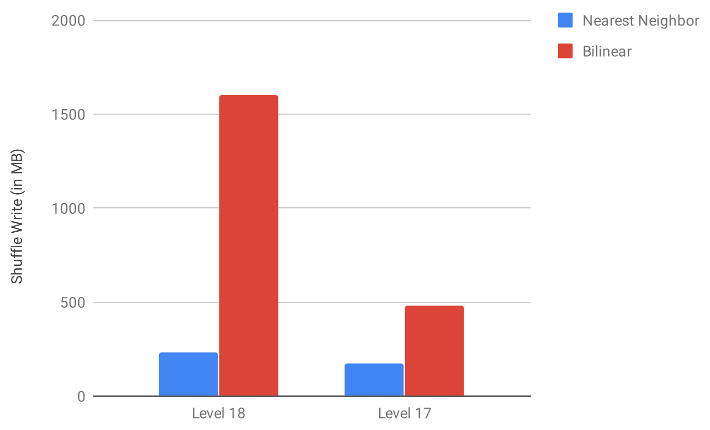

Fig. 15. Total shuffle write depending on the chosen resampling techniques.

superior, as long as the Spark instance has enough disk space to handle the shuffle data. This is counter-intuitive as bilinear filtering requires more calculations to get the interpolated value, while NN just needs to find the closest point.

When inspecting the geometry of the generated meshes (see Figure 13), we can observe that NN on level 18 yields meshes containing geometries neither present in the result of bilinear sampling nor in any results of the conversions on zoom level 17. This behavior is due to the fact that our used dataset has a resolution that approximatively matches the pixel size of the sampled tiles at zoom level 17 (see Section 4.2). Level 17 is the first level without any data loss. Therefore, NN sampling at level 18 produces tiles that contain pixel duplicates from the original raster dataset $\mathcal{D}$. Because of this, the triangulation needs more time to filter relevant points. This leads to an increased runtime that is much greater than the saving during the setup phase. Additionally, NN produces a stair-like effect visible in the triangulated meshes in Figure 13. Bilinear sampling at zoom level 18 removes these pixel duplicates by smoothing the height information in the tiles and therefore yields similar results as both triangulations on level 17.

This implies that bilinear sampling should be preferred as it produces good results with lower runtime compared to NN sampling.

5.3 Benefits and drawbacks of using GeoTrellis

We used GeoTrellis because it provides a lot of functionality and made development faster. Summing up, it provides the following benefits:

– **Loading of GeoTIFFs.** GeoTrellis can load GeoTIFF files and handle different spatial reference systems. We do not need to manage file accesses and can directly use the available data.
– **Applying layout scheme.** For the output files, we have to comply with the specification of the Tile Map Service (TMS). For each zoom level, the generated terrains have to be aligned based on this specification. GeoTrellis provides an easy way to crop the complete dataset in the required parts.
– **Integration with Apache Spark.** We want to compute the output in a distributed environment. Because of the strong integration of GeoTrellis with Apache Spark, we do not need to spend additional effort for parallel execution on multiple instances.

Nevertheless, GeoTrellis has a few drawbacks:

– **Usage of raster data.** GeoTrellis is focused on raster data. Whenever operations are done, new rasters are created. This behavior can correlate with a loss of precision. The results became visible in Section 5.2 and Figure 13 (third column). Whenever the required data granularity does not match with the input data, resampling is required. This can lead to wrong outputs depending on the resampling method.
– **No control of processing steps.** GeoTrellis processes the levels one by one. This leads to a lot of shuffle data, because the calculation results of one level are required for the next one. Until this point is reached in the processing pipeline, the data has to be stored on disk. A custom implementation could process less detailed levels as soon as the required subtiles exist. This would reduce the amount of shuffle data.
– **No control of the job distribution.** Apache Spark handles the distribution of jobs in the background. Especially if the input data is stored in a distributed file system, the calculation speed could be increased if tiles are processed on the instance where they are stored. In this case, the data does not need to be transmitted over the network.

To summarize, GeoTrellis is a good tool to get quick results. It provides a lot of functionality and can easily be set up. However it sets limits to possible improvements and is restricted to raster data.

6 Conclusions and Future Work

Our focus in this paper was to develop a scalable approach to create a hierarchical level-of-detail data structure optimized for web-based visualization. The main contribution is our approach to distribute the processing across a cloud infrastructure and to leverage available resources to scale almost linearly. In order to achieve this, we analyzed the target format and mapped the data and processing structure to the Apache Spark framework. This way, we could parallelize the triangulation by splitting the input data into smaller tiles and processing them individually. The parallelization is managed automatically by Spark and distributed to so-called executors. This data-driven division of processing steps into deployable standalone jobs enables the scalability of our system with regard to the amount of data.

Based on one stable and efficient configuration, we tested our system with a terrain model dataset consisting of $84\,GB$ of GeoTIFF files. We did several runs using this dataset, incrementally increasing the utilized executors. Figure 6 visually compares the resulting runtimes and proves that our system is capable of almost linearly reducing the runtime with regard to the utilized cloud resources.

Our evaluation reveals that the initial setup phase of Spark as well as the CPU usage during triangulation for the bottom level of the hierarchy leaves room for improvements. In a future work, we will investigate the removal of the storage instance and instead use the disk space of the executor or a distributed file system to reduce the I/O and network overhead at the beginning of processing. To reduce the CPU usage for creating the TINs, we will look into other algorithms creating error-bound TINs such as simplification algorithms using quadric error metrics.

References

1. Apache Software Foundation: Apache Storm. http://storm.apache.org/ (2015) accessed: 2019-12-01.
2. Apache Software Foundation: Apache Spark - Unified Analytics Engine for Big Data. https://spark.apache.org/ (2018) accessed: 2019-12-01, Version used: 2.20.
3. Apache Software Foundation: Apache Spark Streaming. https://spark.apache.org/streaming/ (2018) accessed: 2019-12-01.
4. Azavea: GeoTrellis Main Page. https://geotrellis.io/ (2019) accessed: 2019-12-01, Version used: 1.2.1.
5. Cesium Consortium: quantized-mesh-1.0 terrain format. https://cesiumjs.org/data-and-assets/terrain/formats/quantized-mesh-1.0/ (2018) accessed: 2019-12-01.
6. Cesium Consortium: 3d-tiles - Specification for streaming massive heterogeneous 3D geospatial datasets. https://github.com/AnalyticalGraphicsInc/3d-tiles (2019) accessed: 2019-12-01.
7. Cesium Consortium: CesiumJS - Geospatial 3D Mapping and Virtual Globe Platform. https://cesiumjs.org/ (2019) accessed: 2019-12-01, Version used: 1.64.
8. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational Geometry: Algorithms and Applications. Springer (2008)

9. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. Commun. ACM **51**(1) (2008) 107–113
10. Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P.J., Mayani, R., Chen, W., Ferreira da Silva, R., Livny, M., Wenger, K.: Pegasus: a workflow management system for science automation. Future Generation Computer Systems **46** (2015) 17–35
11. Delaunay, B.N.: Sur la sphère vide. Bulletin of Academy of Sciences of the USSR **6** (1934) 793–800
12. Esri: I3S-SPEC. https://github.com/Esri/i3s-spec (2017) accessed: 2019-12-01.
13. Goodrich, M.T.: Randomized fully-scalable bsp techniques for multi-searching and convex hull construction. In: Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms. SODA '97, Society for Industrial and Applied Mathematics (1997) 767–776
14. Hessisches Landesamt für Bodenmanagement und Geoinformation: Hessian Geodata Portal: Geodaten online. https://www.gds.hessen.de/ (2019) accessed: 2019-12-01.
15. Hu, L., Sander, P.V., Hoppe, H.: Parallel view-dependent refinement of progressive meshes. In: Proceedings of the 2009 symposium on Interactive 3D graphics and games, ACM (2009) 169–176
16. Isenburg, M., Liu, Y., Shewchuk, J., Snoeyink, J.: Streaming computation of Delaunay triangulations. ACM transactions on graphics (TOG) **25**(3) (2006) 1049–1056
17. Krämer, M.: A Microservice Architecture for the Processing of Large Geospatial Data in the Cloud. PhD thesis, Technische Universität Darmstadt (2018)
18. Krämer, M., Senner, I.: A modular software architecture for processing of big geospatial data in the cloud. Computers & Graphics **49** (2015) 69–81
19. Kreps, J.: Questioning the lambda architecture. https://www.oreilly.com/ideas/questioning-the-lambda-architecture (2014) accessed: 2019-12-01.
20. Lavender, S., Lavender, A.: Practical Handbook of Remote Sensing. 1 edn. CRC Press (2015)
21. Lee, D.T., Schachter, B.J.: Two algorithms for constructing a Delaunay triangulation. International Journal of Computer & Information Sciences **9**(3) (1980) 219–242
22. Li, J., Humphrey, M., Agarwal, D.A., Jackson, K.R., van Ingen, C., Ryu, Y.: eScience in the cloud: A MODIS satellite data reprojection and reduction pipeline in the Windows Azure platform. In: IEEE International Symposium on Parallel & Distributed Processing (IPDPS). (2010) 1–10
23. Libfuse: SSHFS - A network filesystem client to connect to SSH servers. https://github.com/libfuse/sshfs (2019) accessed: 2019-12-01.
24. Liu, K., Boehm, J., Alis, C.: Change detection of mobile lidar data using cloud computing. ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences **XLI-B3** (2016) 309–313
25. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E.A., Tao, J., Zhao, Y.: Scientific workflow management and the Kepler system: Research articles. Concurr. Comput. : Pract. Exper. **18**(10) (2006) 1039–1065
26. Marz, N., Warren, J.: Big Data: Principles and Best Practices of Scalable Realtime Data Systems. 1 edn. Manning Publications Co. (2015)
27. Nath, A., Fox, K., Agarwal, P.K., Munagala, K.: Massively parallel algorithms for computing TIN DEMs and contour trees for large terrains. In: Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. SIGSPACIAL '16, ACM (2016) 25:1–25:10

28. Open Geospatial Consortium: OpenGIS Web Map Tile Service Implementation Standard. https://www.ogc.org/standards/wmts (2010) accessed: 2020-03-13.
29. OpenStack Foundation: Build the future of Open Infrastructure. https://www.openstack.org (2019) accessed: 2019-12-01.
30. OSGeo: Tile Map Service Specification. http://wiki.osgeo.org/wiki/Tile_Map_Service_Specification (2012) accessed: 2019-12-01.
31. Qazi, N., Smyth, D., McCarthy, T.: Towards a GIS-based decision support system on the Amazon cloud for the modelling of domestic wastewater treatment solutions in Wexford, Ireland. In: 15th International Conference on Computer Modelling and Simulation (UKSim). (2013) 236–240
32. Red Hat, Inc.: Gluster — Storage for your Cloud. https://www.gluster.org/ (2019) accessed: 2019-12-01, Version used: 5.6.
33. Reinsel, D., Gantz, J., Rydning, J.: Data age 2025 - The evolution of data to life-critical. An IDC white paper, sponsored by Seagate. (2017)
34. Spielman, D.A., Teng, S.H., Üngör, A.: Parallel delaunay refinement: Algorithms and analyses. International Journal of Computational Geometry & Applications **17**(01) (2007) 1–30
35. Stöckli, R., Vermote, E., Saleous, N., Simmon, R., Herring, D.: The blue marble next generation - a true color earth dataset including seasonal dynamics from MODIS. Published by the NASA Earth Observatory (2005)
36. Ulrich, T.: Rendering massive terrains using chunked level of detail control. In: SIGGRAPH Course Notes. Volume 3. (2002)
37. Warren, M.S., Brumby, S.P., Skillman, S.W., Kelton, T., Wohlberg, B., Mathis, M., Chartrand, R., Keisler, R., Johnson, M.: Seeing the earth in the cloud: Processing one petabyte of satellite imagery in one day. In: IEEE Applied Imagery Pattern Recognition Workshop (AIPR). (2015) 1–12