A PARALLEL ALGORITHM FOR LOCAL POINT DENSITY INDEX COMPUTATION OF LARGE POINT CLOUDS

Anh Vu Vo[a,*], Chamin Nalinda Lokugam Hewage[a] , Nhien An Le Khac[a], Michela Bertolotto[a], Debra Laefer[b,c]

[a]School of Computer Science, University College Dublin, Ireland - anhvu.vo@ucd.ie,
chamin.lokugamhewage@ucdconnect.ie, (an.lekhac, michela.bertolotto)@ucd.ie
[b]Center for Urban Science and Progress, New York University, USA - debra.laefer@nyu.edu
[c]Department of Civil and Urban Engineering, New York University, USA

KEY WORDS: Point cloud, LiDAR, Parallel, Distributed, Density, Local Point Density Index, Apache Spark.

ABSTRACT:

Point density is an important property that dictates the usability of a point cloud data set. This paper introduces an efficient, scalable, parallel algorithm for computing the local point density index, a sophisticated point cloud density metric. Computing the local point density index is non-trivial, because this computation involves a neighbour search that is required for each, individual point in the potentially large, input point cloud. Most existing algorithms and software are incapable of computing point density at scale. Therefore, the algorithm introduced in this paper aims to address both the needed computational efficiency and scalability for considering this factor in large, modern point clouds such as those collected in national or regional scans. The proposed algorithm is composed of two stages. In stage 1, a point-level, parallel processing step is performed to partition an unstructured input point cloud into partially overlapping, buffered tiles. A buffer is provided around each tile so that the data partitioning does not introduce spatial discontinuity into the final results. In stage 2, the buffered tiles are distributed to different processors for computing the local point density index in parallel. That tile-level parallel processing step is performed using a conventional algorithm with an R-tree data structure. While straight-forward, the proposed algorithm is efficient and particularly suitable for processing large point clouds. Experiments conducted using a 1.4 billion point data set acquired over part of Dublin, Ireland demonstrated an efficiency factor of up to 14.8/16. More specifically, the computational time was reduced by 14.8 times when the number of processes (i.e. executors) increased by 16 times. Computing the local point density index for the 1.4 billion point data set took just over 5 minutes with 16 executors and 8 cores per executor. The reduction in computational time was nearly 70 times compared to the 6 hours required without parallelism.

1. INTRODUCTION

The term point cloud is often used to indicate a set of discrete, three-dimensional (3D) sampling points that represent some spatial entities (e.g., urban infrastructure, building objects, terrain surfaces). Point clouds are usually acquired by Light Detection And Ranging (LiDAR) or photogrammetric techniquess. Point clouds are recognised as an increasingly prominent part of the spatial data infrastructure and a distinguishable kind of spatial data apart from raster and vector data (van Oosterom et al., 2015; Julin et al., 2018). An important property of a point cloud data set is its point density, which measures the number of sampling points per unit area. However, a summation of points on an apparently flat surface and the division of that number over the horizontal projection of the local surface provides an uninformative measure of density due to the high degree of heterogeneity in urban aerial point clouds and in many ways is only an indicator of the upper bound of horizontal coverage (Stanley and Laefer, 2021). Arguably, the division of the all points in the point cloud by the entire horizontal spatial extent is even less informative. Understanding a more localized nature of the density of a point cloud is important, as it has direct implications on its usability in downstream applications (Renslow, 2012). Specifically, insufficient densities can preclude tasks such as detection of objects and features smaller than a certain size. Thus, point density is a primary criterion in point cloud data acquisition and analysis (Heidemann, 2018). While there are various ways to compute the density of a point cloud, such as (Lari and Habib, 2012; Bethel, 2019), computing the exact density at each location across a large point cloud data set is a computationally demanding problem, because of the need to consider the local neighbourhood of each point. Knowing the point density as a function of position with a data set can help inform the likely success of segmentation or object detection algorithms.

To address the issue, this paper introduces an efficient parallel algorithm for computing the Local Point Density index (LPD), a relatively sophisticated density metric. Compared to conventional density computation approaches such as two-dimensional (2D) gridding (Bethel, 2019), LPD is capable of capturing the density on horizontal, as well as non-horizontal, surfaces at the point level. The computation of LPD of a point cloud relies on analysing the local neighbourhood of each individual point in the point cloud. The parallel algorithm proposed in this paper aims to compute the LPD index of large point clouds in a timely manner. Firstly, this is done by enabling out-of-core computation, thereby circumventing the need to load the entirety of the point cloud into the main memory, which removes memory capacity as the limiting factor with respect to data volume. Secondly, the algorithm is parallel, thereby allowing multiple processors to share the computing workload – leading to a significant reduction in computation time. The capability of the algorithm to compute LPD for large point clouds rapidly is particularly important to cope with the massive volumes of point cloud data being collected, which can be billions of points per square kilometer (e.g. AHN, 2014; Laefer et al., 2017; Vo et al., 2016).

* Corresponding author

One of the main challenges in developing the parallel algorithm is partitioning the data without interrupting the spatial continuity inherent to the data. The algorithm proposed in this paper is inspired by the tile-based parallelisation with the use of spatial buffers introduced by Li et al. (2017). Rather than replicating or simply adopting (Li et al., 2017), this research extends those concepts by introducing a parallel pre-processing step to structure the input point cloud into buffered tiles. In addition, the research provides an extended formalisation and in-depth analysis of the computational efficiency of the algorithm and rigorously evaluates the actual computational performance through large-scale experiments. In particular, the research makes the following contributions:

- novel, point-level parallelisation approach to organise points into buffered tiles
- extension of the tile-level parallelisation approach by Li et al. (2017) for computing the LPD index
- analysis of the computational efficiency of the complete parallel algorithm for LPD computation
- demonstration of the ability of the algorithm to compute the LPD index for a large point cloud rapidly by distributing the computation across a distributed memory system.

2. RELATED WORK

This section reviews research related to point density computation and parallel point cloud processing. This starts with the most simplistic such as that recommended by the US Geological Survey's LiDAR Base Specification (Heidemann, 2018), which involves computing the average density for a representative polygonal area larger than 1 km^2. As noted by Bethel (2019), the simplistic representative area approach is insufficient to comprehensively represent the actual density of the entire point cloud, as the mechanism cannot capture local differences. More rigorous alternatives include the Voronoi/Thiessen polygon approach and the gridding approach (Naus, 2010; Bethel, 2019). Those approaches are better at capturing the local point density distribution compared to a representative area approach. However, all of the aforementioned approaches project the 3D data onto a 2D, horizontal plane, thereby obfuscating the actual point distribution, which is inherently 3D. That limitation particularly affects non-horizontal surfaces, which are rarely seen in topographic mapping, but commonly captured in terrestrial and mobile LiDAR,as well as low-altitude airborne LiDAR (Laefer et al., 2017, 2014).

Aiming to address the limitation of the 2D approaches, Lari and Habib (2012) introduced three alternatives for computing the LPD in 3D. The first, referred to as the approximate approach, assumes that the local spherical neighbourhood around the point in question contains adjacent points that approximate a planar surface. The LPD index of a point is, thus, calculated as $LPD = (n+1)/(\pi r^2)$, where r is the radius of the spherical neighbourhood of the point in question, and n is the number of points in that neighbourhood. The approximate LPD index is available in notable point cloud software such as CloudCompare and is widely used in other research (e.g. Vo et al., 2015; Xu et al., 2018; Saglam et al., 2020; Ye et al., 2020). The other two approaches introduced by Lari and Habib (2012) aimed to verify the planarity assumption and exclude non-planar points in the spherical neighbourhood prior to the LPD computation. Compared to the approximate approach, the accuracy of the latter approach is improved at the cost of a higher computational cost.

As LiDAR and photogrammetry technologies advance rapidly, the size and density of point clouds are rapidly increasing in both volume and density. A common approach to accelerate data analyses and accommodate large data volumes is parallel computing. Developing parallel point cloud data analysis algorithms is an active research topic. There is a large body of research work (e.g. Wu et al., 2011; Che and Olsen, 2018; Zhang et al., 2011; Bodenstein et al., 2015; Brédif et al., 2015; Vo and Laefer, 2019) that has investigated both the main types of parallel systems (shared-memory and distributed-memory) for point cloud data analysis. In a shared memory system, all processors share access to the computer's memory. A parallel program that can exploit that type of system is called a multi-threading program. In a distributed-memory system, each processor (called a node) has its private memory and functions similar to an autonomous computer. Individual computing nodes in a distributed-memory system exchange data through explicit communication over a network (e.g. message passing). Compared to shared-memory systems, distributed-memory systems are more difficult to program and are typically not as fast at computing due to significantly higher communication overheads. However, distributed-memory systems can be scaled much more easily and far less expensively (Kleppmann, 2017). Examples of research on multi-threading point cloud analysis algorithms include the parallel Delaunay triangulation algorithm by Wu et al. (2011), the point cloud registration algorithm by Martinez et al. (2013), and the spatial segmentation algorithm by Che and Olsen (2018). The efficiency of multi-threading parallelism was demonstrated in all of those.

The use of distributed memory systems for point cloud data analysis has spawned significant quantities of research. For example, Zhang et al. (2011) and Bodenstein et al. (2015) independently introduced different spatial clustering algorithms that can exploit the high scalability of distributed memory systems. Both research groups followed the explicit parallel programming approach and used the Message Passing Interface (MPI) framework. With MPI, programmers must explicitly instruct how specific processors perform their tasks and coordinate with other processors. Major complexity such as avoiding race and deadlock must be handled by the programmer. Consequently, explicit parallel programming, such as MPI, is powerful method but complex. A simpler way to program distributed-memory systems is to employ frameworks such as MapReduce (Dean and Ghemawat, 2008). That framework abstracts away the actual complexity of parallel programming and provides a simple interface that programmers use to formulate their computational problems. Such an approach is simple and more accessible but usually less computationally efficient when compared to the explicit approach. There are two common implementations of MapReduce: Hadoop MapReduce (https://hadoop.apache.org/) and Spark (https://spark.apache.org/). Examples of Hadoop Map Reduce for point cloud data analysis can be seen in (Aljumaily et al., 2016) and (Krishnan et al., 2010). Its successor, Spark, is typically 10-100s times faster than Hadoop MapReduce due to its more efficient memory usage (Zecevic and Bonaci, 2017). In addition, Spark provides a richer interface so that applications need not follow the rigid structure of one map and one reduce function, as per the original MapReduce framework. The use of Spark for LiDAR point cloud analysis is seen in various research for a diverse range of purposes including orthographic

image rendering (Brédif et al., 2015) and change detection (Liu et al., 2016) and solar radiation simulation with point clouds (Vo et al., 2019).

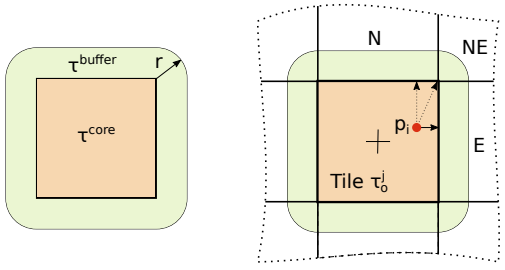
Tiling, rasterisation, and voxelisation are commonly seen in MapReduce and Spark applications for point cloud data analysis (e.g. Krishnan et al., 2010; Rizki et al., 2017; Liu et al., 2016; Li et al., 2017). These techniques group points into spatially coherent groups either in 2D (i.e. tiling, rasterisation) or in 3D (voxelisation). The groups of points (i.e. tiles, cells, or voxels) are then distributed to different processors for parallel processing. The key in applying such a strategy is in decoupling the overall computation into separate cell- or voxel-based computations without introducing spatial discontinuities or other artifacts in the ultimate results. Li et al. (2017) recommended a spatial buffer around each data partition to avoid the spatial discontinuity. The approach was demonstrated as general-purpose, as it could parallelise a class of point cloud processing tasks involving neighbouring information. Notably, the parallelisation introduced by Li et al. (2017) was at the tile level. The parallel processing was conducted on the basis of a cluster of tiles (called a subdomain), and the spatial buffer was set as the whole tile size; the input point clouds were assumed to be already structured in tiles prior to entering the parallel processing workflow. Decomposing an unorganised point cloud into tiles was not part of the algorithm. While techniques such as rasterisation and voxelisation, in addition to the use of spatial buffers, have successfully facilitated the parallelisation of several point cloud processing problems, there is not a known approach to parallelising and scaling out the point density computation to multiple nodes of a distributed-memory system. That is the topic of this paper.

3. METHODOLOGY

This research introduces a more effective algorithm to computing the local point density index based on data parallelism and is implemented in Apache Spark. The key operation in computing the local point density index for a point data set P is the 3D range search for each individual point in P. The search retrieves for each point $p_i \in P$ all neighbouring points $p_{nj} \in P$ that have $d(p_i, p_{nj}) < r$; where r is a user-defined radius and $d(p_i, p_{nj})$ is the 3D Euclidean distance between p_i and p_{nj}. Let n be the number of neighbours of p_i, and r be the maximum distance between p_i and all p_{nj}. The LPD index of point p_i is computed as $LPD = (n+1)/(\pi r^2)$. As P often consists of a large number of points (e.g. billions to hundreds of billions), the 3D range search is computationally expensive and dictates the overall computational cost of the point density determination. A conventional approach to the range search is to use a spatial data structure such as an R-tree (Guttman, 1984) or an octree (Samet, 2006). By using such a data structure, the time required to perform a range search is t_{search}, which is proportional to the logarithm of N (i.e. the number of points in P). Performing a range search for all points in P takes $N \cdot t_{search}(\log N)$, referred to as T_{search} in Equation 1. In addition to the 3D range search, as shown in Equation 1, the total cost of the conventional, serial approach (T_{serial}) consists of the costs for constructing the tree, $T_{tree} = N \cdot t_{tree}(\log N)$ and computing the LPD, $T_{LPD} = N \cdot t_{LPD}$. In Equation 1, t_{tree} proportional to $\log N$ is the time required to insert one point into the R-tree, and t_{LPD}, which is independent of the data size, is the time to compute the LPD index for one query point.

$$\begin{aligned} T_{serial} &= T_{tree} + T_{search} + T_{LPD} \\ &= N \cdot t_{tree}(\log N) + N \cdot t_{search}(\log N) + N \cdot t_{LPD} \end{aligned} \quad (1)$$

The algorithm is composed of two stages. During the first stage, the input point cloud, P, is partitioned into 2D buffered tiles that partially overlap each other (Figure 1). Each tile, τ, contains a rectangular core region, τ^{core}, which does not overlap the core region of any other tile. Around the core region of each tile, a buffer, τ^{buffer}, of size r is created to ensure that the data partitioning does not introduce any spatial discontinuity in the ultimate results (Figure 1a). In the second stage, as the point cloud has been partitioned into buffered tiles, the LPD index can be computed for each tile separately.



(a) Buffered tile (b) Point mapped to tiles

Figure 1. Spatial decomposition into buffered tiles

Algorithm 1: Map point to buffered tiles

Input: $p_i(x, y, z)$ ▷ input point
Output: $T = \texttt{Set}_\texttt{j}([\tau_j, p_i])$ ▷ buffered tiles

1 **Function** `MapPointToTiles`(p_i)**:**
2 $\quad T \leftarrow \emptyset$
3 $\quad X_j^\circ \leftarrow \texttt{round}((p_i.x - x_\circ)/\Delta_x)$
4 $\quad Y_j^\circ \leftarrow \texttt{round}((p_i.y - y_\circ)/\Delta_y)$
5 $\quad \tau_j^\circ \leftarrow [X_j^\circ, Y_j^\circ]$
6 $\quad T \xleftarrow{add} [\tau_j^\circ, p_i]$
7 $\quad T_{neighbours} \leftarrow \texttt{Neighbours}(\tau_j^\circ)$
8 $\quad$ **foreach** $\tau_j \in T_{neighbours}$ **do**
9 $\quad\quad$ **if** $d(\tau_j, p_i) < r$ **then**
10 $\quad\quad\quad T \xleftarrow{add} [\tau_j, p_i]$
11 $\quad\quad$ **end**
12 $\quad$ **end**
13 $\quad$ **return** T
14 **End Function**

Both stages of the proposed algorithm are performed in parallel using multiple processors. In the first stage, each processor is given several groups of points (i.e. data partitions). The processor works on one partition at a time so that it does not have to load all partitions into the main memory. Such a solution is known as an out-of-core solution, which is suitable for large data sets that do not fit in a processor's memory. The processor maps each point p_i in a partition to all buffered tiles that contain p_i. That operation is performed using the `MapPointToTiles` function in Algorithm 1. As the tile cores are disjoint, there is only one tile, τ_j°, that contains p_i in its core. The index of τ_j°, which is a 2-tuple index $[X_j^\circ, Y_j^\circ]$, is computed using the two simple rounding functions on lines 3 and 4 of Algorithm 1. In the functions, $(x_\circ, y_\circ)$ is an arbitrary point defining the origin of the tiling grid and Δ_x and Δ_y are the dimensions of each tile. In addition to τ_j°, p_i can belong to several neighbours of

τ_j°, if the distance from the neighbours to p_i is shorter than the radius r. Lines 8-12 of the algorithm check the neighbours of τ_j° for those containing p_i. Given that r is smaller than the tile size, a maximum of eight direct neighbours of τ_j° have to be checked (Zlatanova et al., 2016). The number of tiles can be further reduced by considering position of p_i relative to the centre of τ_j°. For example, in Figure 1b, p_i is to the North-East of the centre of τ_j°. Thus only three neighbours (i.e. North, East, and North-East) of τ_j° have to be checked using the tile to point distance condition. Nevertheless, the search for neighbouring tiles is always $\mathcal{O}(1)$, because the neighbouring tiles can be located using their 2D grid indices. For example, the 2D index of the North-East neighbour of tile $\tau_j^\circ([X_j^\circ, Y_j^\circ])$ is $[X_j^\circ + 1, Y_j^\circ + 1]$. Consequently, mapping one point to all of its tiles also takes $\mathcal{O}(1)$.

Apart from the point coordinates, all other inputs of the `MapPointToTiles` function are constant. Thus, as seen in the data lineage in Figure 2, multiple partitions of points can be processed completely and in parallel by different processors, which do not need to synchronise data with other processors. The parallelism in this stage is referred to as the point-level parallelism, as opposed to the tile-level parallelism in stage 2. Using k processors to map all N points in P to their tiles takes $\frac{N}{k} \cdot t_{tiling}$, where t_{tiling} is the time required to map one point to its tile and is independent of the total data size. The cost for tiling data makes up the first term (i.e. T_{tiling}) in the total cost of the parallel algorithm (Equation 2). Communication across processors happens at the subsequent step where the points are aggregated by the tiles to which they belong. In other words, the $[\tau_j, p_i]$ pairs resulting from the previous map function are aggregated by the tile index $\tau_j(X_j, Y_j)$. That aggregation is implemented using the built-in `aggregateByKey` function in Spark. While the aggregation is known to be highly optimised and is performed in parallel (Zecevic and Bonaci, 2017), the actual implementation is part of Apache Spark and is not managed by the authors. In this paper, the computational cost of that aggregation is treated as an unknown (T_{aggr} in Equation 2).

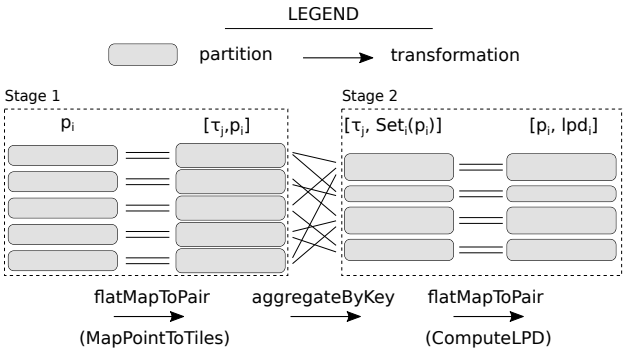


Figure 2. Data lineage

During stage 2 of the algorithm, the LPD index computation is performed for each separate buffered tile, τ_j using the `ComputeLPD` function in Algorithm 2. In the first step (lines 2-5 of Algorithm 2), a local R-tree is created for all points in τ_j to aid the 3D range search. Considering m as the number of tiles, the average time for k processors to construct all local R-trees for the point data set P is $\frac{N'}{k} \cdot t_{tree}(\log \frac{N'}{m})$, where $N' > N$ is the number of points including the duplication due to buffering. At the subsequent step (lines 7-11 of Algorithm 2), the range search is performed for every point in the core of τ_j using the corresponding local R-tree. The total time of the search is $\frac{N}{k} \cdot t_{search}(\log \frac{N'}{m})$. The final step, LPD computation, takes $\frac{N}{k} \cdot t_{LPD}$. As shown in Algorithm 2 and the data lineage in Figure 2, all operations in stage 2 are contained within each separate tile. A processor manipulating a tile only needs the point data within that tile, thereby circumventing the need to synchronise or exchange data with other processors. Notably, the computation in stage 2 is also out-of-core, since the processors only need to load the subset of tiles relevant to the current computation in the main memory.

Algorithm 2: Compute LPD per tile

Input: $\tau_j = \tau_j^{core} \cup \tau_j^{buffer}$ ▷ tile data
Output: $D = \texttt{Set}_\texttt{i}([p_i, lpd_i])$ ▷ local point density index

```
1  Function ComputeLPD(τ_j):
2      Rtree ← ∅
3      foreach p_i ∈ τ_j do
4          Rtree ⟵(insert) p_i
5      end
6      D ← ∅
7      foreach p_i ∈ τ_j^core do
8          η ← RangeSearch(p_i, RTree)
9          lpd_i ← LPD(p_i, η)
10         D ⟵(add) [p_i, lpd_i]
11     end
12     return D
13 End Function
```

The total computational time of the parallel algorithm is shown in Equation 2. All of the terms composing $T_{parallel}$ are inversely proportional to k, except for the unknown T_{aggr}. While the aggregation is not directly controlled, the efficiency of the operation, which relies on the `aggregateByKey` function in Spark, is observable from the relationship between reduction of the total cost when increasing k. If the `aggregateByKey` function is highly parallel or T_{aggr} is insignificant compared to the total cost, $T_{parallel}$ should be inversely proportional to the number of processors, k.

$$\begin{aligned} T_{parallel} &= T_{tiling} + T_{aggr} + T_{tree} + T_{search} + T_{LPD} \\ &= \frac{N}{k} \cdot t_{tiling} + T_{aggr} + \frac{N'}{k} \cdot t_{tree}(\log \frac{N'}{m}) \\ &+ \frac{N}{k} \cdot t_{search}(\log \frac{N'}{m}) + \frac{N}{k} \cdot t_{LPD} \end{aligned} \tag{2}$$

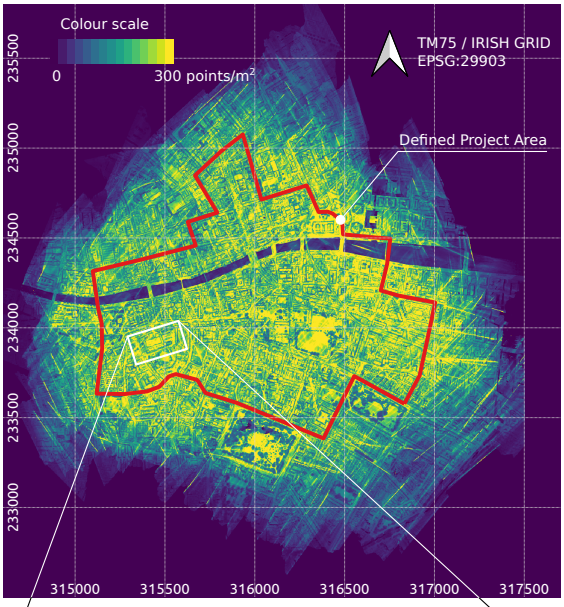
Assuming that the differences between N and N', as well as $\log(N)$ and $\log(\frac{N}{m})$ are negligible, the last three terms of $T_{parallel}$ approximate T_{serial}/k. The first two terms are the overhead required to parallelise the computation. $T_{parallel}$ can be rewritten as in Equation 3. The speedup factor, $\frac{T_{serial}}{T_{parallel}}$ (Pacheco, 2011), is always smaller than k because $T_{overhead}$ is always larger than zero. The efficiency of the algorithm is measured by how close its speedup factor is with respect to to k.

$$\begin{aligned} T_{parallel} &\approx T_{overhead} + \frac{T_{serial}}{k}, \\ \text{where } T_{overhead} &= T_{tiling} + T_{aggr} \end{aligned} \tag{3}$$

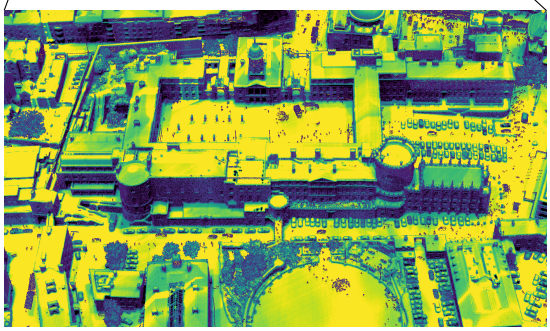
4. RESULTS

The algorithm introduced in Section 3 was used to compute the 3D local point density index of a 1.4 billion point data set acquired over a part of the city centre of Dublin, Ireland in 2015

by Laefer et al. (2017). The LPD index results of the entire data set are shown in Figure 3a. The red polygon indicates the Defined Project Area (DPA), in which the minimum point density was established as part of the project specifications. The entire coverage, including the partially covered area, is called the Buffered Project Area (BPA). The DPA and BPA are established concepts defined in the US Geological Survey's LiDAR Base Specification (Heidemann, 2018). Figure 3b presents a close-up, 3D view of a small region around Dublin Castle. As expected, the 3D LPD index computation automatically adapts to the 3D surface orientations and captures the point density on various surfaces such as the ground, building roofs, and facades. Points reflected from trees and transient objects such as moving people appear as low density points in dark blue colours. This is an inherent feature of the 3D LPD approach, which is unavailable in 2D approaches.



(a) Top view of the entire 2015 Dublin point cloud



(b) Close-up view of Dublin Castle area

Figure 3. Local point density index result; the areas in yellow exceed 300 points/m^2; the dark blue on the building facade is typically 30 points/m^2

The blue and dotted, orange histograms in Figure 4 represents the point density distributions in the BPA and the DPA, respectively. The proportion of lower density points in the BPA is higher than that in the DPA because the BPA includes partially covered areas. Each histogram has two peaks. The higher peak (around 280 points/m^2) represents the typical point density on horizontal surfaces such as the ground and building rooftops. Notably, the density value resulted from the 3D LPD approach

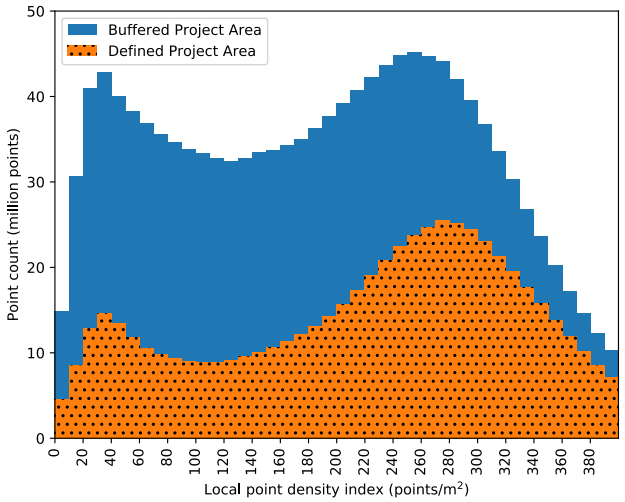


Figure 4. Histogram of local point density index

is always lower than those computed by conventional 2D approaches because the area of a 3D inclined surface is always larger than the corresponding 2D projection of the area on the ground. The majority of points contributing to the second peak (around 30 points/m^2) are those captured on building facades and other vertical surfaces. The density histogram is an effective means to provide insight into the detailed point density distribution. For example, the River Liffey is clearly visible in Figure 3a as points captured on water surfaces are sparse due to specular reflection and refraction. Several flat grassy areas also clearly stand out in yellow due to their high point density. In Figure 3b, the vast density difference between the roof and facade surfaces of Dublin Castle is obvious. The accuracy of the LPD computation can be straightforwardly evaluated using sampled hand calculations or reference solutions from software such as CloudCompare (Girardeau-Montaut et al., 2005). The remainder of this section focuses on evaluating the computational efficiency of the proposed parallel algorithm.

To evaluate the performance of the proposed algorithm, the runtime of the LPD index computation was measured with various numbers of processors. Notably, the concept of processors in Section 3 corresponds to two different concepts, executors and cores, in the implementation of Apache Spark. An executor is a software process residing on a physical node of the cluster and does not share the memory with other executors. An executor can execute multiple tasks using multi-threading. The maximum number of parallel tasks per executor is referred to as the number of cores. Importantly, the concept of core in Spark that is used in this paper does not refer to the physical CPU cores. The experiments were conducted using two different computing clusters. The specifications of the clusters are shown in Table 1. The first cluster from New York University (NYU) was an 18-node, high-end cluster with a high CPU core count (2×24 cores) and a large memory (384 GB). The second cluster from University of Genoa (UniGe) was a cluster of 8 commodity (consumer-grade) nodes, each had 1×6 CPU cores and 16 GB of memory. The purpose of experimenting with the two types of clusters is to understand any limitation of commodity clusters. As the proposed algorithms rely on the spatial structure of the data (i.e. tiles, buffers), the initial arrangement of the input data affects the computational performance. In all experiments reported in this paper, the input point clouds were structured as 500 × 500 tiles, and the points within each tile

This contribution has been peer-reviewed. The double-blind peer-review was conducted on the basis of the full paper.
https://doi.org/10.5194/isprs-annals-VIII-4-W2-2021-75-2021 | © Author(s) 2021. CC BY 4.0 License.

were ordered by their timestamps as they were delivered from the flight vendor (Laefer et al., 2017). No restructuring was performed prior to the density computation.

	NYU Cluster	UniGe Cluster
Type	High-end	Commodity
Nodes	18	8
CPU	2×24 cores Intel Xeon @2.90 GHz	1×6 cores Intel Core i5 @3 GHz
RAM	384 GB	16 GB

Table 1. Cluster specifications

All experiments were performed with $r = 0.5m$ and $\delta_x = \delta_y = 5m$. The selection of the radius parameter, r, is dependent on the data characteristics and target applications. The tile size parameters, δ_x and δ_y, do not affect the density results and depend on the amount of memory allocated to the cores. A large tile size would overload the core memory, while a small tile size would increase the data redundancy due to buffering, increase communication costs, and, thus, increase the computing time. The experiments' runtimes were measured, as reported in Table 2 and Figure 5. The reported runtimes include the costs for data input and output from the Hadoop Distributed File System, in which the data resided. The runtimes were inversely proportional to the number of executors in all experiments. The only exception was the cessation of runtime reduction when increasing the number of executors from 8 to 16 in the UniGe cluster. As the UniGe cluster had 8 compute nodes, requesting more than 8 executors did not yield a further speedup. Another aspect related to the capacity of the UniGe cluster is that the cluster was configured to allow a maximum of 4 cores per executor, while up to 8 cores per executor could be requested from the NYU cluster. Nevertheless, when the clusters were not pushed beyond their capacity, $T_{parallel}$ reduced as k increased. The observed relationship between $T_{parallel}$ and k demonstrated that the unknown cost of data aggregation (T_{aggr} in Equation 2) did not impede the efficiency of the proposed algorithm. With respect to the relative performance of the two clusters, the smaller, commodity UniGe cluster was surprisingly faster than the NYU cluster consistently in all of the tests, except for those beyond the cluster's capacity. This is probably attributable to the fact that during the time of the testing, the NYU cluster conducted over 75 other concurrent jobs. In contrast, the only job running in the UniGe cluster was the point density computation. Thus, resource sharing (e.g. disks, network bandwidth) is a likely factor contributing to the lower computational speed of the NYU cluster.

Cores	Cluster	Number of executors				
		1	2	4	8	16
1	NYU	21317	11227	5738	2882	1445
	UniGe	15326	7584	3935	2079	2090
2	NYU	12001	6024	2812	1474	775
	UniGe	7766	4361	2190	1171	1170
4	NYU	5836	2979	1498	831	462
	UniGe	5481	2555	1408	743	758
8	NYU	3125	1584	929	562	308
	UniGe	-	-	-	-	-

Table 2. Runtime (in seconds)

The efficiency of the proposed algorithm was further analysed by computing the speedup factors, $S = T_{serial}/T_{parallel}$. The factor represents how much faster the parallel computation is with respect to the serial computation. In this research, an actual serial algorithm as described at the beginning of Section 3 was not implemented. Instead, a pseudo value of T_{serial} was

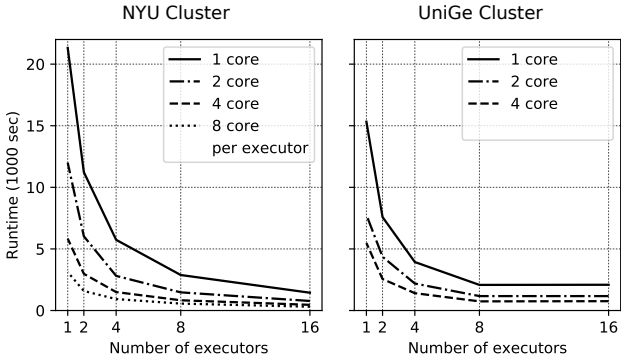


Figure 5. Runtime

used by running the parallel computation with 1 executor × 1 core. The speedup factors with various numbers of executors and cores are reported in Table 3 and Figure 6. Given 1 core per executor (i.e. the solid lines in Figure 6), the speedup was nearly linear. Namely, from the NYU cluster experiments, the speedup factor was 14.8 when the number of executors increased by 16 times. The efficiency factor, in this case, was $E = S/k = 14.8/16 = 0.93$. As $E = 0.93 \approx 1$ indicates that the speedup of the parallel algorithm was nearly ideal at 16 executors × 1 core/executor. If the efficiency factor equals one, the computation is said to have a linear speedup. A linear speedup is usually very difficult to achieve in practice (Pacheco, 2011). As the number of cores per executor (hence, the total number of cores) increased, the efficiency reduced. The reduction in efficiency is seen in Figure 6a as the dotted and the dashed lines, which correspond to the experiments with 8 and 4 cores/executor. With 16 executors × 8 cores/executor, the efficiency was $E = 69.2/(16 \times 8) = 0.54$. While the reduction in computational time by almost 70 times was great, the speedup was less significant, compared to the increase in the computing capacity of $16 \times 8 = 128$ times. The reduction in efficiency in relation to the increase in the number of processors (or cores) is well known and appears in almost every parallel program. The reason is that the presence of more processors translates to additional data transmission across the network. In addition, having more processors usually incurs a higher overhead for coordinating and synchronising the processors.

Cores	Cluster	Number of executors				
		1	2	4	8	16
1	NYU	1.0	1.9	3.7	7.4	14.8
	UniGe	1.0	2.0	3.9	7.4	7.3
2	NYU	1.8	3.5	7.6	14.5	27.5
	UniGe	2.0	3.4	7.0	13.1	13.0
4	NYU	3.7	7.2	14.2	25.7	46.1
	UniGe	2.8	6.0	10.9	20.6	20.2
8	NYU	6.8	13.5	22.9	37.9	69.2
	UniGe	-	-	-	-	-

Table 3. Speedup

With respect to the experiments conducted using the UniGe cluster, the speedup was almost linear in all cases when the cluster capacity was not exceeded. In the experiments that allocated 1 or 2 cores per executor, the speedup factors obtained from the UniGe cluster were similar to those from the NYU cluster. When the maximum of 4 cores were allocated per executor, the UniGe cluster's speedup was lower compared to that of the NYU cluster. For example, given 8 executors × 4 cores, the speedup factor in the NYU cluster was 25.7, while the UniGe cluster's factor was 20.6.

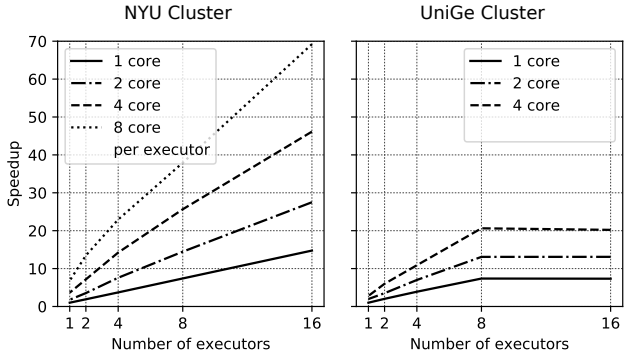


Figure 6. Speedup

5. CONCLUSIONS

This paper presents an algorithm for computing the 3D local point density index for point clouds. The algorithm consists of two stages. In the first stage, points are aggregated into buffered tiles, and the computation is parallel at the point level. In the second stage, the tiles resulting from stage 1, are received and the LPD index is computed in parallel at the tile level. The algorithm is particularly suitable for processing large point clouds as it supports out-of-core computation and provides a high level of parallelism. An implementation of the algorithm was introduced using Apache Spark, a general-purpose distributed computing platform. The implementation can employ multiple nodes and cores of a distributed memory computing cluster to perform the LPD computation rapidly. Computing the LPD index for a data set of 1.4 billion points required just over 5 minutes with 128 cores (16 executors $\times$ 8 cores/executor). The efficiency of the algorithm was rigorously evaluated through two sets of experiments conducted using both a high-end cluster and a commodity cluster. While the larger, high-end cluster had a greater capacity, it was consistently, slightly slower than the smaller, commodity cluster, because the resources of the high-end cluster were shared with many other concurrent computations. In contrast, the commodity cluster was not shared during the experiments.

The efficiency of the parallel algorithm was successfully demonstrated in both sets of experiments. An efficiency factor of 14.8/16 was achieved when 16 executors $\times$ 1 core/executor were used. The efficiency reduced to 69.2/128 with 16 executors $\times$ 8 cores. The 3D LPD index computed by the algorithm is an effective means to evaluate and subsequently visualise the 3D distribution of points in a point cloud. The LPD index allows discerning the point density on surfaces that have different orientations such as building facades, roofs, and ground surfaces. While the algorithm was demonstrated using an aerial laser scanning point cloud, the algorithm is applicable to any kind of point cloud, including aerial, terrestrial, and mobile laser scanning data, as well as point clouds derived from photogrammetry data. In addition, the proposed algorithm could be straightforwardly extended to compute other local geometric features of a point cloud such as normal vector, surface roughness, and curvature. Future research should investigate those possibilities.

ACKNOWLEDGEMENTS

This publication has emanated from research supported in part by a grant from Science Foundation Ireland under Grant number SFI - 17US3450. For the purpose of Open Access, the author has applied a CC BY public copyright licence to any Author Accepted Manuscript version arising from this submission. Further funding for this project was provided by the National Science Foundation as part of the project "UrbanARK: Assessment, Risk Management, & Knowledge for Coastal Flood Risk Management in Urban Areas" NSF Award 1826134, jointly funded with Science Foundation Ireland and Northern Ireland Trust (Grant USI 137). The clusters used for the testing were provided by NYU High Performance Computing Center and University of Genoa. The authors thanks the NYU HPC staff and Mr Federico Dassereto for the excellent technical support. The aerial LiDAR data of Dublin were acquired with funding from the European Research Council [ERC-2012-StG-307836] and additional funding from Science Foundation Ireland [12/ERC/I2534].

References

AHN, 2014. Actueel Hoogtebestand Nederland - Actualisatie Van Het 2. https://www.ahn.nl/ (Last accessed 30 July 2021).

Aljumaily, H., Laefer, D., Cuadra, D., 2016. Big-data approach for three-dimensional building extraction from aerial laser scanning. *Journal of Computing in Civil Engineering*, 30(3), 04015049.

Bethel, M., 2019. Airborne LiDAR point density, more to the point. https://www.slideshare.net/MattBethel/airborne-lidar-point-density.

Bodenstein, C., Gotz, M., Riedel, M., 2015. Analysis of 3D point clouds using a parallel DBSCAN clustering algorithm. *Innovatives Supercomputing in Deutschland*, 3, 33–35.

Brédif, M., Vallet, B., Ferrand, B., 2015. Distributed dimensionality-based rendering of LiDAR point clouds. *International Archives of the Photogrammetry, Remote Sensing & Spatial Information Sciences*, XL-3/W3, 559–564.

Che, E., Olsen, M. J., 2018. Multi-scan segmentation of terrestrial laser scanning data based on normal variation analysis. *ISPRS Journal of Photogrammetry and Remote Sensing*, 143, 233–248.

Dean, J., Ghemawat, S., 2008. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), 107-113.

Girardeau-Montaut, D., Roux, M., Marc, R., Thibault, G., 2005. Change detection on points cloud data acquired with a ground laser scanner. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 36(3), W19.

Guttman, A., 1984. R-trees: a dynamic index structure for spatial searching. *ACM SIGMOD*, Association for Computing Machinery, Boston, 47–57.

Heidemann, 2018. Lidar base specification (ver. 1.3). *U.S. Geological Survey standards - Collection and delineation of spatial data*, U.S. Geological Survey. https://pubs.er.usgs.gov/publication/tm11B4 (Last accessed by 7 April 2020).

Julin, A., Jaalama, K., Virtanen, J.-P., Pouke, M., Ylipulli, J., Vaaja, M., Hyyppä, J., Hyyppä, H., 2018. Characterizing 3D city modeling projects: Towards a harmonized interoperable system. *ISPRS International Journal of Geo-Information*, 7(2), 55.

Kleppmann, M., 2017. Reliable, scalable, and maintainable applications. *Designing data-intensive applications - The big ideas behind reliable, scalable, and maintainable systems*, O'Reilly Media, 3–22.

Krishnan, S., Baru, C., Crosby, C., 2010. Evaluation of MapReduce for gridding LiDAR data. *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, IEEE, 33–40.

Laefer, D., Abuwarda, S., Vo, A., Truong-Hong, L., Gharibi, H., 2017. 2015 Aerial Laser and Photogrammetry Survey of Dublin City Collection Record. https://doi.org/10.17609/N8MQ0N/ (Last accessed 30 July 2021).

Laefer, D., O'Sullivan, C., Carr, H., Truong-Hong, L., 2014. Aerial laser scanning (ALS) data collected over an area of around 1 square km in Dublin city in 2007.

Lari, Z., Habib, A., 2012. Alternative methodologies for the estimation of local point density index: Moving towards adaptive LiDAR data processing. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 39, B3.

Li, Z., Hodgson, M., Li, W., 2017. A general-purpose framework for parallel processing of large-scale LiDAR data. *International Journal of Digital Earth*, 11(1), 26–47.

Liu, K., Boehm, J., Alis, C., 2016. Change detection of mobile lidar data using cloud computing. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences-ISPRS Archives*, 41, International Society of Photogrammetry and Remote Sensing (ISPRS), 309–313.

Martinez, J. L., Reina, A. J., Morales, J., Mandow, A., García-Cerezo, A. J., 2013. Using multicore processors to parallelize 3d point cloud registration with the coarse binary cubes method. *2013 IEEE International Conference on Mechatronics (ICM)*, IEEE, 335–340.

Naus, T., 2010. LiDAR density and spacing specification v1.0. https://www.asprs.org/wp-content/uploads/2010/12/Naus.pdf.

Pacheco, P., 2011. Parallel hardware and parallel software. *An introduction to parallel programming*, Morgan Kaufmann, chapter P2, 15–77.

Renslow, M., 2012. *Manual of airborne topographic lidar*. American Society for Photogrammetry and Remote Sensing.

Rizki, P., Eum, J., Lee, H., Oh, S., 2017. Spark-based in-memory DEM creation from 3D LiDAR point clouds. *Remote Sensing Letters*, 8(4), 360–369.

Saglam, A., Makineci, H. B., Baykan, N. A., Baykan, Ö. K., 2020. Boundary constrained voxel segmentation for 3D point clouds using local geometric differences. *Expert Systems with Applications*, 157, 113439.

Samet, H., 2006. *Foundations of multidimensional and metric data structures*. 1^{st} edn, Morgan Kaufmann, San Francisco, CA, USA.

Stanley, M. H., Laefer, D. F., 2021. Metrics for aerial, urban lidar point clouds. *ISPRS Journal of Photogrammetry and Remote Sensing*, 175, 268–281.

van Oosterom, P., Martinez-Rubi, O., Ivanova, M., Horhammer, M., Geringer, D., Ravada, S., Tijssen, T., Kodde, M., Gonçalves, R., 2015. Massive point cloud data management: Design, implementation and execution of a point cloud benchmark. *Computers & Graphics*, 49, 92–125.

Vo, A., Laefer, D., 2019. A Big Data approach for comprehensive urban shadow analysis from airborne laser scanning point clouds. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, IV-4/W8, 111–116.

Vo, A., Laefer, D., Bertolotto, M., 2016. Airborne laser scanning data storage and indexing: State of the art review. *International Journal of Remote Sensing*, 37(24), 6187–6204.

Vo, A., Laefer, D., Smolic, A., Zolanvari, S., 2019. Per-point processing for detailed urban solar estimation with aerial laser scanning and distributed computing. *ISPRS Journal of Photogrammetry and Remote Sensing*, 155(June), 119–135.

Vo, A., Truong-Hong, L., Laefer, D., Bertolotto, M., 2015. Octree-based region growing for point cloud segmentation. *ISPRS Journal of Photogrammetry and Remote Sensing*, 104, 88–100.

Wu, H., Guan, X., Gong, J., 2011. ParaStream: A parallel streaming Delaunay triangulation algorithm for LiDAR points on multicore architectures. *Computers & Geosciences*, 37(9), 1355–1363.

Xu, Y., Yao, W., Tuttas, S., Hoegner, L., Stilla, U., 2018. Unsupervised segmentation of point clouds from buildings using hierarchical clustering based on gestalt principles. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 11(11), 4270–4286.

Ye, Z., Xu, Y., Huang, R., Tong, X., Li, X., Liu, X., Luan, K., Hoegner, L., Stilla, U., 2020. LASDU: A Large-Scale Aerial LiDAR Dataset for Semantic Labeling in Dense Urban Areas. *ISPRS International Journal of Geo-Information*, 9(7), 450.

Zecevic, P., Bonaci, M., 2017. *Spark in action*. Manning.

Zhang, J., Wu, G., Hu, X., Li, S., Hao, S., 2011. A parallel k-means clustering algorithm with mpi. *2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming*, IEEE, 60–64.

Zlatanova, S., Nourian, P., Gonçalves, R., Vo, A., 2016. Towards 3D raster GIS: On developing a raster engine for spatial DBMS. *ISPRS WG IV/2 Workshop Global Geospatial Information and High Resolution Global Land Cover/Land Use Mapping*, 45–60.