Parallel 3D Point Cloud Segmentation

Alessio Piroli

18/08/2025

1 Abstract

Sensors play a very important role in detecting and perceiving the environment. LiDAR sensors are often used in robotics and autonomous driving applications. They provide the underlying systems with three dimensional point clouds which can be processed and used to safely and reliably navigate the environment. The aim of this project is to implement and parallelize the algorithm described in the paper *Fast Segmentation of 3D Point Clouds for Ground Vehicles, Himmelsbach et al. (2010)* [1], which focuses on the segmentation of ground points in a LiDAR point cloud. The algorithm is first implemented sequentially in the C++ programming language and then parallelized by exploiting it's inherent parallelism. In contrast to existing implementations of the method, this project utilizes the OpenMP application programming interface (API) [2] for parallelizing the approach. Experiments were conducted to evaluate the effectiveness of the parallelization, demonstrating performance improvements. The rest of the report is structured as follows: in Sect. 2 a description of the problem is provided and other implementations are discussed, in Sect. 3 a description of the algorithm and its analysis are given, in Sect. 4 the methods used during the parallelization approach are illustrated, in Sect. 5 experiments are described and results are presented, and finally in Sect. 6 the conclusion and future works are reported.

2 Problem Description

The problem addressed in the paper is that of accurately and efficiently segmenting 3D point clouds recorded from LiDAR sensors. The method focuses on distinguishing between "ground" and "non-ground" points to facilitate the subsequent classification of objects. Parallelizing the algorithm while preserving speed and robustness is crucial as autonomous systems need to process large amounts of data in real time to guide decision making.

Various techniques have been proposed in the literature for the segmentation of LiDAR point clouds. Early approaches used intermediate representations like 2.5D points to segment "ground" points [3]. These approaches could achieve real-time performances, however, their performance suffered from under-segmentation. In contrast, the implemented method [1], works directly on the 3D point cloud. This has the benefit of achieving higher segmentation performance while maintaining real-time run capabilities. Although modern approaches like [4] use Deep Neural Networks for LiDAR point clouds segmentation (achieving higher accuracy compared to geometric approaches), the implemented paper remains relevant for applications with hardware restrictions (i.e., no GPU available).

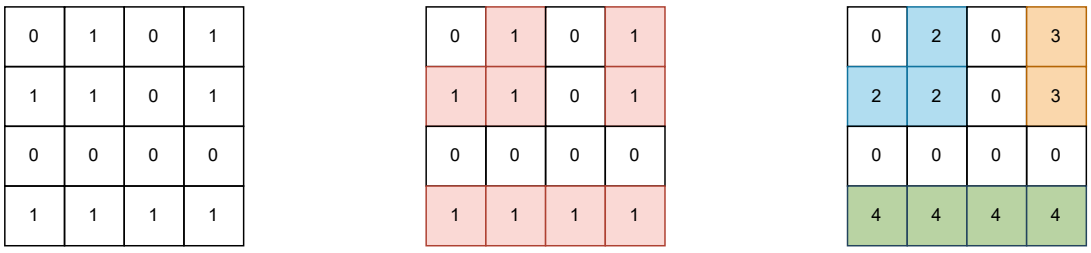
Other implementations of the selected paper exist:

- A python implementation of the method [5].
- A C++ implementation which allows for multi-threading through the standard library (`std::thread`) [6].
- A C++ implementation [7] which builds upon the previous one [6].

Figure 1: Illustration of the five stages of the algorithm.

3 Algorithm Description

The paper presents an algorithm for fast segmentation of long-range unordered 3D points cloud data. It achieves real-time performance and is less prone to under-segmentation when compared to previous methods. The proposed approach mainly aims to classify points into "ground" and "non-ground". As shown in Figure 1, the algorithm can be split in five main stages. In the first stage, as illustrated in Figure 3a, the xy-plane is represented as a circle and divided into a number S of segments. Then, all points belonging to a segment are mapped to one of the B bins in which a segment is divided, as shown in Figure 3b. Next, prototype points are defined as those with the lowest z coordinate in a bin. All prototype points belonging to the same segment are then grouped and sorted in ascending distance from the origin (where the LiDAR sensor is positioned). Afterwards, the incremental algorithm described in the paper is employed to extract lines from all sets of prototype points. These are used to determine what the ground should correspond to at a certain distance from the center. When lines are extracted they can be discarded if they are too steep (slope threshold) or if their starting height (y intercept) is excessive. This is done to prevent elevated surfaces such as car roofs or side walks to be classified as ground points. In the ground classification process, each point is matched with the closest line and its expected height is calculated. If the difference between the real and the computed height exceeds a threshold, the point is classified as "non-ground". If conditions are instead met, they are classified as "ground". The remaining points are then processed to identify clusters of "non-ground" points. To facilitate the the identification of clusters, the scene is divided in to a 2D grid. Cells that do not contain points are marked with a "zero" ID while non-empty ones are marked with a "one" ID. In the final step a Breadth-First Search algorithm (BFS) is applied to classify as single entities connected non-empty cells. A visual illustration of how this is accomplished is shown in Figure 2.



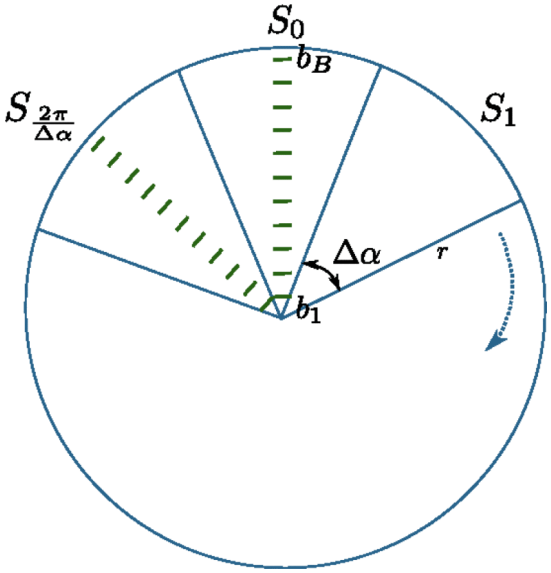
(a) Marking occupied and unoccupied cells with IDs one and zero.

(b) Keeping track of all connected occupied cells.

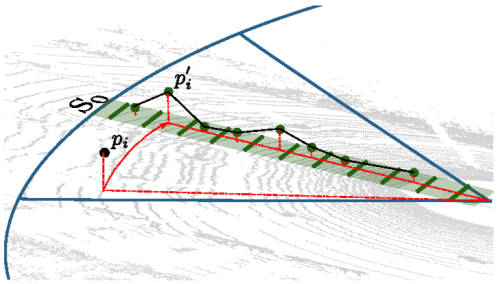
(c) Classification of all occupied and connected cells as a single object.

Figure 2: A visual representation of how connected occupied cells are classified as a single object.

As mentioned before, the algorithm can be split in five steps. For each of these parts, the time and work complexity are analyzed. Time complexity is defined as the amount of computer time it takes to run an algorithm while work complexity is defined as the number of operations



(a) Partitioning the 3D space into (small) segments of equal size. Taken from [1].



(b) Mapping of 3D points p to a bin of the corresponding segment and resulting mapped point p'. Taken from [1].

Figure 3: Partitioning of the 3D space into segments and mapping of points into bins. Taken from [1].

it performs. When assigning points to the respective bins and segments we must iterate through all of the N points and perform a fixed number of calculations for each of them. The same reasoning can be applied to determine the work complexity. Therefore, if we define N as the size of the LiDAR point cloud, time and work complexities are

$$T_{\text{seg\_bin\_sort,seq}}(N) = W_{\text{seg\_bin\_sort,seq}}(N) = \mathcal{O}(N). \tag{1}$$

When finding prototype points we iterate over all segments and associated bins to find points with the lowest z-coordinate. Then, prototype points are sorted in ascending distance from the origin. To find prototype points, all points in the scene must be analyzed. Let S denote the number of segments and B the number of bins. To sort the points, the `std::sort` function from the C++ standard library was used, which has a time complexity of $\mathcal{O}(n \log n)$, where n is the size of the sorted vector. This sorting is required for as many vectors as there are segments, S. Assuming a worst-case scenario where no bin is empty this section requires time and work complexities equal to

$$T_{\text{ass\_prot,seq}}(N, S, B) = W_{\text{ass\_prot,seq}}(n) = \mathcal{O}(N) + S \cdot \mathcal{O}(B \log B). \tag{2}$$

To create ground lines we iterate over all prototype points and fit lines through them. Assuming in the worst-case scenario that all prototype points make up for a single line, an increasing number of points will be tested. This number can be computed with the formula

$$\sum_{n=1}^{k} n = \frac{k \cdot (k+1)}{2} \approx \frac{k^2}{2} \tag{3}$$

where k is the number of prototype points. Therefore, the time and work complexity of fitting the lines is

$$T_{\text{line\_fit,seq}}(S, B) = W_{\text{line\_fit,seq}}(n) = \mathcal{O}(S \cdot B^2). \tag{4}$$

When classifying points as "ground" or "non-ground" all points in a segment are compared to the respective lines. Let us define with L the maximum number of lines belonging to a segment. In the worst case scenario, we compare the point to the last possible line belonging to a segment. This gives a time and work complexity of

$$T_{\text{ground\_class,seq}}(N, L) = W_{\text{ground\_class,seq}}(n) = \mathcal{O}(N \cdot L). \tag{5}$$

In the final stage, when segmenting "non-ground" points, we first iterate over all points to determine which need to be analyzed. Let N_{ng} define the number of "non-ground" points. Over all "non-ground" points we find the maximum x and y values to establish what the range of the "non-ground" scene is. As this range can not be known beforehand, let us define with N_{cells} the number of cells we divide the scene in. To classify cells as occupied or not, all "non-ground" points need to be considered. A standard Breadth-First Search algorithm is then applied to find all connected and occupied cells. This algorithm has a time complexity of $\mathcal{O}(V + E)$, with V the number of vertices and E the number of edges in a graph. In our case, $V = N_{\text{cells}}$ and $E = 4$ as each cell has a maximum of four connections. Therefore, the last step of the algorithm has a time and work complexity of

$$T_{\text{non\_ground\_class,seq}}(N) = W_{\text{non\_ground\_class,seq}}(N) = \mathcal{O}(N) + \mathcal{O}(N_{\text{ng}}) + \mathcal{O}(N_{\text{cells}}). \tag{6}$$

Many parallelization techniques exist in the literature. Among them are Tasks and Directed acyclic graphs, which consist of sets of tasks connected by directed dependency edges [8]. Looking at Figure 1, the previously described algorithm can be seen as a task graph, where the root task is assigning points to segments and bins and the final task is segmenting the remaining points. The graph must be computed sequentially as tasks present input and output dependencies. Conversely, each task can be made parallel by employing data decomposition, a technique where data can be partitioned and processed in parallel [9]. This is made possible by using OpenMP's API, which provides a solution for assigning to each thread a specific sub-domain of data to compute. The independence between data sub-domains makes the parallelization strategy effective as threads compute the assigned data concurrently. Nonetheless, the employment of synchronization mechanisms is still required to safely merge results and prevent data races.

To later evaluate the performance of the parallel implementation, absolute and relative speedups are evaluated. The absolute speedup of a parallel algorithm over the best known or possible sequential version for an input of size $\mathcal{O}(n)$ is the ratio of sequential to parallel runtime

$$S_p(n) = \frac{T_{\text{alg,seq}}(n)}{T^p_{\text{alg,par}}(n)}. \tag{7}$$

Relative speedup measures the performance improvements when increasing the number of threads. It is defined as the ratio between parallel runtime with one thread and parallel runtime with p threads [10]

$$S^p_{rel}(n) = \frac{T^1_{\text{alg,par}}(n)}{T^p_{\text{alg,par}}(n)}. \tag{8}$$

Let p define the number of threads employed in the parallelization method. When associating points to segments and bins, the data can be split and assigned. This is achieved by dividing the point cloud between threads, and each creating its own local data structure to store the results. As all local data structures are merged in the same container all points need to be copied. This gives a time complexity of

$$T^p_{\text{seg\_bin\_sort,par}}(n) = \mathcal{O}(N/p) + \mathcal{O}(N). \tag{9}$$

The theoretical speedup is computed as the ratio of the sequential and parallel complexities 1 and 9

$$S_{\text{seq\_bin\_sort,p}}(N) = \frac{\mathcal{O}(N)}{\mathcal{O}(N) + \mathcal{O}(N/p)}. \tag{10}$$

To find prototype points, we first scan the scene and then we sort all lists of independent points. As both operations can be split among threads, the resulting time complexity is

$$T^p_{\text{ass\_prot,par}}(N, S, B) = \mathcal{O}(N/p) + (S/p) \cdot \mathcal{O}(B \log B). \tag{11}$$

The theoretical speedup is then defined as

$$S_{\text{ass\_prot,p}}(N) = \frac{\mathcal{O}(N) + S \cdot \mathcal{O}(B \log B)}{\mathcal{O}(N/p) + (S/p) \cdot \mathcal{O}(B \log B)}. \tag{12}$$

In the Line Fitting function, line fitting must be performed S times. The procedure can be shared among threads, providing a time complexity of

$$T^p_{\text{line\_fit,par}}(S, B) = \mathcal{O}((S \cdot B^2)/p). \tag{13}$$

The theoretical speedup achieved is

$$S_{\text{line\_fit,p}}(S, B) = \frac{\mathcal{O}(SB^2)}{\mathcal{O}((SB^2)/p)} = p. \tag{14}$$

When classifying points as "ground" or "non-ground", we go through each point and we perform in the worst case L operations. Because the classification of each point is independent, the time complexity for this stage is

$$T^p_{\text{ground\_class,par}}(N, L) = \mathcal{O}((N \cdot L)/p). \tag{15}$$

The theoretical speedup achieved would then be

$$S_{\text{ground\_class,p}}(N, L) = \frac{\mathcal{O}(NL)}{\mathcal{O}((NL)/p)} = p. \tag{16}$$

The final step of classifying "non-ground" points can not be trivially parallelized, as a Breadth-First Search algorithm is implemented. Therefore, the step employed in the parallel algorithm is mostly identical to its sequential counterpart, except for four reduction operations to find the maximum and minimum values of the remaining scene and assigning increasing IDs to connected cells.

4 Implementation

The OpenMP API is a portable and scalable model that provides a flexible interface for developing parallel applications. It is a standardized way for writing concurrent programs for shared memory multiprocessors [2]. This makes it suitable for the parallelization of the algorithm through data decomposition.

A subset of all available features were used to parallelize the sequential algorithm. The `#pragma omp parallel` directive was used to create a team of threads. The code under this directive is executed in parallel by all threads. Mostly used was the `#pragma omp for` construct. This feature allows the splitting of a `for` loop among threads. It it mainly employed in the recognition of prototype points and line fitting. The `#pragma omp critical` directive is used to define blocks of code that only one thread at a time can execute. For instance, this was used to prevent race conditions when merging locally computed bins in to a single data structure. Essential was the `num_threads(integer)` clause, used to specify the number of threads to be used in a parallel region. This clause enabled performance testing with different numbers of threads.

Issues emerged during the parallel implementation. For example, when sorting points in the respective segments and bins, a simple parallelization of the for cycle can not be employed, as

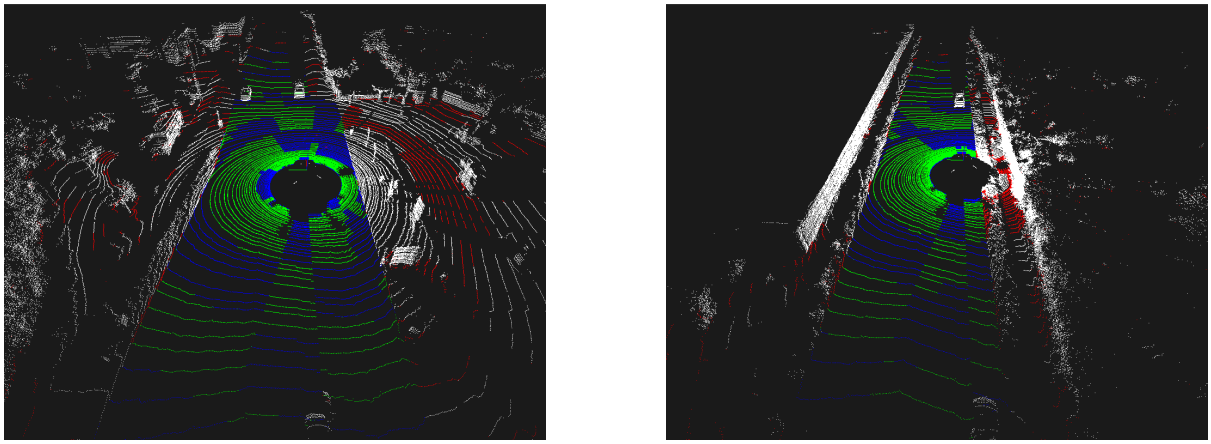


Figure 4: Qualitative results of the implemented ground segmentation algorithm. The figure shows a LiDAR point cloud in an urban environment. In white "non-ground" points are shown (true negatives). In green are shown correctly predicted "ground" points (true positives). In red are shown wrongly predicted "ground" points, which are ground points (false positive). In blue are shown are incorrectly classified "non-ground" points which are "ground" points (false negatives).

different threads might need to write in the same memory location. This phenomenon leads to race conditions which might result in data corruption or loss. To solve this, a team of threads is created with the `#pragma omp parallel` directive. For each thread, local storage is allocated and a portion of the scene is assigned. Once all threads complete their computation, a `#pragma omp critical` section is instantiated. By doing so, threads can singularly and therefore safely merge their local storage to the original data structure.

5 Benchmark and Scalability

To assess the effectiveness of the algorithm parallelization, three experiments were conducted. For each experiment, the overall execution and each sub-part is timed trough the C++ `chrono` library. This library is suitable for these experiment as it provides a flexible collection of types that track time with varying degrees of precision [11]. In the first experiment the algorithm is timed when different amounts of data are used. This is done to simulate performance when scenes are acquired using LiDAR sensors with different resolutions (e.g., sensors with 16, 64 or 128 channels). In the second experiment, the full scene size is maintained and the number of segments is varied. In the third experiment, similar to the previous one, the entire scene is used and the number of bins is varied. The last two experiments are used to evaluate the performance of parallelized functions whose performance is directly dependent on these two parameters. For all experiments, the remaining parameters are fixed at the values yielding the best mean intersection-over-union metric (mIoU), also known as the mean Jaccard Index, over the "ground" class [4]. The metric is defined as

$$\text{mIoU} = \frac{TP}{TP + FP + FN}, \tag{17}$$

where TP, FP and FN correspond to the number of true positive, false positive and false negative predictions.

In Figure 4 a qualitative evaluation of the segmentation performance of the implemented method is reported.

5.1 Experiment 1

In the first experiment different percentages (25%, 50%, 75%, 100%) of the entire data size are used to time the functions which performance is influenced by it. The functions affected are the sorting into segments and bin, the ground point classification and the complete function.

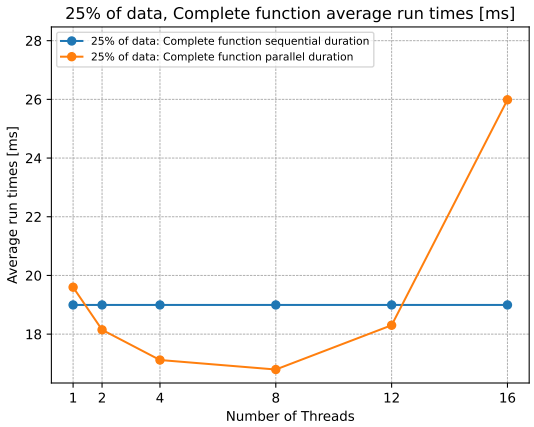
The results of the experiment are shown in Figures 5, 6, 7. As expected, for the complete function the absolute speedup increases with the number of threads employed. When the percentage of data used is 25%, 50% and 75% the best speed-up is achieved with 8 threads. When the full data is used, best performance is reached with 12 as the number of threads. For each data size, the best absolute speedups reached are respectively $1.13, 1.30, 1.42, 1.45$. Similar values are achieved for the relative speedups, with 1.17, 1.33, 1.39, 1.45. As the number of threads increases, overhead does as well, increasing in turn execution time. For the sorting into segments and bins functions a similar behavior is observed. For the first three percentages of data, best speedups are achieved with 4 and 8 threads. When the full data is used, 12 yields the best values. For each input size, the best absolute speedups reached are respectively 1.36, 1.73, 2.04, 2.09 while the relative speedups are 1.56, 1.97, 2.13, 2.24. For the ground point classification function, the best speedup is always achieved with a number of threads equal to 12. For each input size, the best absolute and relative speedups reached are respectively 6.87, 5.15, 5.84, 7.22 and 6.86, 5.26, 5.72, 7.02.

5.2 Experiment 2

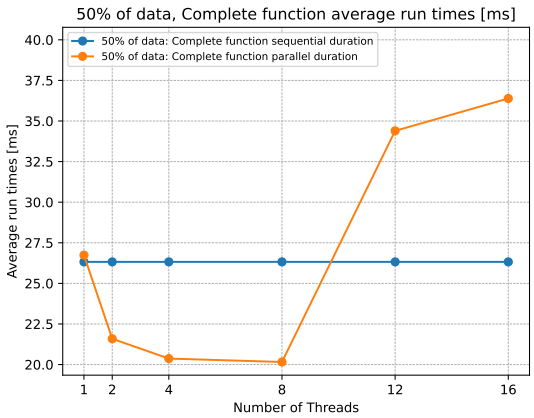
In the second experiment the full scene is used and the number of segments is varied. The functions affected are the assignment of prototype points and the line fitting function. Performance was valued for number of segments equal to 1, 10, 50 and 100, as shown in Figure 8 and 9. In the Assign Prototype Points when the number of segments is 1, the benefit of parallelism is outweighed by the overhead necessary to instantiate it. As can be seen from Figure 8a the parallel execution is always slower. When the number of segments is 10 and 50, similar trends are observed. Parallel execution times decrease with the number of threads but increases when the number reaches 16, becoming significantly slower. When the number of segments is 100 the parallel execution is always faster than the sequential one, independent on the number of threads. The absolute and relative speedups are respectively 0.97, 5.06, 5.92, 4.47 and 0.98, 5.91, 5.80, 4.40. In the Fit Lines function a general trend can be observed. Speedups greater than 1 are recorded as the number of threads increases from 1 to 12. However, execution times rise when 16 threads are used. Almost identical values are achieved for both the absolute and relative speedups, with 1.03, 2.38, 3.71, 3.10 and 1.10, 2.74, 3.70, 3.08 respectively.

5.3 Experiment 3

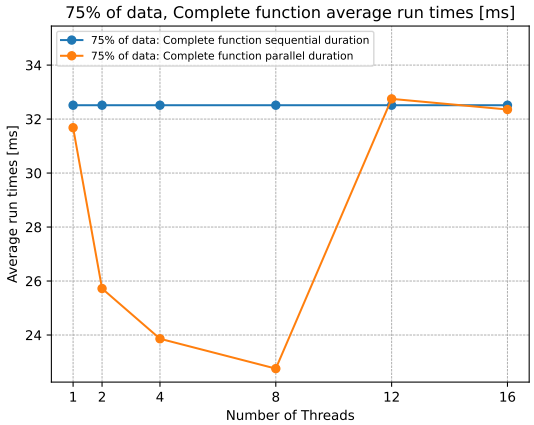
In the third experiment, as in the previous one, the full scene is used while the number of bins is varied. The functions affected are the complete function and the assignment of prototype points. Performance was evaluated for number of bins equal to 1, 10, 50 and 100, as shown in Figures 10 and 11. For the full function, when the number of bins is 1, 10 and 50, best performance is achieved with 8 as the number of threads. When the number of bins increases to 100 best performance is achieved with 4 as the number of threads. Values recorded for the best absolute and relative speedups are 1.22, 1.45, 1.62, 1.68 and 1.24, 1.33, 1.70, 1.73. For the Assign Prototype Points function a general trend is observed. For all bin values, best execution times are achieved with 12 as the number of threads. In contrast, 16 threads generally slows down execution, rendering it slower than the sequential counterpart. Values for the best absolute and relative speedups are 5.45, 4.47, 4.54, 4.14 and 5.56, 4.10, 4.67, 4.16.



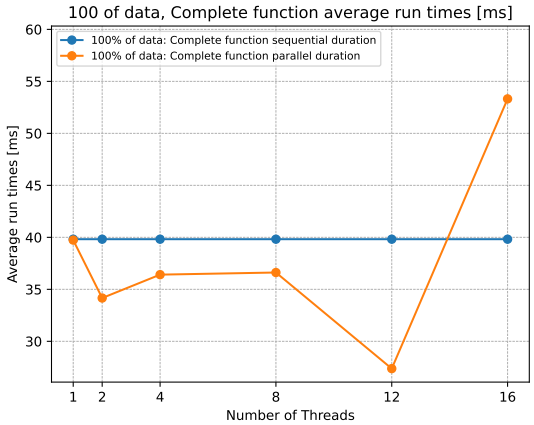
(a) Average run times [ms] of the entire function when using 25% of the entire data.



(b) Average run times [ms] of the entire function when using 50% of the entire data.



(c) Average run times [ms] of the entire function when using 75% of the entire data.



(d) Average run times [ms] of the entire function when using 100% of the entire data.

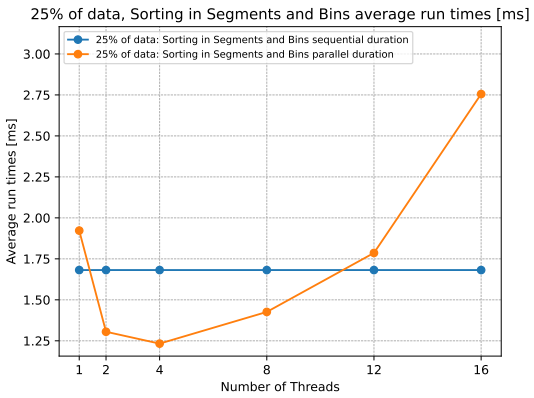
Figure 5: Experiment 1: average run times [ms] of the entire function when using 25%, 50%, 75% and 100% of the entire data.

5.4 Experiments Results

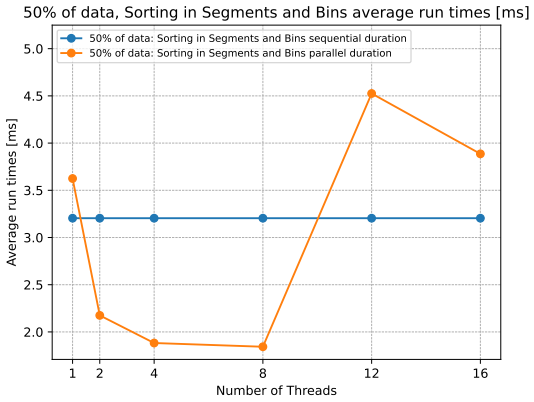
The parallelization techniques employed in the parallel algorithm resulted in speedups lower than those hypothesized. The difference between the theoretical and practical values is due to two main reasons. First, there is the intrinsic overhead associated with spawning a team of threads, which slows down execution. Second, point clouds are non-uniformly distributed. The scenes used for the purposes of this project were taken from the SemanticKITTI dataset [12]. They were recorded across various locations and involve substantially different scenarios. This translates in scenes that can be very different from one another, particularly concerning point cloud density. This characteristic further impacts how the work is split between threads, creating a load imbalance problem.

6 Conclusion and Future Work

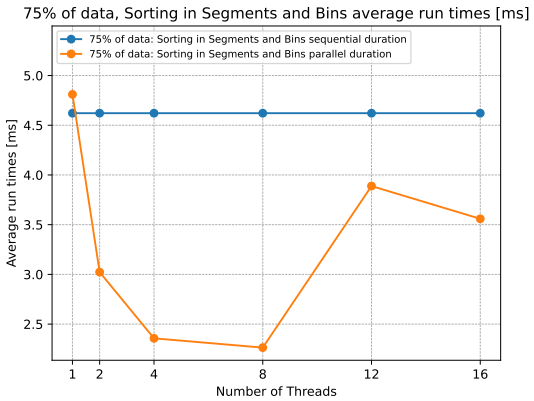
In this project the algorithm introduced in *Fast Segmentation of 3D Point Clouds for Ground Vehicles, Himmelsbach et al. (2010)* [1] to segment ground points in a LiDAR point cloud was implemented. The method was split in five parts and each of these was analyzed. A data decomposition technique was adopted to parallelize the sequential algorithm. Experiments



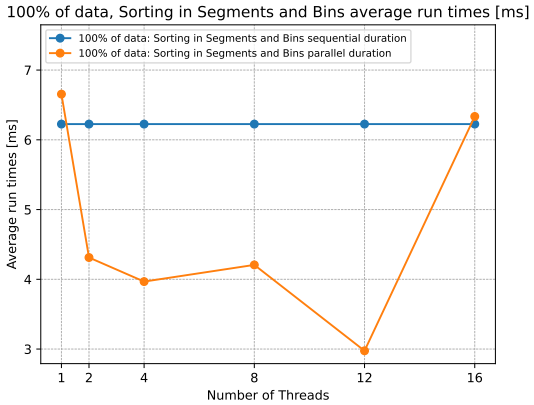
(a) Average run times [ms] of Sorting in Segments and Bins function when using 25% of the entire data.



(b) Average run times [ms] of Sorting in Segments and Bins function when using 50% of the entire data.



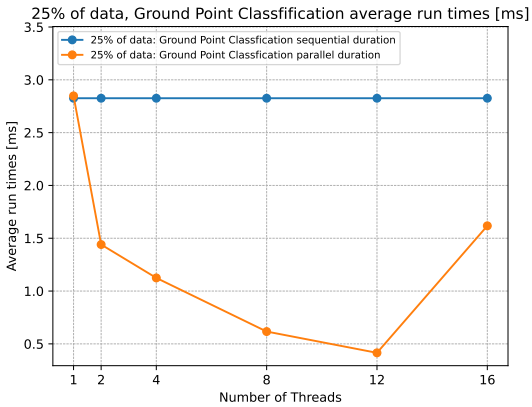
(c) Average run times [ms] of Sorting in Segments and Bins function when using 75% of the entire data.



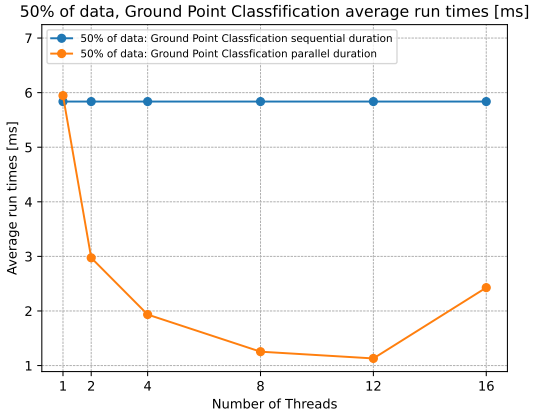
(d) Average run times [ms] of Sorting in Segments and Bins function when using 100% of the entire data.

Figure 6: Experiment 1: average run times [ms] of Sorting in Segments and Bins function when using 25%, 50%, 75% and 100% of the entire data.

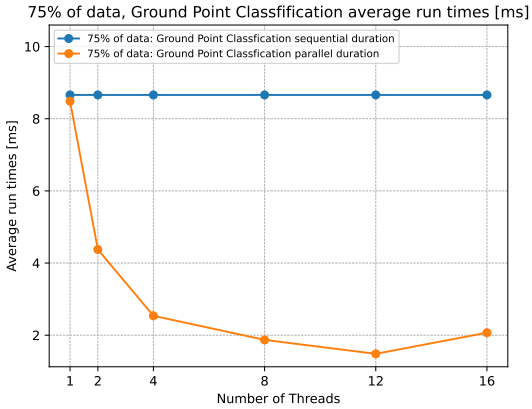
were then conducted to evaluate the effectiveness of the parallelization technique. Increases in performance were found but lower than theoretical predictions. Future work could explore different parallelization strategies to further optimize the workload distribution among threads.



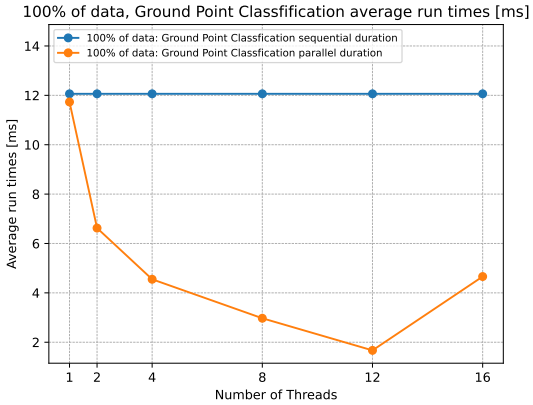
(a) Average run times [ms] of Ground Point Classification function when using 25% of the entire data.



(b) Average run times [ms] of Ground Point Classification function when using 50% of the entire data.

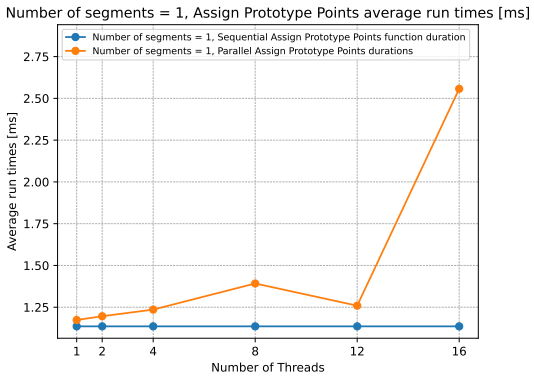


(c) Average run times [ms] of Ground Point Classification function when using 75% of the entire data.

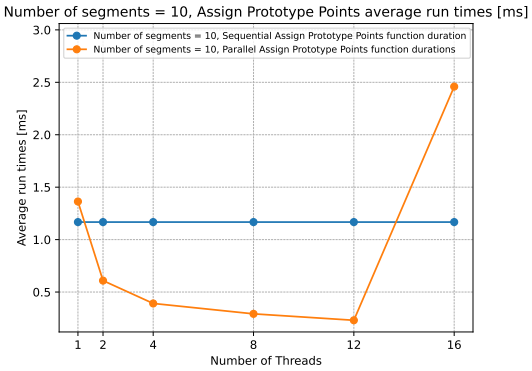


(d) Average run times [ms] of Ground Point Classification function when using 100% of the entire data.

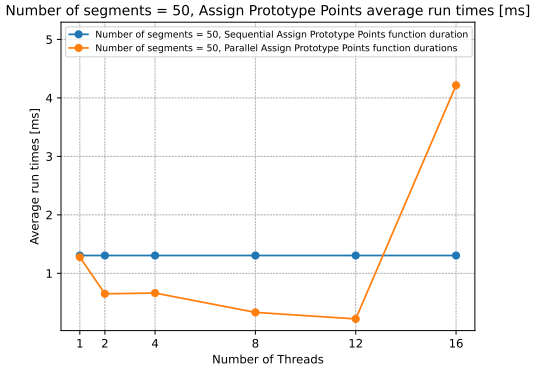
Figure 7: Experiment 1: average run times [ms] of Ground Point Classification function when using 25%, 50%, 75% and 100% of the entire data.



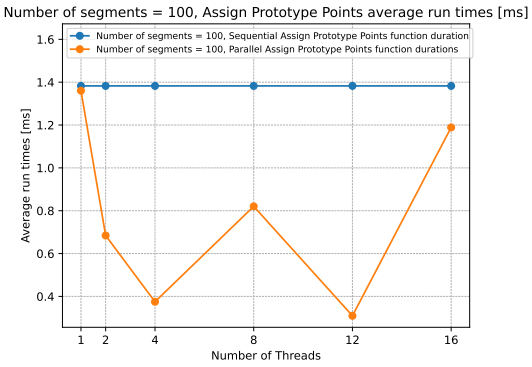
(a) Average run times [ms] of Assigning Prototype Points function when the number of segments is 1.



(b) Average run times [ms] of Assigning Prototype Points function when the number of segments is 10.

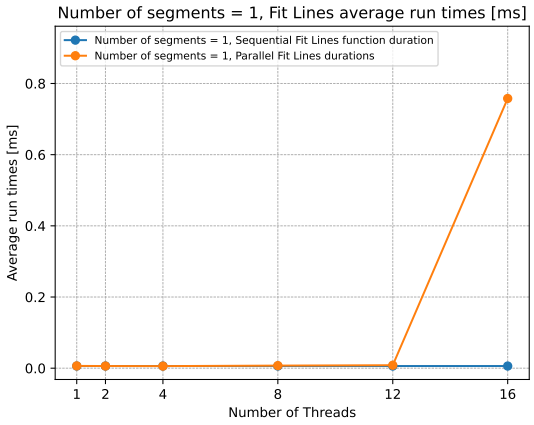


(c) Average run times [ms] of Assigning Prototype Points function when the number of segments is 50.

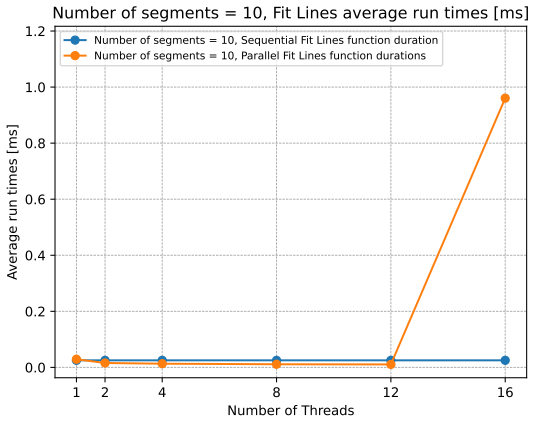


(d) Average run times [ms] of Assigning Prototype Points function when the number of segments is 100.

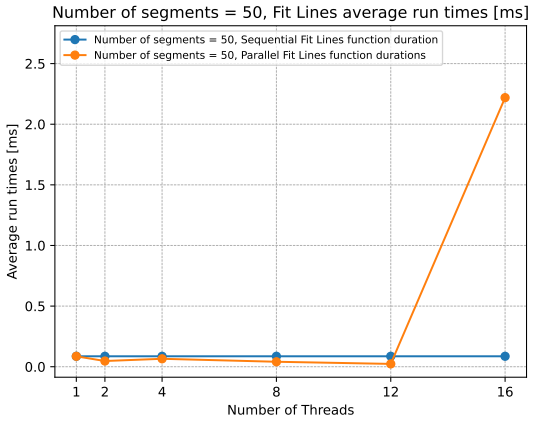
Figure 8: Average run times [ms] of Assigning Prototype Points function when the number of segments is 1, 10, 50 and 100.



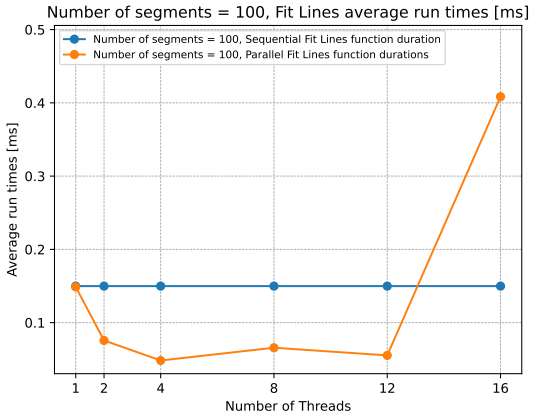
(a) Average run times [ms] of Line Fitting function when the number of segments is 1.



(b) Average run times [ms] of Line Fitting function when the number of segments is 10.

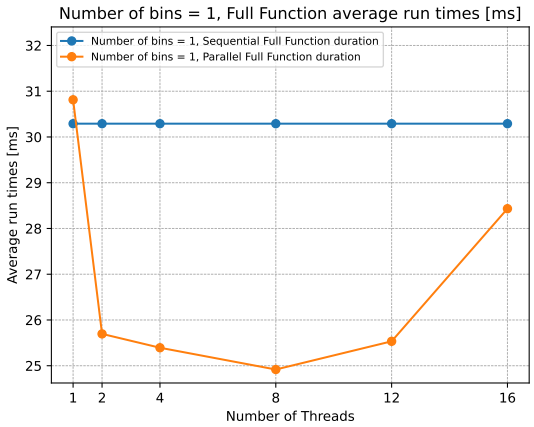


(c) Average run times [ms] of Line Fitting function when the number of segments is 50.

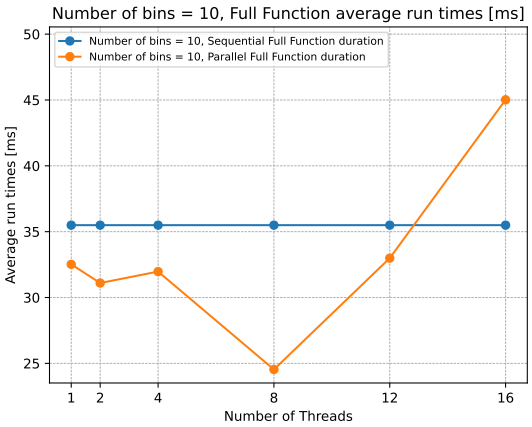


(d) Average run times [ms] of Line Fitting function when the number of segments is 100.

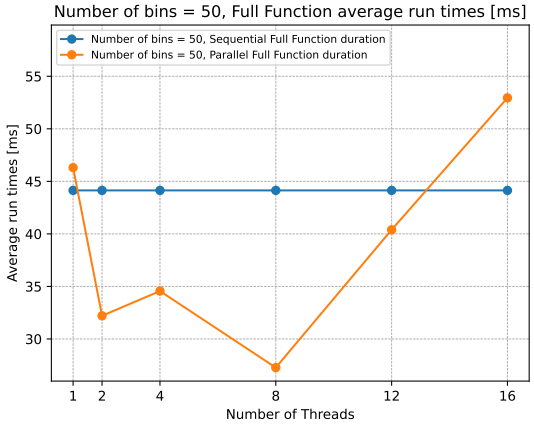
Figure 9: Average run times [ms] of Line Fitting function when the number of segments is 1, 10, 50 and 100.



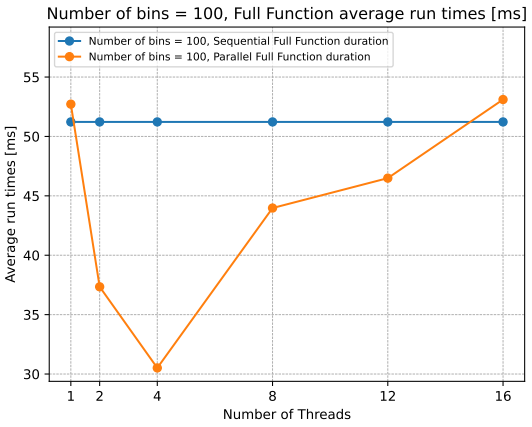
(a) Average run times [ms] of the Complete function when the number of bins is 1.



(b) Average run times [ms] of the Complete function when the number of bins is 10.

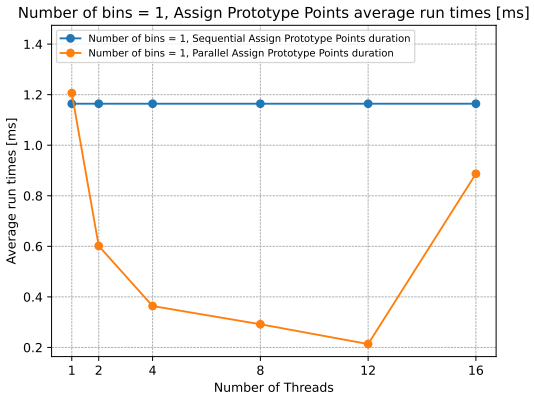


(c) Average run times [ms] of the Complete function when the number of bins is 50.

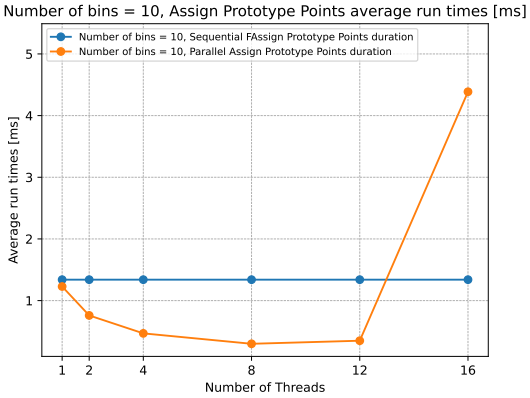


(d) Average run times [ms] of the Complete function when the number of bins is 100.

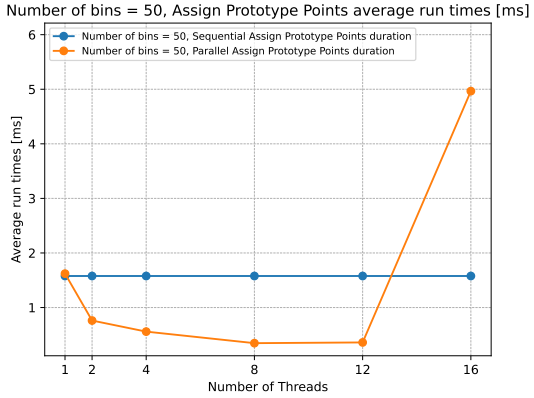
Figure 10: Average run times [ms] of the Complete function when the number of bins is 1, 10, 50 and 100.



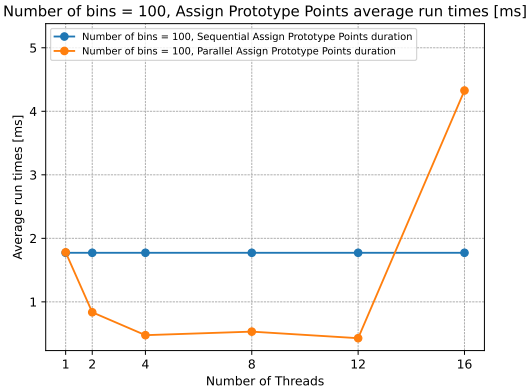
(a) Average run times [ms] of the Assign Prototype Points function when the number of bins is 1.



(b) Average run times [ms] of the Assign Prototype Points function when the number of bins is 10.



(c) Average run times [ms] of the Assign Prototype Points function when the number of bins is 50.



(d) Average run times [ms] of the Assign Prototype Points function when the number of bins is 100.

Figure 11: Average run times [ms] of the Assign Prototype Points function when the number of bins is 1, 10, 50 and 100.

References

[1] M. Himmelsbach, Felix v. Hundelshausen, and H.-J. Wuensche. "Fast segmentation of 3D point clouds for ground vehicles". In: *2010 IEEE Intelligent Vehicles Symposium*. ISSN: 1931-0587. June 2010, pp. 560–565. DOI: 10.1109/IVS.2010.5548059. URL: https://ieeexplore.ieee.org/document/5548059/ (visited on 08/02/2025).

[2] *Home - OpenMP*. URL: https://www.openmp.org/ (visited on 08/15/2025).

[3] M. Himmelsbach, T. Luettel, and H.-J. Wuensche. "Real-time object classification in 3D point clouds using point feature histograms". In: *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2009. URL: https://cir.nii.ac.jp/crid/1362262946077599744 (visited on 08/15/2025).

[4] Andres Milioto et al. "RangeNet ++: Fast and Accurate LiDAR Semantic Segmentation". en. In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Macau, China: IEEE, Nov. 2019, pp. 4213–4220. ISBN: 978-1-7281-4004-9. DOI: 10.1109/IROS40897.2019.8967762. URL: https://ieeexplore.ieee.org/document/8967762/ (visited on 08/17/2025).

[5] SilvesterHsu. *LiDAR_ground_removal*. https://github.com/SilvesterHsu/LiDAR_ground_removal. Accessed: 11-07-2025.

[6] lorenwel. *linefit_ground_segmentation*. https://github.com/lorenwel/linefit_ground_segmentation. Accessed: 11-07-2025.

[7] HuangCongQing. *linefit_ground_segmentation_details*. https://github.com/HuangCongQing/linefit_ground_segmentation_details. Accessed: 11-07-2025.

[8] Pratik Nayak. *Lecture 3: Strategies for designing parallel algorithms*. en. May 2025. URL: https://pratikvn.github.io/2025WS-Parallel-Computing-slides/Lecture3/lecture3.html#/tasks-and-dags (visited on 08/17/2025).

[9] Ananth Grama et al. "Principles of parallel algorithm design". In: *Introduction to Parallel Computing, 2nd ed. Addison Wesley, Harlow* (2003). URL: https://dsrajnor.wordpress.com/wp-content/uploads/2023/02/hpc-unit-ii-ppts.pdf (visited on 08/16/2025).

[10] Pratik Nayak. *Lecture 2: Metrics in parallel computing*. en. May 2025. URL: https://pratikvn.github.io/2025WS-Parallel-Computing-slides/Lecture2/lecture2.html#/speedup (visited on 08/17/2025).

[11] *Date and time library - cppreference.com*. URL: https://en.cppreference.com/w/cpp/chrono.html (visited on 08/17/2025).

[12] Jens Behley et al. "Towards 3D LiDAR-based semantic scene understanding of 3D point cloud sequences: The SemanticKITTI Dataset". en. In: *The International Journal of Robotics Research* 40.8-9 (Aug. 2021), pp. 959–967. ISSN: 0278-3649, 1741-3176. DOI: 10.1177/02783649211006735. URL: https://journals.sagepub.com/doi/10.1177/02783649211006735 (visited on 08/17/2025).