Multithreading in Laser Scanning Data Processing

Vladimir Badenko[1(✉)], Serafim Tammsaar[1], Kirill Beliaevskii[1], Alexander Fedotov[1], and Konstantin Vinogradov[2]

[1] Peter the Great St. Petersburg Polytechnic University, Polytechnicheskaya 29, 195251 St. Petersburg, Russia
vbadenko@gmail.com, {serafim.tammsaar, kirill.beliaevskii}@spbpu.com, afedotov@spbstu.ru
[2] Saint-Petersburg State University, Universitetskaya Emb. 13B, 199034 St. Petersburg, Russia
kostyal495@mail.ru

Abstract. Laser scanning is one of the modern and actively-developing remote sensing techniques, resulting in a point cloud, containing a set of different attributes for each point. One of the positive features of laser scanning is the high accuracy of the results; this is achieved by obtaining a large number of points describing the scanned object. In some circumstances, point clouds may contain billions of points, which require hundreds of gigabytes to be stored. Loading and processing of such huge data require large time and computational resources. The first problem is such massive point clouds initial downloading and pre-processing. The standard approach is the sequential processing of laser scanning results, which requires a significant amount of time. In this paper, we have conducted research and testing of various approaches for loading and processing of point clouds, one of the proposed approaches is the use of multithreading to significantly reduce time. The guideline for improvement of processing of laser scanning point clouds with use of multithreading is presented.

Keywords: Algorithm · Laser scanning data · Point cloud · Parallel processing · Multithreading

1 Introduction

Laser scanning is a common method for 3D data capture, which allows obtaining information about the shape (three-dimensional coordinates) and appearance (color and intensity of reflection beams) of objects in the real world [1, 2], the results are presented in the form of a point cloud, then these data can be used to build spatial digital models of measured objects [3, 4]. The use of laser scanning point cloud is common in many spheres, for example, land surveying, architecture, geologic exploration activities, industrial enterprises, civil engineering, and more [5, 6]. Specific examples of the tasks to be solved are forest management [7], 3D city modeling [8], road inventory [9],

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11619, pp. 289–305, 2019.
https://doi.org/10.1007/978-3-030-24289-3_22

ground surface reconstruction [10], pipeline detection [11], and wire detection [12]. Moreover, this is only a small part of the issues solved using laser scanning data, for example, the fastest growing branch is as-built BIM [13–17].

One of the key issues when working with point clouds is a large amount of information that needs to be processed [18–21]. Any work with laser scanning data can be divided into two stages: loading and pre-processing of data; data processing (specific algorithm) [15, 22]. Most of the research is focused on optimizing the processing stage [1, 2, 4, 23]. However, there are tasks that do not require a lot of time for the data processing stage, for example, a preview of a point cloud [15, 22]. In such tasks, the loading and pre-processing stage takes most of the time [24]. The most common approach to increasing load speed is to increase the bandwidth of the data bus, which can be achieved by using a faster solid state drive (SSD) or RAID [25]. But in order to increase the speed of loading even more, multithreading can be used.

The results of research and testing of various approaches in the loading and pre-processing stage is discussed in this paper. A sequential reading of a point cloud from a file with subsequent processing can be considered the standard approach. We hypothesized that in some situations the use of multithreading can help significantly speed up the loading of a point cloud. To test this hypothesis in this paper, experiments have been conducted using various approaches to loading laser scanning data. In addition, some standard libraries will be tested and compared. In conclusion, we will provide recommendations and describe further plans.

2 Materials and Methods

2.1 Point Cloud Loading and Pre-processing

The main problem of loading laser scanning data is too large volume, the size of point clouds can amount to hundreds of gigabytes. Thus, it will not be possible to load all data into RAM at once. There are many solutions to this problem, the two simplest are: (1) splitting the original point cloud into separate sections; (2) loading the point cloud with part by part.

The first approach is that each section is a small point cloud, so that each section can be loaded and processed separately. In the second approach, the cloud is sequentially loaded in parts, the loaded part is processed, and then the next part of the file is loaded.

These approaches are not suitable for all algorithms since there are algorithms that need neighboring points for all points to work, and when processing only a section of the cloud, there are no neighbor points for endpoints. In the first approach, this problem is easily solved by modifying the partitioning algorithm; it's enough to break up a point cloud into sections with small intersecting indents, the size of which depends on the algorithm used. In the second approach, you will have to use a spatial data structure, for example, an octree [26], but for this, you first need to build a spatial data structure for the entire point cloud. In further tests, the second approach is used, since to implement the first one, it is still necessary to load the initial point cloud.

2.2 Existing Approaches

Many different formats are used to store point clouds, such as OBJ, PLY, XYZ, PCD, LAS and many others [27]. During the tests, we used LAS VERSION 1.4 file format, since this is common and public format [28].

In this paper the existing approaches are considered on the example of LAStools [29] and PDAL [30] libraries, as they are the most widely used at the moment [31].

These libraries also use the approach of block loading of the source point cloud, which was described above. Thus, the point cloud sections are loaded and pre-processed sequentially in a single thread. Thus, the next point cloud section will be loaded only after the completion of the previous one, which may slow down the loading process.

We propose to load and pre-process point cloud data in parallel manner to increase algorithm throughput. This can be achieved on modern multi-core systems using multithreading. Thus, our approach is to separate the loading process into two phases executed in a distinct thread.

2.3 Background

Several tests have been developed and implemented to study various methods of laser scanning data loading (Table 1). The tests were written in the C++ programming language, as this language allows you to perform low-level work with memory, which is necessary for working with large amounts of data.

Table 1. List of tests

Title	Description
Test 1	Loading a file using a temporary buffer. Buffer creation time is not measured
Test 2	Loading a file using the temporary buffer. The buffer creation time is measured
Test 3	Loading a file using the temporary buffer. The buffer creation time is measured. All loaded data are copied to the internal data structure
Test 4	Loading a file using the temporary buffer. The buffer creation time is measured. Only the coordinates of the loaded points are copied to the internal data structure
Test 5	Loading a file using several temporary buffers. The buffer creation time is measured. All loaded data is copied to the internal data structure. Using two threads: the first one loads the data into one of the free buffers, the second one copies the data
Test 6	Loading a file using several temporary buffers. The buffer creation time is measured. A part of the data will be preprocessed. All loaded data is copied to the internal data structure. Using two threads: the first one loads the data into one of the free buffers, the second one copies the data
Test 7	Loading a file using several temporary buffers. The buffer creation time is measured. A part of the data will be preprocessed. All loaded data is copied to the internal data structure. Using three threads: the first one loads the data into one of the free buffers, the second one and third one copies the data
Test 8	Loading a file using PDAL
Test 9	Loading a file using LAStools

Test 1 and Test 2 are standard sequential readings of a file in parts, these tests are necessary to determine the dependence of the file loading time on the size of the temporary buffer. Test 3 and Test 4 based on Test 2. These tests allow to estimate the effect of delays between loading parts of a file on the total loading time. Test 8 and Test 9 are tests of existing libraries.

One of the features of the LAS format is various point formats. Depending on the point format specified in the file header (point format ID), each point has its own set of attributes described in the file format specification, so the size of one point can vary from 20 bytes to 67 bytes. Parameters of laser scanning point clouds for tests are presented in Table 2.

Test bench configuration:

CPU: Intel® Core™ i7-7820HQ CPU @ 2.90 GHz
RAM: 2 x SODIMM DDR4 2400 MHz
SSD: Samsung SSD 850 PRO 1 TB
OS: Ubuntu 18.04
File system: Ext4

Table 2. Clouds for testing

Title	Number of point records	Point format ID	Point size, bytes	Point cloud size, GB
Cloud 1	42'211'048	3	24	1.4
Cloud 2	150'000'000	5	63	9.5
Cloud 3	300'000'000	10	67	20.1

3 Results and Discussion

3.1 Loading with Temporary Buffer

To begin with, it is necessary to investigate the dependence between the file read speed and the size of the temporary buffer. Test 1 and Test 2 were developed for this.

Pseudocode for Test 1:

```
Reader r(filepath);       // Point Cloud reader
Buffer b(bufferSize);       // Temporary buffer
/* Time measurement starts here */
while(r.readChunk(b, b.size())) {}    // Read all points
/* Time measurement ends here */
```

The results of the first test, the dependence between the file read speed and the size of the temporary buffer are shown in Fig. 1.

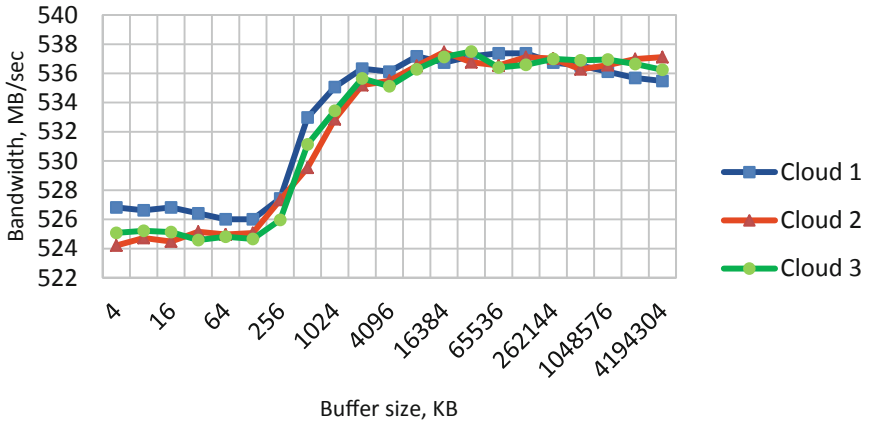

Fig. 1. The results of the first test, the dependence between the file read speed and the size of the temporary buffer.

From the observed data (Fig. 1), we can conclude that for clouds of different sizes the dependence of the load time on the buffer size is the same, it is also worth noting that using too small buffer slows down the loading speed by about 2–3%. In Fig. 1 you can see that in most cases the loading speed is close to the maximum for the SSD [12] used in the test bench.

Pseudocode for Test 2 are following:

```
Reader r(filepath);      // Point Cloud reader
/* Time measurement starts here */
Buffer b(bufferSize);      // Temporary buffer
while(r.readChunk(b, b.size())) {}    // Read all points
/* Time measurement ends here */
```

The results of the second test, the dependence between the file read speed and the size of the temporary buffer are shown in Fig. 2.

Based on the observed results (Fig. 2), it can be concluded that after exceeding a certain size, the creation time of the temporary buffer can greatly affect the total file load time. The smaller the difference between the file size and the buffer size, the stronger the influence. In the case where the buffer size exceeds the size of the point cloud itself more than three times, the file load speed drops by almost 60%. From the above data, we can conclude that the optimal buffer size for loading is somewhere between 2 and 32 MB. In Fig. 2, you can see that the load speed depends on the data bus bandwidth.

Possible approaches to accelerate the load of the point cloud: (1) multithreaded reading from a file; (2) reading part of the data from a file. Using multithreaded reading from the file does not make sense, since the maximum bandwidth of the data bus has been reached, therefore, using this approach will not improve the load speed.

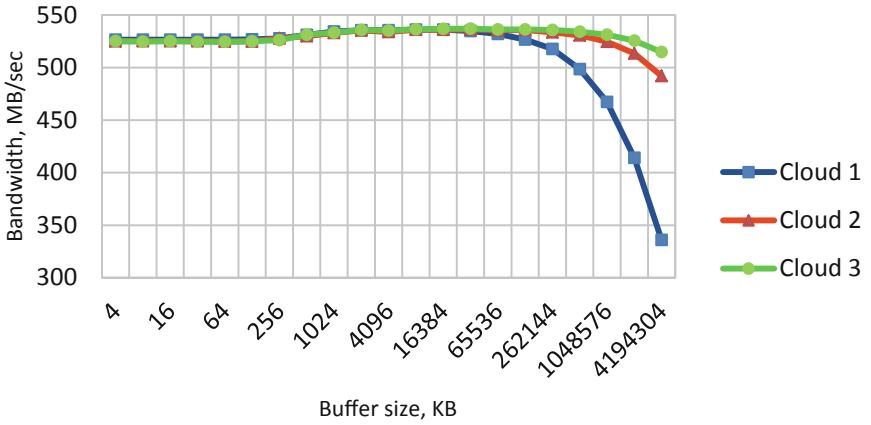

Fig. 2. The results of the second test, the dependence between the file read speed and the size of the temporary buffer.

Since many algorithms need only the coordinates of the points to work, then it would be possible to read only the coordinates, ignoring the other attributes. For example, for a format of point 10, it would be possible to save 55 bytes since the coordinates occupy only 12 bytes, this would allow downloading 82% less data. Unfortunately, in LAS format, the data of each point is stored sequentially, which means that it will be necessary to read small pieces of a file many times. But the operating system reads from the file in large blocks (the minimum block size for Ubuntu is 4 KB), so you will still have to read the entire file. This approach is suitable for storage formats where point attributes are stored sequentially, but not the points themselves.

Due to the first two tests, we found out the optimal buffer size for a simple file load, but no one loads a point cloud without a reason. The following tests will simulate a common situation where data downloaded from a file need to be copied into internal data structures. For this, Test 3 and Test 4 were developed.

Pseudocode for Test 3 are following:

```
Reader r(filepath);    // Point Cloud reader
Container c(r.getPointCnt()); // Point Cloud container

/* Time measurement starts here */
Buffer b(bufferSize);
while(r.readChunk(b, b.size())) // For all chunks
  c.pushAllData(b);// Push all readed data to end of con-
tainer
/* Time measurement ends here */
```

The results of the third test, the dependence between the file read speed and the size of the temporary buffer are shown in Fig. 3.

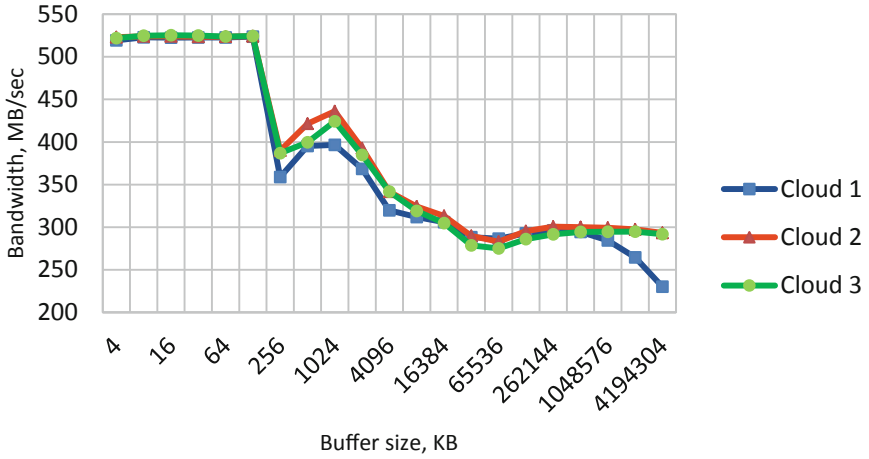

Fig. 3. The results of the third test, the dependence between the file read speed and the size of the temporary buffer.

Figure 3 presents the results of the third test; according to the data, it is clear that the load time dependence on the buffer size is the same for point clouds of different sizes. An interesting fact is that at a certain point the load time greatly increases by about 35–45% for different clouds, this happens because the delays caused by copying the loaded data interrupt the file system cache. To solve this problem, you can try to apply a slightly upgraded approach, described above, not to copy all the data, but to copy only the necessary data. In the fourth test, only the coordinates of the point are copied, which theoretically should reduce the delays between reading from the file.

Pseudocode for Test 4 is following:

```
Reader r(filepath);    // Point Cloud reader
Container c(r.getPointCnt()); // Point Cloud container
/* Time measurement starts here */
Buffer b(bufferSize);
while(r.readChunk(b, b.size())) // For all chunks
  c.pushOnlyXYZData(b);  // Push only xyz coordinates to
end of container
/* Time measurement ends here */
```

The results of the fourth test, the dependence between the file read speed and the size of the temporary buffer are shown in Fig. 4.

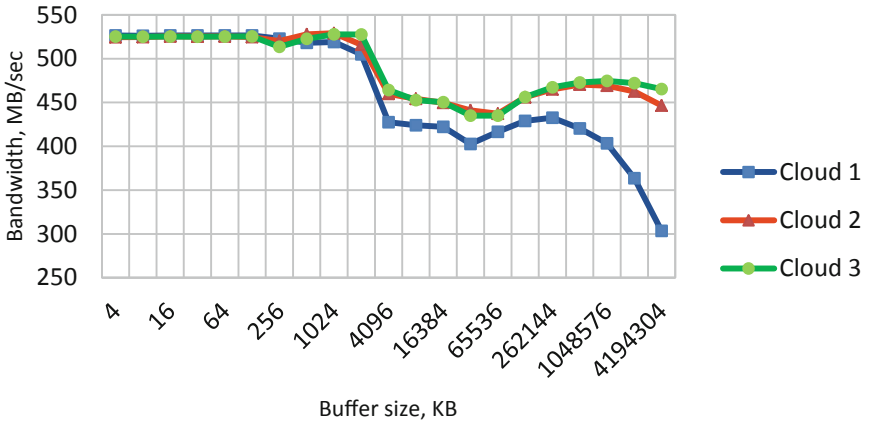

Fig. 4. The results of the fourth test, the dependence between the file read speed and the size of the temporary buffer.

Figure 4 presents the results of the fourth test. According to the data, you can really notice that the throughput increased on average by 12–23% for larger buffers. This is due to the reduction of delays between reading from the file. In addition, you may notice that the increase in load time comes earlier for Cloud 1 and is higher, this is due to the fact that copying only the coordinates of the point reduced the amount of copy data for Cloud 1 to only 8 bytes (per point), whereas for Cloud 2–51 bytes, for Cloud 3–55 bytes. Thus, the optimal buffer size for loading and copying of data is 16–128 KB, in addition, reducing the amount of copied data also improves the overall situation with the file load time.

3.2 Parallel Loading and Preprocessing

The next step is to use multithreading to eliminate delays between reading from a file, for this we developed Test 5. The main idea of this test is the use of several temporary buffers and two lock-free queues, one queue contains free buffers, the second one contains the occupied ones. In this test, two threads are used: the first thread loads data from the file into the free buffer, the second thread copies the data from the occupied buffer to the internal data structure. In the test, we used five temporary buffers.

Pseudocode for Test 5 is following:

```
Reader r(filepath);      // Point Cloud reader
Container c(r.getPointCnt());   // Point Cloud container

/* Time measurement starts here */

// Queue for freed buffers (Lock-free)
Queue<Buffer> freeQ(queueLength);
/* Push free buffers to freeQ */
// Queue for loaded buffers (Lock-free)
Queue<Buffer> loadedQ(queueLength);

auto t1 = std::thread([&]
{
  while(/* not all points readed from file */) {
    Buffer b = freeQ.pop();    // Pop free buffer from
queue
    r.readChunk(b, b.size());    // Read file chunk to
buffer
    loadedQ.push(buffer);    // Push points to queue
  }
});
auto t2 = std::thread([&]
{
  while(/* not received stop signal */) {
    if(Buffer b = loadedQ.pop()){ // Pop points from
queue
      c.pushAllData(b);  // Push all readed data to end of
container
      freeQ.push(buffer);  // Push used buffer to queue
    }
  }
});
// Wait for threads ends
t1.join();
t2.join();
/* Time measurement ends here */
```

The results of the fifth test, the dependence between the file read speed and the size of the temporary buffer are shown in Fig. 5.

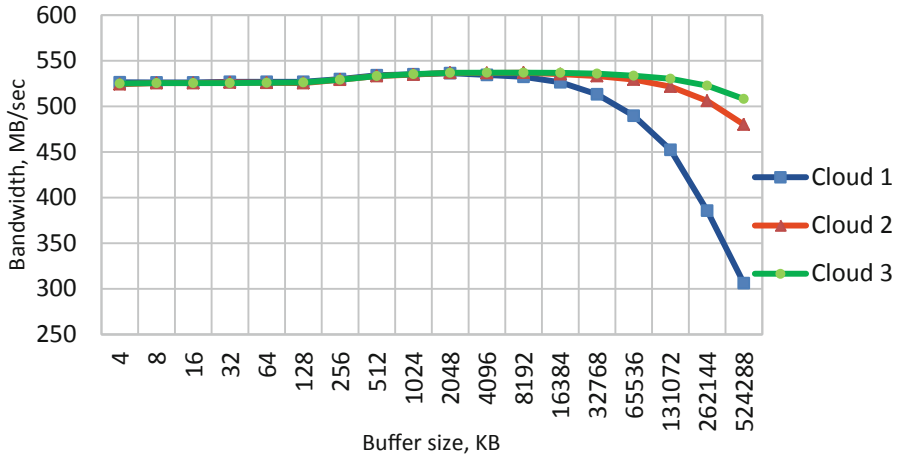

Fig. 5. The results of the fifth test, the dependence between the file read speed and the size of the temporary buffer.

Figure 5 presents the results of the fifth test. According to the data, the use of multithreading has reduced the delay between reading parts of the file. With the correct choice of the number of temporary buffers and their size, you can achieve such a file load time as in Test 2, taking into account that in Test 5, in addition to reading the file, all the data is also copied. Unfortunately, due to the use of several temporary buffers, an increase in load time occurs much earlier than in Test 2.

Test 6 simulates the usual situation for processing laser scanning data when some of the loaded data needs to be pre-processed. The simplest example of such a situation is the application of displacement and scaling to the coordinates of a point. Pseudocode for Test 6 is following:

```
Reader r(filepath);      // Point Cloud reader
Container c(r.getPointCnt());   // Point Cloud container

/* Time measurement starts here */

// Queue for freed buffers (Lock-free)
Queue<Buffer> freeQ(queueLength);
/* Push free buffers to freeQ */
// Queue for loaded buffers (Lock-free)
Queue<Buffer> loadedQ(queueLength);

auto t1 = std::thread([&]
{
  while(/* not all points readed from file */) {
    Buffer b = freeQ.pop();    // Pop free buffer from
queue
    r.readChunk(b, b.size());    // Read file chunk to
buffer
    loadedQ.push(buffer);    // Push points to queue
  }
});
auto t2 = std::thread([&]
{
  while(/* not received stop signal */) {
    if(Buffer b = loadedQ.pop()){ // Pop points from
queue
      preprocess (b);  // Preprocess XYZRGBData
      c.pushAllData(b);  // Push all readed data to end of
container
      freeQ.push(buffer);  // Push used buffer to queue
    }
  }
});
// Wait for threads ends
t1.join();
t2.join();
/* Time measurement ends here */
```

The results of the sixth test, the dependence between the file read speed and the size of the temporary buffer are shown in Fig. 6.

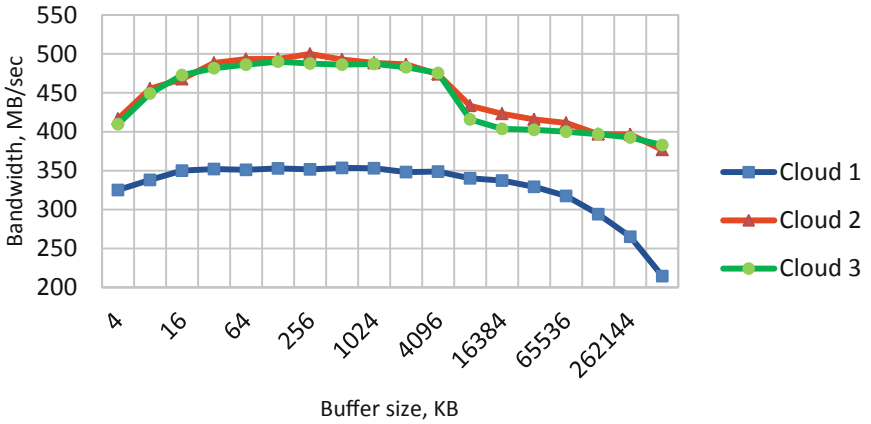

Fig. 6. The results of the sixth test, the dependence between the file read speed and the size of the temporary buffer.

Figure 6 presents the results of the sixth test, the data show an increase in file load time, for the best cases, up to almost 54% for Cloud 1 and about 11% for Cloud 2 and Cloud 3. This is due to the fact that the thread responsible for pre-processing and copying is no longer up to its tasks, as a result of which there are delays in reading the file. In Fig. 6, you can see a big difference in the load speed of Cloud 1, this is because the pre-processing function affects 90% of Cloud 1 data, and only 28.6% of Cloud 2 data and 26.9% of Cloud 3 data.

To solve the problem in Test 6, we developed Test 7. The main idea is the use of three threads, the third thread, like the second one, is used for pre-processing and copying of the loaded data.

Pseudocode for Test 7 is following:

```
Reader r(filepath);      // Point Cloud reader
Container c(r.getPointCnt());   // Point Cloud container

/* Time measurement starts here */

// Queue for freed buffers (Lock-free)
Queue<Buffer> freeQ(queueLength);
/* Push free buffers to freeQ */
// Queue for loaded buffers (Lock-free)
Queue<Buffer> loadedQ(queueLength);

auto t1 = std::thread([&]
{
  while(/* not all points readed from file */) {
    Buffer b = freeQ.pop();    // Pop free buffer from
queue
    r.readChunk(b, b.size());    // Read file chunk to
buffer
    loadedQ.push(buffer);    // Push points to queue
  }
});
auto f = [&]()
{
  while(/* not received stop signal */) {
    if(Buffer b = loadedQ.pop()){ // Pop points from
queue
      c.pushAllData(b);// Push all readed data to end of
container
      freeQ.push(buffer);  // Push used buffer to queue
    }
  }
};

auto t2 = std::thread(f);
auto t3 = std::thread(f);
// Wait for threads ends
t1.join();
t2.join();
t3.join();
/* Time measurement ends here */
```

The results of the seventh test, the dependence between the file read speed and the size of the temporary buffer are shown in Fig. 7.

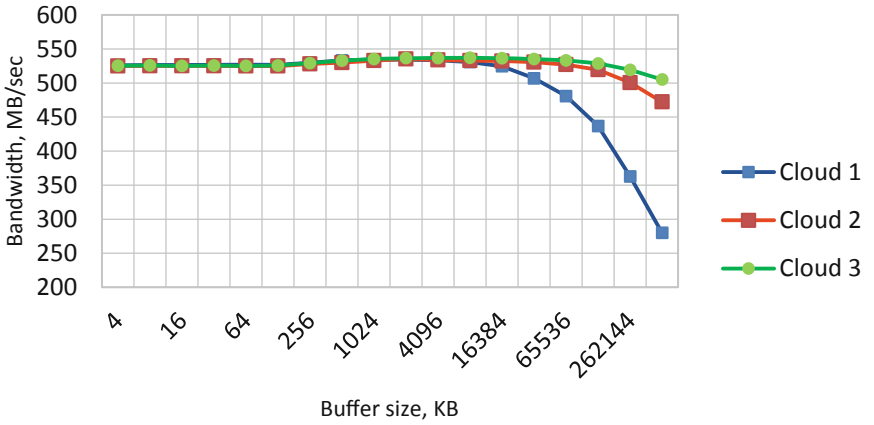

Fig. 7. The results of the seventh test, the dependence between the file read speed and the size of the temporary buffer.

Figure 7 presents the results of the seventh test, the data almost coincides with the results of Test 5, this was achieved by introducing an additional thread for pre-processing and copying, which made it possible to eliminate delays in reading the file by the first thread. Due to this, almost the maximum load speed was achieved, taking into account that in parallel with the loading, all data are pre-processed and copied.

At this point, we can conclude when correctly setting parameters such as the number of temporary buffers, the size of temporary buffers and the number of processing threads, you can achieve optimal results when loading and processing a point cloud.

3.3 Third-Party Libraries

Test 8 and Test 9 were developed for testing third-party libraries. We chose two most common libraries for loading LAS files in C++: PDAL and LAStools.

PDAL is an open source library for working with point cloud data, one of the possibilities provided by this library is load files in LAS format. Unfortunately, this library does not support loading of Cloud 2 and Cloud 3 because of the point format, so testing was performed only on Cloud 1.

LAStools is one of the most common open source laser scanning data processing tools. This set of tools allows you to load LAS files using the LASlib library. Unfortunately, this library supports reading the file only by one point, so there is no possibility to adjust the size of the temporary buffer.

The test results are presented in the Table 3.

Table 3. Results for Test 8 and Test 9

Test cloud	PDAL (Test 8) load time, ms	LAStools (Test 9) load time, ms	Our approach load time, ms
Cloud 1	10552	5089	2552
Cloud 2	–	23895	16791
Cloud 3	–	44057	35698

4 Conclusions

As a result of a survey of various approaches for loading a point cloud, we identified problems with the standard data loading method. The problem is the increase in file load time when an intermediate pre-processing of loaded data occurs. Pre-processing makes delays between reading parts of a file because of what bandwidth drops.

To solve this problem, it was proposed to separate the loading of the raw section of the file and its pre-processing. For this, multithreading was used, and this decision eliminated the delays between reading. Proper configuration of parameters such as the size of the temporary buffer, the number of temporary buffers and the number of preprocessing threads can allow you to achieve the maximum file load speed. The study showed that for all cases the optimal buffer size is in the range of 32–128 KB, the number of temporary buffers and the number of pre-processing threads is very dependent on the pre-processing algorithm, the more complex the algorithm, the more pre-processing threads are necessary to maintain a balance between loading and pre-processing. As a result, it was possible to reach the maximum file load speed limited by the data bus bandwidth.

This study was conducted using the Ubuntu operating system with the Ext4 (Fourth Extended File System). The next step is to run these tests using a Windows operating system with the NTFS (New Technology File System) to confirm the results of our tests.

Acknowledgements. The research was supported by Ministry of Education and Science of Russia within the framework of the Federal Program "Research and Development in Priority Areas for the Development of the Russian the Science and Technology Complex for 2014–2020" (project ID RFMEFI58417X0025).

References

1. Puente, I., González-Jorge, H., Martínez-Sánchez, J., Arias, P.: Review of mobile mapping and surveying technologies. Measurement **46**(7), 2127–2145 (2013)
2. Guan, H., Li, J., Cao, S., Yu, Y.: Use of mobile LiDAR in road information inventory: a review. Int. J. Image Data Fusion **7**(3), 219–242 (2016)
3. Badenko, V., Zotov, D., Fedotov, A.: Hybrid processing of laser scanning data. In: E3S Web of Conferences – EDP Sciences, vol. 33, article number 01047 (2018)
4. Dore, C., Murphy, M.: Current state of the art historic building information modelling. Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci. **42**, 185–192 (2017)

5. Vosselman, G., Coenen, M., Rottensteiner, F.: Contextual segment-based classification of airborne laser scanner data. ISPRS J. Photogramm. Remote Sens. **128**, 354–371 (2017)
6. Tomljenovic, I., Höfle, B., Tiede, D., Blaschke, T.: Building extraction from airborne laser scanning data: an analysis of the state of the art. Remote Sens. **7**(4), 3826–3862 (2015)
7. Penner, M., Woods, M., Pitt, D.: A comparison of airborne laser scanning and image point cloud derived tree size class distribution models in boreal Ontario. Forests **6**(11), 4034–4054 (2015)
8. Heo, J., Jeong, S., Park, H.-K., Jung, J., Han, S., Hong, S., Sohn, H.-G.: Productive high-complexity 3D city modeling with point clouds collected from terrestrial LiDAR. Comput. Environ. Urban Syst. **41**, 26–38 (2013)
9. Pu, S., Rutzinger, M., Vosselman, G., Elberink, S.O.: Recognizing basic structures from mobile laser scanning data for road inventory studies. ISPRS J. Photogramm. Remote Sens. **66**(6), S28–S39 (2011)
10. Badenko, V., Fedotov, A., Vinogradov, K.: Algorithms of laser scanner data processing for ground surface reconstruction. In: Gervasi, O., et al. (eds.) ICCSA 2018. LNCS, vol. 10961, pp. 397–411. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95165-2_28
11. Son, H., Kim, C., Kim, C.: Fully automated as-built 3D pipeline extraction method from laser-scanned data based on curvature computation. J. Comput. Civ. Eng. **29**(4), B4014003 (2014)
12. Guo, B., Li, Q., Huang, X., Wang, C.: An improved method for power-line reconstruction from point cloud data. Remote Sens. **8**(1), 36 (2016)
13. Hichri, N., Stefani, C., De Luca, L., Veron, P., Hamon, G.: From point cloud to BIM: a survey of existing approaches. Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci. ISPRS Arch. **40**(5W2), 343–348 (2013)
14. Barazzetti, L.: Parametric as-built model generation of complex shapes from point clouds. Adv. Eng. Inform. **30**(3), 298–311 (2016)
15. Volk, R., Stengel, J., Schultmann, F.: Building Information Modeling (BIM) for existing buildings—literature review and future needs. Autom. Constr. **38**, 109–127 (2014)
16. Badenko, V., Fedotov, A., Zotov, D.: Extracting features from laser scanning point cloud. In: SHS Web of Conferences, vol. 44, article number 00013 (2018)
17. Badenko, V., Volgin, D., Lytkin, S.: Deformation monitoring using laser scanned point clouds and BIM. In: MATEC Web of Conferences, vol. 245, article number 01002 (2018)
18. Liu, X., Meng, W., Guo, J., Zhang, X.: A survey on processing of large-scale 3D point cloud. In: El Rhalibi, A., Tian, F., Pan, Z., Liu, B. (eds.) Edutainment 2016. LNCS, vol. 9654, pp. 267–279. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40259-8_24
19. Kukko, A., Kaartinen, H., Hyyppä, J., Chen, Y.: Multiplatform mobile laser scanning: usability and performance. Sensors **12**(9), 11712–11733 (2012)
20. Murphy, M., McGovern, E., Pavia, S.: Historic building information modelling-adding intelligence to laser and image based surveys of European classical architecture. ISPRS J. Photogramm. Remote Sens. **76**, 89–102 (2013)
21. Axelsson, P.: Processing of laser scanner data - algorithms and applications. ISPRS J. Photogramm. Remote Sens. **54**(2–3), 138–147 (1999)
22. Tang, P., Huber, D., Akinci, B., Lipman, R., Lytle, A.: Automatic reconstruction of as-built building information models from laser-scanned point clouds: a review of related techniques. Autom. Constr. **19**(7), 829–843 (2010)
23. Zhang, J., Lin, X.: Advances in fusion of optical imagery and LiDAR point cloud applied to photogrammetry and remote sensing. Int. J. Image Data Fusion **8**(1), 1–31 (2017)
24. Pfeifer, N., Mandlburger, G., Otepka, J., Karel, W.: OPALS - a framework for airborne laser scanning data analysis. Comput. Environ. Urban Syst. **45**, 125–136 (2014)

25. Patterson, D.A., Gibson, G., Katz, R.H.: A case for redundant arrays of inexpensive disks (RAID). SIGMOD Rec. **17**(3), 109–116 (1988)
26. Han, S.: Towards efficient implementation of an octree for a large 3D point cloud. Sensors **18**(12), 4398 (2018)
27. Vo, A.V., Laefer, D.F., Bertolotto, M.: Airborne laser scanning data storage and indexing: state-of-the-art review. Int. J. Remote Sens. **37**(24), 6187–6204 (2016)
28. LAS Specification, Version 1.4, Revision 13, 15 July 2013. http://www.asprs.org/a/society/committees/standards/LAS_1_4_r13.pdf. Accessed 15 Mar 2019
29. rapidlasso GmbH Homepage. https://rapidlasso.com/lastools/. Accessed 15 Mar 2019
30. PDAL - Point Data Abstraction Library Homepage. https://pdal.io/. Accessed 15 Mar 2019
31. Van Natijne, A.L., Lindenbergh, R.C., Hanssen, R.F., Lindenbergh, R.C., Hanssen, R.F.: Massive linking of PS-InSAR deformations to a national airborne laser point cloud. Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci. **42**(2), 1137–1144 (2018)