International Journal of
Geo-Information

Article

A Hybrid Process/Thread Parallel Algorithm for Generating DEM from LiDAR Points

Yibin Ren [1,2], Zhenjie Chen [3], Ge Chen [1,2], Yong Han [1,2,*] and Yanjie Wang [1,2]

1 Qingdao Collaborative Innovation Center of Marine Science and Technology, College of Information Science and Engineering, Ocean University of China, No. 238, Songling Road, Qingdao 266100, China; ryb0911@163.com (Y.R.); gechen@ouc.edu.cn (G.C.); yanjie_w@sina.cn (Y.W.)
2 Laboratory for Regional Oceanography and Numerical Modeling, Qingdao National Laboratory for Marine Science and Technology, No. 1, Wenhai Road, Qingdao 266237, China
3 Jiangsu Provincial Key Laboratory of Geographic Information Science and Technology, Nanjing University, No. 163, Xianlin Avenue, Nanjing 210023, China; chenzj@nju.edu.cn
* Correspondence: yonghan@ouc.edu.cn; Tel.: +86-0532-6678-6365

Received: 10 August 2017; Accepted: 24 September 2017; Published: 28 September 2017

Abstract: Airborne Light Detection and Ranging (LiDAR) is widely used in digital elevation model (DEM) generation. However, the very large volume of LiDAR datasets brings a great challenge for the traditional serial algorithm. Using parallel computing to accelerate the efficiency of DEM generation from LiDAR points has been a hot topic in parallel geo-computing. Generally, most of the existing parallel algorithms running on high-performance clusters (HPC) were in process-paralleling mode, with a static scheduling strategy. The static strategy would not respond dynamically according to the computation progress, leading to load unbalancing. Additionally, because each process has independent memory space, the cost of dealing with boundary problems increases obviously with the increase in the number of processes. Actually, these two problems can have a significant influence on the efficiency of DEM generation for larger datasets, especially for those of irregular shapes. Thus, to solve these problems, we combined the advantages of process-paralleling with the advantages of thread-paralleling, forming a new idea: using process-paralleling to achieve a flexible schedule and scalable computation, using thread-paralleling inside the process to reduce boundary problems. Therefore, we proposed a hybrid process/thread parallel algorithm for generating DEM from LiDAR points. Firstly, at the process level, we designed a parallel method (PPDB) to accelerate the partitioning of LiDAR points. We also proposed a new dynamic scheduling strategy to achieve better load balancing. Secondly, at the thread level, we designed an asynchronous parallel strategy to hide the cost of LiDAR points' reading. Lastly, we tested our algorithm with three LiDAR datasets. Experiments showed that our parallel algorithm had no influence on the accuracy of the resultant DEM. At the same time, our algorithm reduced the conversion time from 112,486 s to 2342 s when we used the largest dataset (150 GB). The PPDB was parallelizable and the new dynamic scheduling strategy achieved a better load balancing. Furthermore, the asynchronous parallel strategy reduced the impact of LiDAR points reading. When compared with the traditional process-paralleling algorithm, the hybrid process/thread parallel algorithm improved the conversion efficiency by 30%.

Keywords: parallel geo-computing; hybrid process/thread paralleling; DEM generation; LiDAR points

1. Introduction

1.1. Background

Grid digital elevation models (DEM) are very important in geographic information science (GIS). It is widely used in hydrological analysis, visibility analysis and terrain analysis [1,2]. With the progress of data acquisition technology and the development of spatial analyzing methods, the demand for DEMs in different spatial scales has been increasing in recent years. The airborne Light Detection and Ranging (LiDAR) has advantages of wide scanning range, fast data acquisition speed and high point destiny, making it very suitable for DEM generation, especially for large areas [3–5]. However, the volume of the LiDAR dataset is very large. For example, a full scanning project of airborne LiDAR can generate a raw dataset with hundreds of millions of points, whose volume is dozens of gigabytes (GB) [6,7]. Thus, it is a great challenge for the traditional computer to make an effective conversion from LiDAR points to DEM grids.

Parallel computing based on multi-core computers provides a new way to solve this challenge. Compared with the traditional serial algorithm, parallel algorithms can accelerate the processing procedure by making full use of multiple cores. Thus, using parallel computing to improve the processing efficiency of spatial data has been a hot topic in the field of geo-computing. A series of parallel geo-computing algorithms has been proposed, such as spatial data conversion algorithm [8], remote sensing images processing algorithm [9], and raster computing algorithms [10–12], etc. This provides important references for LiDAR dataset processing. We can also use parallel computing to accelerate the conversion efficiency from LiDAR points to DEM. Therefore, designing a high-performance parallel algorithm for generating DEM from LiDAR points is of great significance.

1.2. Related Work

In recent years, a series of parallel algorithms for generating DEM from LiDAR points has been designed by researchers. According to the differences between parallel strategies, these algorithms can be grouped into three categories: (1) thread-paralleling algorithms; (2) process-paralleling algorithms; and (3) heterogeneous-paralleling algorithms. The representation of the first category is in study [13] where the authors partitioned the LiDAR dataset into a lot of square blocks and scheduled them statically with a pipeline model by Open Multi-Processing (OpenMP) in a multi-core computer. Each thread completed four steps including inputting, indexing, interpolating and outputting to generate DEM independently; all threads were synchronized in time. In the second category, typically, a cluster with several computation nodes is selected as the hardware environment; the Message Passing Interface (MPI) is selected to implement the programming of process-paralleling. Huang et al. [14] implemented the basic master-slave model by MPI in a high-performance cluster (HPC) where the master distributed raw data blocks to slaves and gathered computation results, merged them into the final DEM. Han et al. [15] implemented a process-paralleling algorithm in a PC-cluster, using MPI to pass values of points located at the boundary of each data block to solve the data barrier existing in the different processes. The last category is the heterogeneous paralleling algorithms combining Central Processing Unit (CPU) programming with Graphics Processing Unit (GPU) programming. These algorithms can leverage the power of floating-point computing in GPU to accelerate the computation of spatial interpolation [16,17].

As known to all in parallel programming, the process is the unit of computation resource allocation; a parallel algorithm designed in multiple processes can run on a HPC consisting of a series of computation nodes. Its advantages are its flexibility of scheduling and the scalability of computation [18]; however, the memory between processes is independent, which makes it difficult to share data between each other. In contrast, thread has no independent memory space which means they all share the memory of the same process [19]. Compared with process, thread requires fewer resources of the operating system which can maximize the power of a hyper-threading computer [20]. However, the algorithm implemented by threads cannot run across multiple computation nodes, which makes it more suitable to run on a computer with multiple cores. Generally, the above two programming modes are used in parallel programming of multiple CPUs. In recent years, GPU parallel programming has rapidly developed as its powerful floating-point computing capability is suitable for handling computationally-intensive problems [21]. However, the memory capacity limitations of the GPU and the gap between computation and communications can negatively influence the performance of the algorithm [22]. Additionally, as compared with CPU clusters, GPU clusters have more expensive hardware and the associativity between software and hardware requires improvement [23]. Therefore, considering a series of factors, most geo-computing parallel algorithms use the HPC with multiple CPUs as the parallel environment to achieve a variety of characteristics of GIS computing tasks [11,24,25]. Thus, the aim of this research was to design a high-performance parallel algorithm for generating DEM from LiDAR points in the HPC with multiple CPUs.

1.3. Problems

The general parallel procedure of generating DEM from LiDAR points is shown in Figure 1. To achieve a high-performance parallel algorithm, we must solve two key problems. The first one is how to achieve load balancing between computation units (Figure 1b). Load balancing is one of the most important parts of parallel programming and can affect the performance of the whole algorithm [26]. Most existing algorithms use the static scheduling strategy, which partitions the LiDAR dataset and assigns them to computation units before the algorithm running [13,27,28]. This is useful when the shape of the LiDAR dataset is regular. However, if the shape is irregular, the static strategy will not respond dynamically according to the computation progress, leading to load unbalancing. To achieve more flexible scheduling, a dynamic strategy implemented by master-slave mode was proposed by [14]. However, this basic dynamic strategy had a limitation: if the data block with the largest computation quantity was assigned last, the running time of the computation unit that processes this data block would be much longer than the others, which would affect the efficiency of the whole algorithm. Additionally, since the number of LiDAR points is very large, we should parallelize the data partitioning procedure as much as possible to minimize the partitioning time.

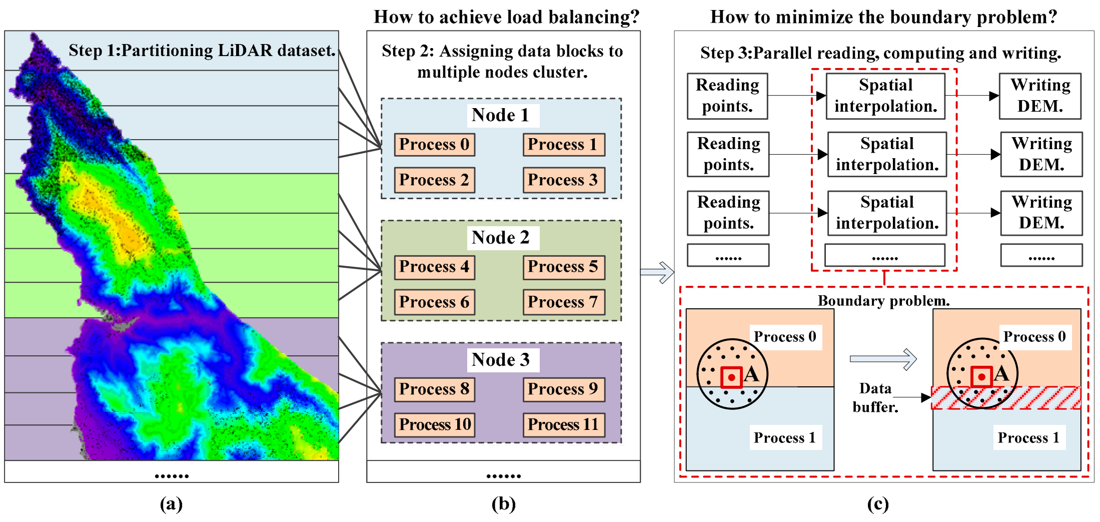

Figure 1. General parallel procedure of generating a digital elevation model (DEM) from Light Detection and Ranging (LiDAR) points: (**a**) Partition the LiDAR dataset; (**b**) Load balancing problem; and (**c**) Boundary problem.

In addition to load balancing, the other key point was how to minimize the boundary problem, especially for algorithms running on the HPC. Existing algorithms running on the HPC to generate DEM from LiDAR points have been designed in multiple processes mode. As we all known, the core computation of this procedure is spatial interpolation [29]: when a grid cell near the boundary of the data block is interpolated, points fall in the search radius of this cell, but are read by other processes that cannot be accessed directly, forming the boundary problem (Figure 1c). Usually, there are two methods to solve this problem: (1) sending the values of the point to processes that need them [15]; and (2) adding a data buffer on each data block [13]. For the first method, if there are too many processes or data blocks, the volume of data that be passed between processes will be great, which will affect the efficiency of the algorithm obviously. Compared to the first method, the cost of adding a data buffer is relatively small, which has been adopted by most algorithms. However, data buffers between adjacent data blocks overlap each other, generating redundant data reading. Especially for a larger dataset on the HPC, hundreds of data buffers form a large amount of redundant data, increasing the unnecessary data reading time. Furthermore, the parallel processing of LiDAR data is data-intensive and computation-intensive [30], which makes the cost of data reading unable to be ignored. When different processes read points falling in the same area simultaneously, data reading competition is generated [31]. Generally, there are hundreds of processes on a HPC reading LiDAR points in parallel, generating strong competition, which further increases the data reading cost.

In summary, in aiming to achieve more efficient conversion from LiDAR points to DEM, we need to design a more flexible schedule strategy and do our best to minimize the boundary problem. Thus, it is of great significant to propose a new higher performance parallel algorithm for generating DEM from LiDAR points.

1.4. Our Idea

As stated before, the advantages of process-paralleling are the flexible capability of scheduling and the scalability of computation across multiple nodes, where the advantage of thread-paralleling is memory sharing. Thus, if we read a partitioned data block with data buffer from the LiDAR file into the memory by a process, dividing it into a series of small data blocks and creating multiple threads to perform the spatial interpolation, we will not need to add data buffers on these small data blocks since all threads belong to the same process can access all points directly from the memory. In comparison with the only process-paralleling strategy, this two-level hybrid paralleling strategy can reduce the number of boundary problems obviously. At the same time, as compared with only thread-paralleling, process-paralleling at the first level can offer more flexible scheduling strategy and more scalable computation. Additionally, threads need fewer resources of the operating system than processes; using threads paralleling inside the computational node can maximize the power of hyper-threading. Based on the above analyses, we made full use of the advantages of both parallel modes and proposed a hybrid process/thread parallel algorithm for generating DEM from LiDAR points.

There were four novel contributions in this research: (1) for the first time, we proposed a hybrid strategy combine processes with threads for parallel DEM generation from LiDAR points; (2) we designed a parallel data partitioning method to reduce the partitioning time of LiDAR dataset; (3) we designed a hybrid process/thread dynamic scheduling strategy, including a new dynamic scheduling strategy based on computation quantity at the process level to achieve load balancing and an asynchronous parallel strategy between threads to hide the cost of LiDAR points' reading; and (4) we tested the hybrid parallel algorithm on a HPC using datasets of different volumes (4 GB, 30 GB, and 150 GB) , and conducted a series of comparative experiments and comprehensive analyses.

The rest of this paper is organized as follows. Section 2 introduces the overall architecture of the hybrid process/thread parallel algorithm. Section 3 introduces the parallel data partitioning at the process level and the logical data partitioning at the thread level. The hybrid process/thread dynamic scheduling strategy is introduced in Section 4. The performance experiments and comparative analyses are introduced in Section 5. Then, in Section 6, we discussed the applicability of our algorithm to different terrains and different LiDAR applications. The last section gives our conclusions and future research directions.

2. Overall Design of the Hybrid Process/Thread Parallel Algorithm

The hybrid process/thread parallel algorithm for generating DEM from LiDAR points was based on the classic master-slave strategy. Slave was the unit to read the partitioned data block of LiDAR dataset and the unit to communicate with the master. In addition to the master, all slaves needed to create multiple threads to perform the computation of DEM generation. The dynamic scheduling strategy based on computation quantity was implemented by communications between master and slaves. The overall design of the hybrid process/thread parallel algorithm for generating DEM from LiDAR points was shown in Figure 2, including eight steps:

1. All processes partitioned the LiDAR dataset in parallel. We named this procedure the Parallel Partitioning Method Considering Data Buffer (PPDB) (detailed in Section 3.1).
2. The master created a task queue based on the partitioned results. Each task was a data block waiting to be processed.
3. The master assigned one task number to each slave.
4. Each slave read LiDAR points into the memory based on the task number.
5. Each slave logically partitioned the data block stored in the memory into small blocks.
6. Threads used an asynchronous parallel strategy (detailed in Section 4.2) to perform the DEM generation computation for each small data block.
7. Slave requested a new task from the master when it finished the computation of current data block.
8. If the task queue was not empty, a new task number would be sent back to the slave, looping this procedure until the task queue was empty.

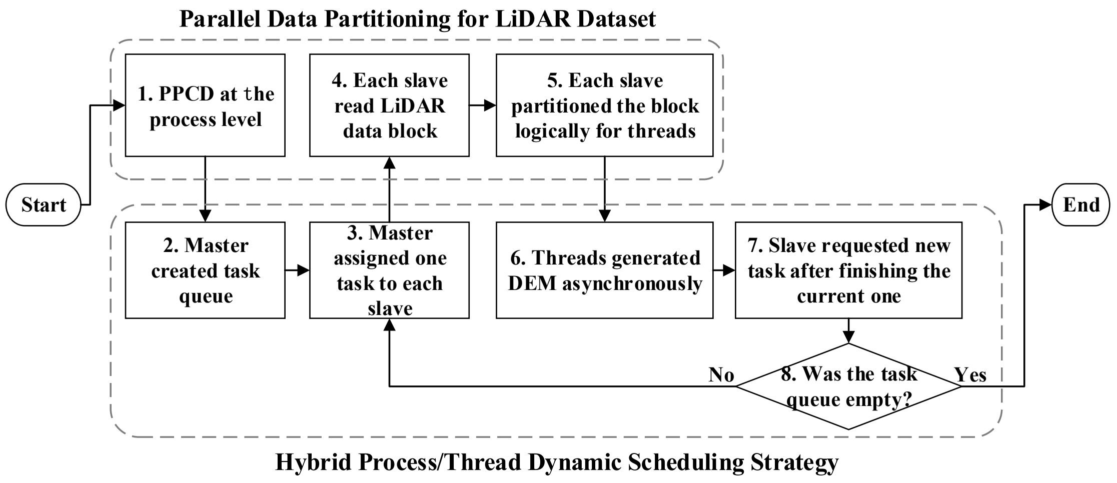
Parallel Data Partitioning for LiDAR Dataset
1. PPCD at the process level
4. Each slave read LiDAR data block
5. Each slave partitioned the block logically for threads
Start
End
2. Master created task queue
3. Master assigned one task to each slave
6. Threads generated DEM asynchronously
7. Slave requested new task after finishing the current one
No
8. Was the task queue empty?
Yes
Hybrid Process/Thread Dynamic Scheduling Strategy

Figure 2. Overall design of the hybrid process/thread parallel algorithm in DEM generation.

Generally, a parallel algorithm consists of two parts: data partitioning and scheduling strategy. Therefore, aiming to explain our algorithm more clearly, we grouped these eight steps into two parts: parallel data partitioning for the LiDAR dataset (1, 4, 5) and hybrid process/thread dynamic scheduling strategy (2, 3, 6, 7, 8). In the following, we will detail these two parts.

3. Parallel Data Partitioning for LiDAR Dataset

3.1. Parallel Partitioning Method Considering Data Buffer (PPDB) at the Process Level

We used a horizontal or vertical strip as the spatial unit to partition the whole LiDAR dataset at the process level. In this way, the data buffer of one strip only needed to extend in one direction (X or Y), which was important for paralleling the partitioning procedure. The direction of the strip was determined by the extent of LiDAR dataset. As shown in Figure 3, if the horizontal span (X direction) was bigger than the vertical span (Y direction), we would use the vertical strip as the partitioning unit, otherwise the horizontal strip would be used. The main idea of the PPDB was to obtain the coordinate of each LiDAR point by traversing the dataset parallel and calculating which strip the point belonged to. This consisted of three steps (Figure 4), as shown by the use of the horizontal strip as an example to detail the procedure of the PPDB. A series of variables was defined as below: N is the number of strips related to the volume of internal memory and the volume of the dataset; B is the size of the data buffer; M is the number of total points in the LiDAR file; N_p is the number of all processes ; P_r $(0 \leq r < N_p)$ is the process number; S_p is the number of bytes for one point in the file; P_{start} is the starting position of the first point in the LiDAR file; X_{max}, X_{min}, Y_{max}, Y_{min} are four extents of the target DEM.

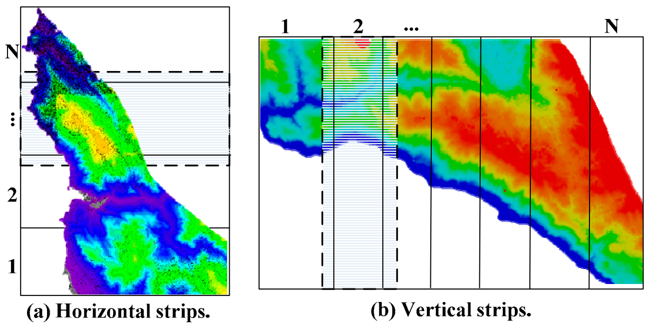

Figure 3. Strip for data partitioning: (**a**) Horizontal strips; and (**b**) Vertical strips.

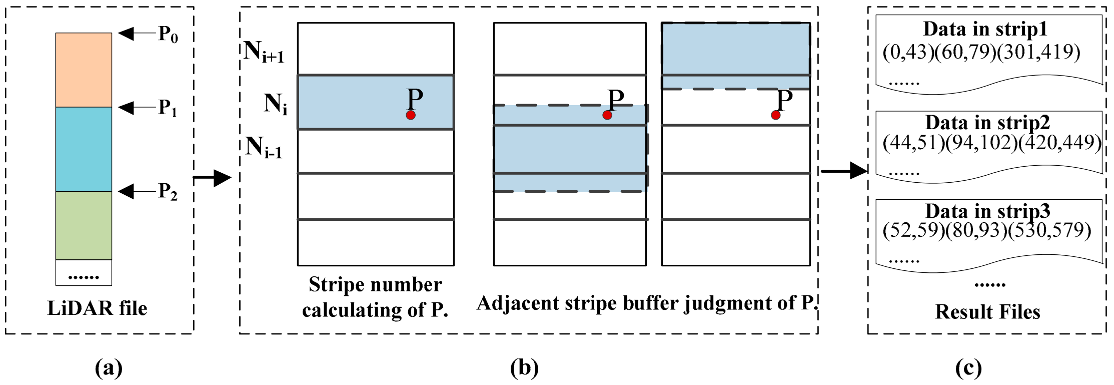

Figure 4. Procedure of the Parallel Partitioning Method Considering Data Buffer (PPDB): (**a**) Initial position calculation; (**b**) Strip span and spatial mapping; and (**c**) Writing results.

1. Initial Position Calculation for Each Process

To obtain the points' coordinates, parallel processes need to read data from different positions of the LiDAR file. Thus, we needed to calculate the initial position for the data reading pointer of each process. First, we calculated the starting LiDAR point number i for each process based on Equation (1). Next, the initial data reading position P_i in the file for each process was calculated by Equation (2), Figure 4a.

2. Strip Span Calculation and Spatial Mapping

Calculating the average span of each strip in Y direction D_y was based on Equation (3). If strips were vertical, Equation (4) was used. This step as executed by the master and the results were broadcast to all processes. All processes read points parallel and obtained the Y coordinate Y_i of each point. Based on Y_i, each process calculated the strip N_i that the point belonged to according to Equation (5) and Figure 4b. Next, we calculated whether the point belonged to the data buffer of the adjacent strips as follows: if the point satisfied Equation (6), it belonged to the buffer of strip N_{i-1} and should be added to N_{i-1}; if it satisfied Equation (7), it belonged to the buffer of strip N_{i+1}.

3. Writing Results in Turn

Resultant files were created by the master and each strip corresponded to a resultant file. All processes wrote partitioned results into files in turn when they finished traversing. As LiDAR points are stored continuously by point number, we stored points based on run-length encoding: for a series of consecutive points, only the start point number and the end point number were recorded. In this way, we not only improved the writing speed, but also saved storage space.

In the above-mentioned procedure, data partitioning was done in parallel by all processes and each process only needed to read M/N_p points. When processes read data for computing, they can obtain points located in one strip by point numbers stored in the partition file directly and do not need to judge the spatial relationship between points and strips. Point numbers stored in the same file were recorded from small to large, thus, data reading was sequential I/O instead of random I/O which improved the reading speed.

$$i = P_r \times (M/N_p) \tag{1}$$

$$P_i = P_{start} + i \times P_s \tag{2}$$

$$D_y = (Y_{Max} - Y_{Min})/N \tag{3}$$

$$D_x = (X_{Max} - X_{Min})/N \tag{4}$$

$$N_i = (Y_i - Y_{Min})/D_y + 1 \tag{5}$$

$$(N_i \neq 1)\&(Y_i \leq ((N_i - 1) \times D_y + B)) \tag{6}$$

$$(N_i \neq N)\&(Y_i \geq (N_i \times D_y - B)) \tag{7}$$

3.2. Logical Partition at the Thread Level

After one slave reads all points in one strip into memory, it needs to partition the data stored in the memory for threads. As all threads belonging to the same process share memory space, they can access any point directly. Therefore, we partitioned the data block logically: the object to be partitioned was not the data itself, but the space range corresponding to the strip. We partitioned it into multiple small rectangles logically along the long edge, and numbered rectangles as the partitioned results (Figure 5). Points stored in the memory were not partitioned physically, which saved partitioning time. Compared with the whole dataset, the shape of one data strip was more regular and the point density was also more even. Thus, we assigned all tasks to threads circularly, using multiple loops to balance the computation load between threads, which also saved communication cost across threads. The thread computed the DEM values of all cells located in the rectangle each time until it finished all rectangles assigned to it. In the interior of each process, the partition procedure as a serial operation; however, there were multiple slaves doing the partition at the same time, which meant that it was also a parallel procedure.

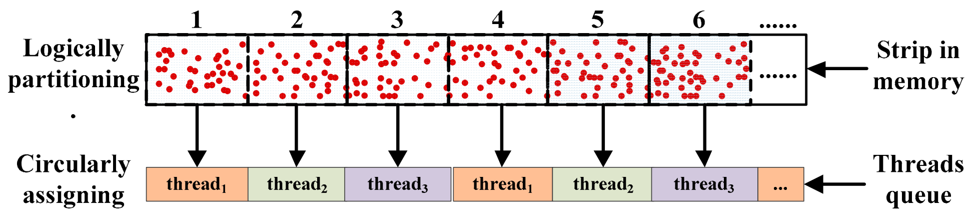

Figure 5. Logical partitioning at the thread level.

4. Hybrid Process/Thread Dynamic Scheduling Strategy

To balance the computation loads between processes, we proposed a new dynamic scheduling strategy based on computation quantity at the process level. As mentioned previously, the parallel procedure of LiDAR points is computation and data intensive. To minimize the influence of data inputting cost, we proposed an asynchronous parallel strategy to hide the reading time of LiDAR points.

4.1. Dynamic Scheduling Strategy Based on Computation Quantity

The master scheduled tasks to slave dynamically based on the computation progress in each slave; tasks with larger amounts of computation were assigned early and the smaller ones assigned later. It consisted of three steps: task queue creating, initially assigning and dynamically distributing.

1. Task Queue Creating

Generally, the frequency of the laser is fixed during scanning. Thus, the density of the LiDAR points is relatively uniform. Thus, the more points, the longer the computing time. We counted the number of points in each strip when we partitioned the data at the process level. Next, all strips were descended based on the number of points in each strip, forming the task queue which stored the numbers of the strips.

2. Initial Assigning

The master took out the first *n* tasks (*n* is the number of slave) and sent them to the slaves. Slaves received tasks and read points from the LiDAR file based on the strip number. Next, threads in each slave generated DEM from LiDAR points in parallel (Figure 6a).

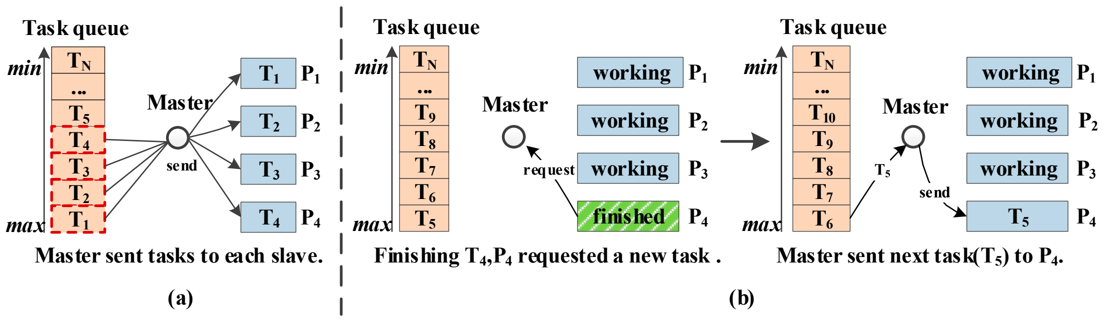

Figure 6. Dynamic scheduling based on computation quantity: T_i is the task number; P_i is the process number; (**a**) Step 2, initial assigning; and (**b**) Step 3, dynamic assigning.

3. Dynamic Assigning

Slave requested a new task from the master when the preprocessing of the current data block was completed. Next, the master retrieved the task queue. If the queue was not empty, the next task would be sent to the slave; otherwise, the NULL flag would be sent. The above procedure was repeated until the task queue was empty, as shown in Figure 6b.

4.2. Asynchronous Parallel Between Threads

Threads in the process executed the conversion computing from LiDAR points to DEM. As all threads in the same process share all points of one strip, we designed an asynchronous parallel strategy to hide the points' reading time.

The main computation of the conversion computing is spatial interpolation. To speed up the neighborhood points search during interpolation computing, we established a grid index where the main thread read points. Threads in each process needed to execute five tasks: data reading, index building, data partitioning for threads, spatial interpolation and result writing. We attributed them into two steps: data preprocessing and DEM generation. Data preprocessing contained the first three of all five tasks; where data reading was the main task; the computational complexity of this step was relatively small. Thus, this step was executed serially by the main thread. DEM generation contains spatial interpolation and result writing. Therefore, this step was executed in parallel by multiple threads. We made these two steps run asynchronously (Figure 7a): (1) the main thread created a computing thread and the computing thread was blocked; (2) the main thread preprocessed the first data strip and the computing thread was unblocked when the main thread completed its work; and (3) the computing thread created multiple threads to generate DEM from LiDAR points, while simultaneously, the main thread preprocessed the next data strip. In this way, the preprocessing time of the next data strip was hidden behind the computing procedure of the current data. As shown in Figure 7b, compared with the synchronous parallel strategy, the preprocessing time of all data strips (except the first one) was hidden behind the processing of DEM generation.

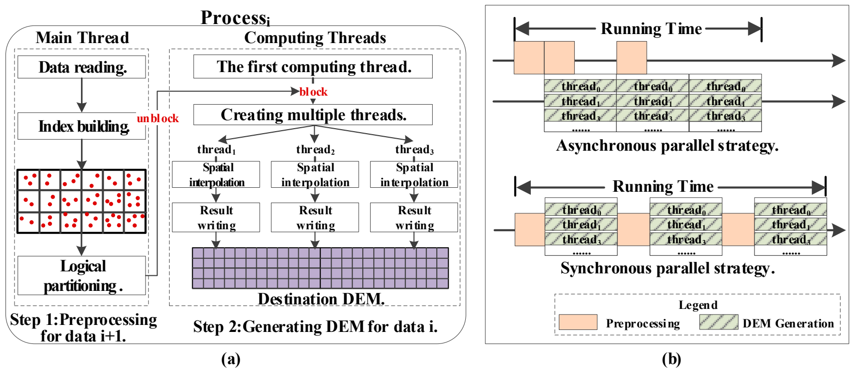

Figure 7. Asynchronous parallel between threads: (**a**) Procedure flow of the asynchronous parallel strategy; and (**b**) The running time comparison of the two strategies.

4.3. Hybrid Process/Thread Communicating Procedure

We used MPI to implement the process programming and used POSIX threads (Pthreads) to implement the thread programming. Communication was the key point of the hybrid process/thread parallel algorithm. The main thread in the slave not only communicated with the master, but also communicated with the computation threads in the same slave. Communication with the master was implemented by MPI message; communication between the threads was implemented by the condition variables of the Pthreads. We used the function named "*MPI_Init_thread*" to initialize the program, which allowed functions of the Pthreads to be called in MPI. The main communication steps were as follows (Figure 8): (1) the master sent the task by *MPI_Send* and the main thread in the slave received the task by *MPI_Recv*; (2) the main thread released the condition variable by *pthread_cond_signal* to unblock the computation thread (block by *pthread_cond_wait*) once it completed preprocessing the first data strip; (3) the main thread requested a new task from the master by *MPI_Send* ; (4) the master fed information back to the slave by *MPI_Send* and the main thread received it by *MPI_Recv*; and (5) if the feedback was NULL, the main thread modified the end tag and released the condition variable by *pthread_cond_signal*.

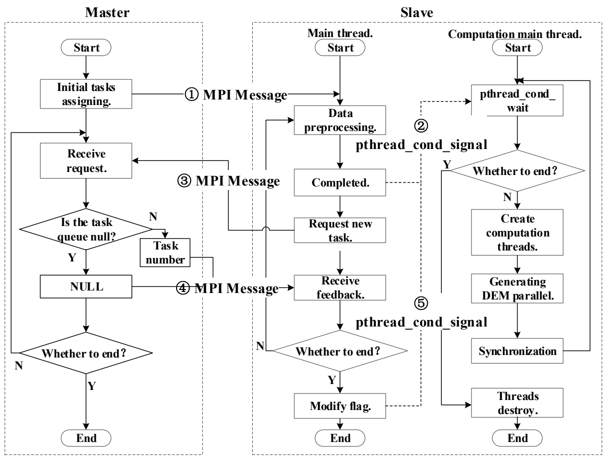

Figure 8. Communications in the scheduling procedure.

5. Experiments and Discussions

5.1. Environment and Experimental Datasets

The hybrid process/thread parallel algorithm for generating DEM from LiDAR points was implemented with standard C++ in Visual Studio 2015; Open MPI v2.0.1 was selected for the implementation of MPI; data reading of LiDAR file was implemented by libLAS 1.2; and DEM writing was implemented by the Geospatial Data Abstraction Library (GDAL 2.1.0). The parallel environment was a high-performance computing cluster that consisted of five nodes; each node had two CPUs (Intel Xeon E5-2640 v2, 2.0 GHz) with a memory of 16 GB; and each CPU contained 12 logical cores. The operation of the cluster as CentOS Linux 7.0 with the parallel file system Lustre.

We used three LiDAR datasets to test our algorithm (Table 1). Data$_1$ had 1.2 billion points; its volume was 30 GB, as shown in Figure 9a. Data$_2$ as a part extracted from Data$_1$ with 0.16 billion points with a volume of 4 GB (Figure 9b). Data$_3$ was far bigger than the other two datasets, with 60 billion points; its volume was 150 GB (Figure 9c).

Table 1. Details of the experimental datasets.

Name	Volume	Number of Points	With	Length
Data$_1$	30 GB	1.2 billion	15 km	19 km
Data$_2$	4 GB	0.16 billion	8 km	4 km
Data$_3$	150 GB	6.0 billion	77 km	141 km

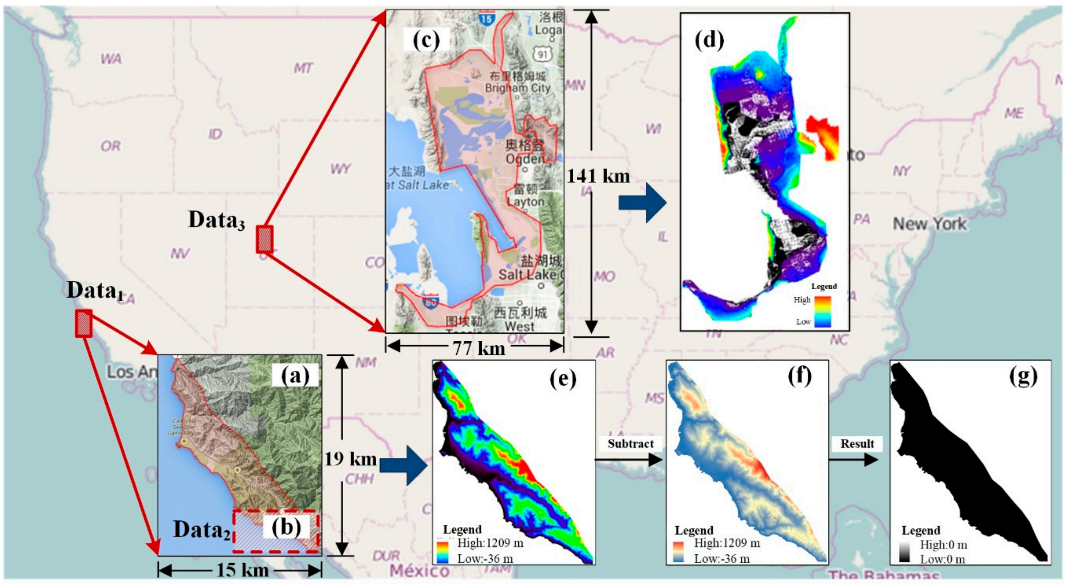

Figure 9. Locations of experimental datasets (the background basic map was from OpenStreetMap): (**a**) Location of Data$_1$; (**b**) Location of Data$_2$; (**c**) Locations of Data$_3$; (**d**) Resultant DEM of Data$_3$, generated by our algorithm; (**e**) Resultant DEM of Data$_1$, generated by our algorithm; (**f**) Resultant DEM of Data$_1$ generated by ArcGIS; and (**g**) Resultant data of (**e**) subtract (**f**).

A series of parameters was set before our experiments: the resolution of the target DEM was 1 m; the spatial interpolation method used in this research was the Inverse Distance Weighted (IDW) [32] with an initial searching radius of 10 m; the minimum number of points involved in interpolation was 10 and the maximum was 30; if the number of points initially searched was less than 10, the searching radius increased in increments of 10 m until there were more than 10 points being searched; the maximum search radius was set to 30 m.

5.2. Accuracy Analysis

In order to analyze whether our parallel algorithm would affect the accuracy of the resultant DEM, we compared the DEM generated by our parallel algorithm with the one generated by ArcGIS. We used Data$_1$ as the experimental dataset. For our parallel algorithm, the DEM was generated by the experiment detailed in Section 5.3.2, with 10 slaves and 12 threads in each slave. The interpolation parameters were set as Section 5.1. We also used IDW as the interpolation method in ArcGIS, and all interpolation parameters were same with our algorithm. The resultant DEM of our parallel algorithm was shown in Figure 9e. Figure 9f was the resultant DEM of ArcGIS. To analyze the difference between them, we used the DEM of our algorithm to subtract the one of ArcGIS, obtaining the result shown in Figure 9g. All values in the resultant dataset are 0 m, which means there was no difference between these two datasets. Although we partitioned the LiDAR dataset into multiple blocks in our parallel algorithm, DEM cells near the boundaries of each data block were not influenced. This was because the data buffer allowed each DEM cell to search all adjacent points. Thus, the hybrid parallel algorithm proposed in this paper had no influence on the accuracy of DEM.

5.3. Performance Analysis

We conducted this experiment to analyze the performance of the parallel algorithm proposed in this paper. Before we analyzed the optimal performance, we discussed two important factors that may affect the results: the number of partitioned data blocks; and the combination manner between processes and threads. For the first one, we conducted experiments at the thread and process level separately to find the optimal number of data blocks. For the second one, we fixed the total number of threads on the HPC, therefore changing the number of computation processes (slaves) and the number of threads in each process to analyze the parallel performance.

5.3.1. Optimal Number of Data Blocks

1. Blocks' Number at the Thread Level

At the thread level to find the optimal number, we tuned the number of logical partitioned blocks to test the performance of the algorithm. To eliminate the effect of multiple loops, we used Data$_2$, which could be processed by a single node at one time. We used one CPU of the node0 as the experimental environment, and created a process which had twelve computation threads, binding the process to the CPU by the command *"mpirun -1 –bind-to-socket –bysocket"*; the interpolation parameters were set as described in Section 5.1. We used N_t to represent the number of data blocks, setting N_t as 12, 24, 36, ..., 96, respectively, gathering the running time of the algorithm. The result is shown in Figure 10.

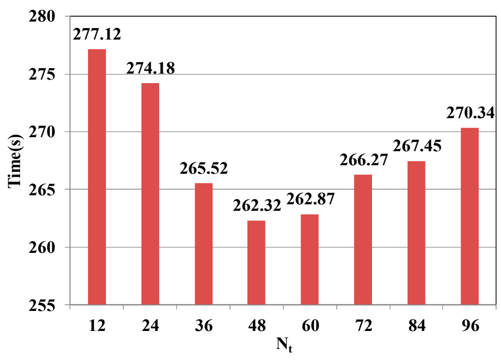

Figure 10. Running time with different N_t.

From Figure 10, it can be seen that when N_t was 12, the loop of the assigning task only executed once, leading to load unbalancing between different threads, making the running time the longest;

with the increasing of N_t, the task assigning loops multiple times made the computation load balanced, so the running time reduced gradually; when N_t was 48, it reached the shortest value; when N_t was greater than 60, the running time increased gradually. This was because too many blocks affected the continuity of computing and data accessing, thus slightly increasing the running time. Therefore, when the number of data blocks at the thread level was 4–5 times as many as the number of computation threads, the algorithm could achieve better performance.

2. Strips' Number at the Process Level

At the process level, we explored the relationship between the number of data strips (named N_p) and the performance of our algorithm. For the dataset with a volume larger than the memory capacity of the HPC, the number of strips was determined by memory capacity. Thus, in this experiment, we focused on the dataset whose volume was close to or less than the memory capacity. We used Data$_1$ as the test data, setting the number of processes as six, including one master and five slaves (each slave corresponded to one node); the number of threads in each slave was 24, as there were 24 logical cores in each node. We set N_p as 10, 20, 40, 80; the number of data blocks at the thread level was fixed at 96. Interpolation parameters were set as described in Section 5.1.

The result is shown in Figure 11 (P_i is the number of process, $i = 0, 1, \ldots, 5$): when N_p was 20, the running time was the shortest; when N_p was 10, the algorithm ran longer. This was as the number of strips was too small, leading to fewer loops during the dynamic scheduling strategy. Therefore, computation loads between different processes were unbalanced. When N_p was larger than 20, the number of loops increased and the size of data blocks decreased, which was useful for balancing computational loads. However, the number of data buffers, the volume of redundant data, and the partitioning time also increased. Thus, the running time did not decrease, but increased. Finally, we reached the following conclusion: for the dataset with a volume close to or less than memory capacity, the dynamic strategy proposed in this paper could achieved a better performance when the number of strips was about four times as many as the number of slaves.

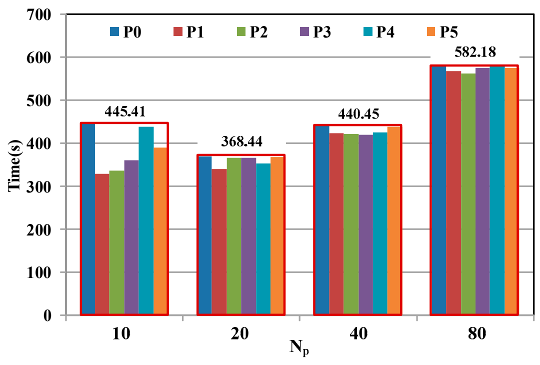

Figure 11. Running time with different N_p.

5.3.2. Optimal Combination Manner

In this experiment, we focused on finding an optimal combination manner between processes (excluding the master, and consisting of all slaves) and threads for our hybrid parallel algorithm. We fixed the total number of threads on the HPC to 120, changing the combination manner of processes and threads (we used P and T to represent the number of slaves and threads separately; and used $P \times T$ represent their combination): 5×24, 10×12, 20×6, 30×4. We used Data$_1$ as the test data. According to the experiment above, at the process level, we set N_p at four times as many as the number of slaves; we also set N_t to four times as many as the number of computation threads. Interpolation parameters remained unchanged.

The result is shown in Figure 12. When the combination manner was 10×12, the running time was the shortest. With other combining manners, the algorithm ran longer as there were 10 CPUs on the HPC. When P was less than 10 (5×24), the process switching between CPUs in one node affected the efficiency; when P was larger than 10, although there was no switching cost, the increase in processes led to an increase in the number of strips, which made data buffers and redundant data readings increase, leading to the longer running time. Thus, for our hybrid parallel strategy, we obtained better parallel results when the number of computation process was equal to the number of CPUs.

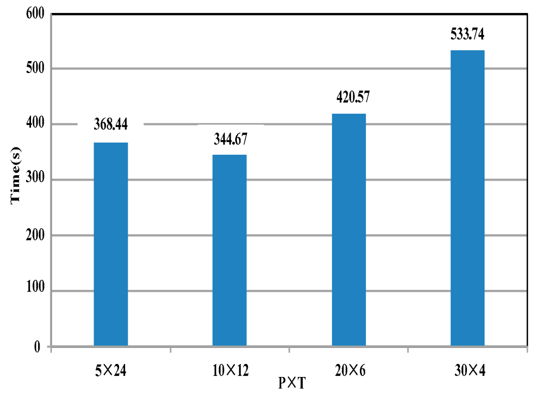

Figure 12. Running time of different combinations of slaves and threads.

The resultant DEM is shown in Figure 9d. The running time in the serial state was 112,486 s (this is not shown in Figure 13 as the column was too high.). It decreased with the number of increasing threads and the speedup increased gradually (Figure 13). The execution time reached its minimum value with 120 threads (2342 s) and the speedup reached its maximum value (48). With more than 120 threads, the running time tended to increase gradually due to the computation limitation of the hardware (a total of 120 logical cores in our HPC). This indicated that the parallel algorithm showed good scalability with an increase in computation resources.

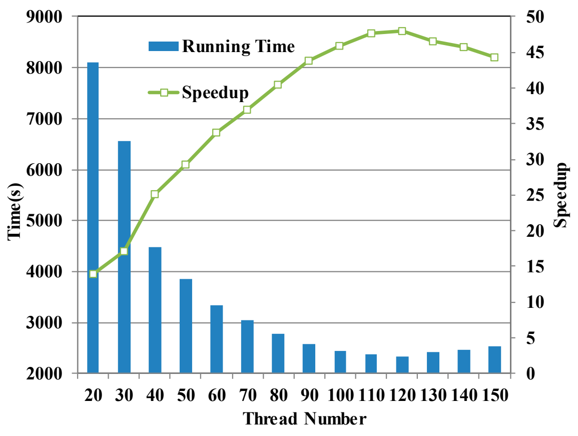

Figure 13. Running time and speedup of Data$_3$.

5.3.3. Performance Analysis with the Largest Experimental Dataset

After finding the optimal number of data blocks and the optimal combination manner between processes and threads, we tested the performance of our hybrid parallel algorithm by using it to generate DEM from the largest experimental LiDAR dataset (Data$_3$). In this experiment, we set the number of slaves as 10; the initial number of threads in each slave was 2 and increased by 1 each time. Considering the memory capacity and the experiment of Section 5.3.1, we partitioned the data into 50 strips; the volume of each strip was approximately 3 GB. The data buffer was 40 m and the interpolation parameters were set as per Section 5.1. Besides the running time, we use the speedup [14] as the other evaluation criterion for the algorithm. It is expressed as the ratio of T_s (the running time for the serial algorithm) to T_p (the running time for the parallel algorithm), Equation (8).

$$speedup = T_s / T_p \tag{8}$$

5.4. Comparative Experiments

5.4.1. Comparison with the Static Schedule Strategy

To analyze the effectiveness of the new dynamic load balancing strategy proposed in this paper, we statistically obtained the running time of each process in the experiment described in Section 5.3.3 when the speedup was at its peak. We also statistically obtained the number of points processed by each process. We compared them with values obtained from the static scheduling strategy. The static scheduling strategy assigned task circularly as below: strips whose numbers were 1, 11, and 21, . . . , 51 were assigned to process 1; strips 2, 12 and 22, . . . , 52 were assigned to process 2; and all strips were assigned before the algorithm was run. Other parameters remained unchanged. The results are shown in Figure 14. In the static algorithm, the longest execution time was 2656 s (No. 5 process), with the most points (0.873 billion). The shortest execution time was 1804 s (No. 1 process). The time difference was about 852 s. In the dynamic strategy, the longest time was 2342 s (No. 0 process) and the shortest time was 2010 s (No. 1 process). The time difference was 332 s, which was smaller than the static one. Furthermore, the time curve of the dynamic strategy was more stable than the static one; the same was true of the point quantity. This indicated that the dynamic scheduling strategy based on computation quantity could achieve a better load balancing than the traditional static strategy.

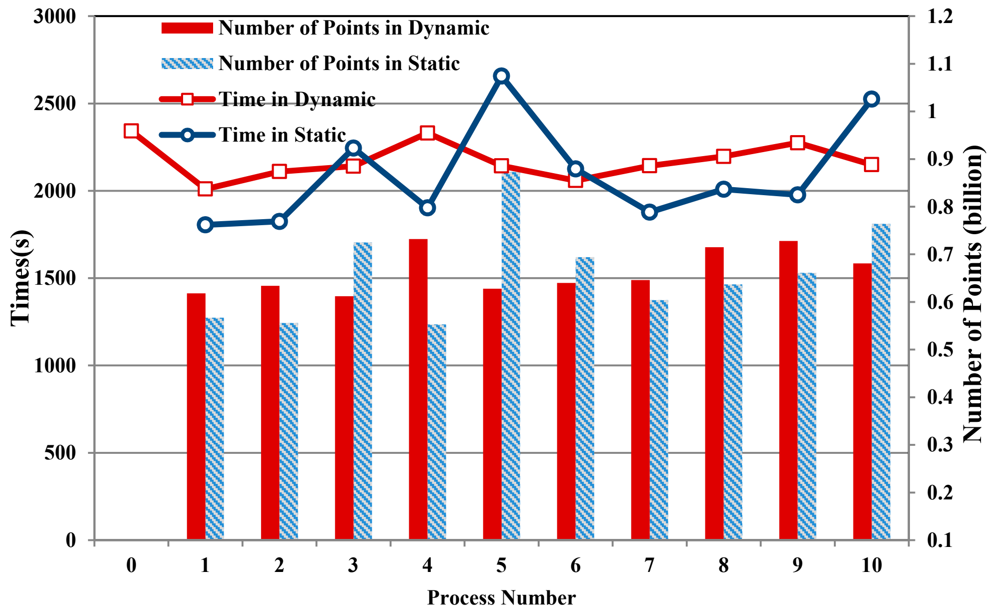

Figure 14. Load balancing information of two scheduling strategies.

5.4.2. Comparison with the Synchronous Parallel Strategy

We compared our parallel algorithm with the synchronous algorithm. The procedure of the synchronous algorithm is shown as Figure 7b, whose preprocessing step did not overlap the generating DEM step. For our asynchronous parallel algorithm, we used the experiment in Section 5.3.3. For the synchronous algorithm, as it does not need to cache another data strip when the DEM was generating, we partitioned Data$_3$ into 40 strips, which made the volume of each strip slightly bigger. The strategy at the process level and the interpolation parameters remained unchanged. The results are shown in Figure 15. The running time of the asynchronous algorithm was longer than the synchronous one when the number of threads was small as the main thread in the asynchronous algorithm did not participate in the computation. With an increase of threads, the time difference decreased, which is where the advantage of the asynchronous strategy appeared. Finally, the minimum time of the synchronous algorithm was 2878 s. For the asynchronous algorithm, it was 2342 s and the efficiency was improved by 18.6%. Compared with the synchronous algorithm, although our algorithm added 10 strips, which generated more data buffers and redundant data, their influence was almost negligible by hiding the preprocessing time. This proved that our asynchronous parallel strategy was effective.

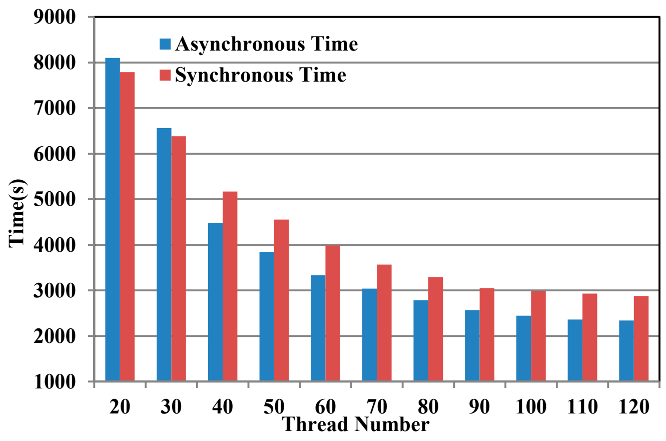

Figure 15. Running time of two strategies at the thread level.

5.4.3. Comparison with Process Parallel Algorithm

We made a comparative experiment between the hybrid process/thread parallel algorithm (HPA) proposed in this paper and the traditional process parallel algorithm (PPA). The data partitioning methods and scheduling strategies at the process level of the two algorithms remained the same. However, for PPA, each process was the unit to read and generate the DEM. The number of processes in the PPA was equal to the number of all threads in the HPA. For HPA, the initial number of threads in each process was 2, which was increased by step 1. For PPA, the initial number of processes as 21 (No. 0 is the master.) and it increased by 10 each time. The number of strips in HPA was fixed to 50. For PPA, it varied dynamically, maintaining four times as many as the number of slaves, which ensured load balancing while minimizing the number of strips.

As shown in Figure 16, the running time of HPA was shorter than the time of PPA; the increasing speed of the speedup of HPA was faster than that of PPA. There were two reasons for this phenomenon. First, HPA improved the efficiency of the input of LiDAR points by reducing the number of data buffers. Figure 17 shows the input time of LiDAR points in two algorithms, which were extracted from the whole time of each algorithm. It contained two parts: the time of parallel data partitioning by PPDB, and time of data reading by each process. As Figure 17a shows, the overall data inputting time was about 7000 s in a serial state, with each part about 3500 s. When the number of threads increased to 20, it decreased to approximately 1000 s, with each part about 500 s (Figure 17c). After that, they maintained stable and did not increase with an increase in threads. However, for PPA, the time of both parts increased with the increase in processes (Figure 17b). This is as the number of data buffers in PPA increased obviously with the increasing of data strips, which made a lot of redundant data reading. Furthermore, when different processes read LiDAR points stored in the same file simultaneously, data reading competition increased the reading time, especially for a larger number of processes. The data partitioning time increased with the increase in strips and the increase of reading requests. However, in the HPA, the number of data buffers remained unchanged; parallel inputting requests were no more than 10 constantly, which made the input time remain steady. Thus, the speedup of HPA increased faster than that in PPA. Second, the asynchronous parallel strategy in HPA effectively hid the time of reading points as the time of reading points in Figure 17c overlapped with computation time, which further increased the speedup of HPA.

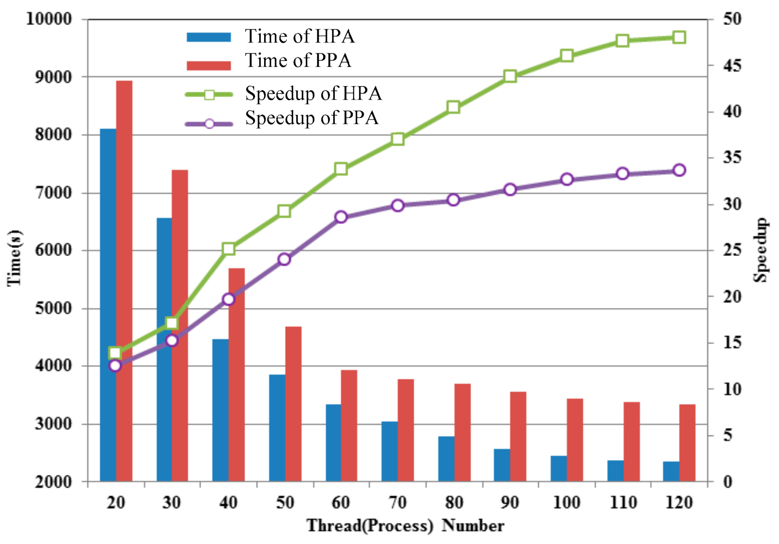

Figure 16. Running time and speedup of two algorithms.

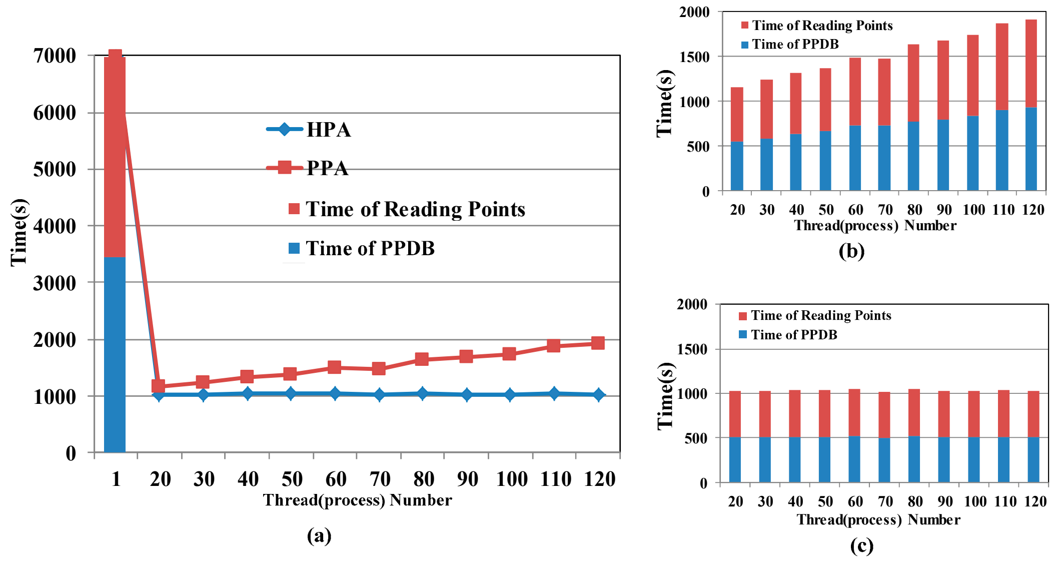

Figure 17. LiDAR input time in two algorithms: (**a**) the overall input time of hybrid process/thread parallel algorithm (HPA) and process parallel algorithm (PPA); (**b**) the detailed time of PPA; and (**c**) the detailed time of HPA.

Finally, the minimum execution time of PPA was 3347 s; for HPA, it was 2342 s. The efficiency improved by 30%. Compared with PPA, the HPA proposed in this research could affect a more efficient conversion from LiDAR points to DEM, in particular, an increase of computing resources, showed that the HPA had better computational scalability than the PPA.

6. Discussion

This section primarily discusses the feasibility of the proposed hybrid parallel algorithm to different types of datasets and applications. On one hand, our test datasets included multiple terrain types: Data$_1$ was representative of a hilly area and Data$_3$ was a mixed terrain of flat and hilly areas. This indicated that our hybrid parallel algorithm was well suited for DEM generation from different types of LiDAR datasets. One the other hand, parallel generation of DEMs from LiDAR points is a typical representation of the data-intensive and computation-intensive procedure of a LiDAR dataset. It included points' partitioning, data reading, task scheduling, main computing, and result-writing. Generally, most of these steps can be used in other applications of LiDAR datasets, such as parallel Triangulated Irregular Network (TIN) construction and parallel LiDAR point filtering. When we apply our hybrid parallel strategy to other LiDAR applications, we only need to replace the spatial interpolation with the main computing steps of other applications. Thus, the hybrid process/thread parallel strategy proposed in this research will be useful for people who want to accelerate the processing efficiencies of their LiDAR datasets. In the future, we will package all steps of the hybrid parallel strategy, forming a set of tools that can be easily used by more LiDAR applications.

7. Conclusions

This research analyzed the parallel load balancing and the boundary problems existing in generating DEM from LiDAR points. To solve these two problems, a hybrid process/thread parallel algorithm was proposed. At the process level, we designed a parallel data partitioning method (PPDB) and a dynamic scheduling strategy based on computation quantity to achieve load balancing. At the thread level, we partitioned the data block stored in the memory logically. To reduce the impact of data reading further, we proposed an asynchronous parallel strategy between threads to hide the reading time of LiDAR points. Finally, we used OpenMPI 2.0.1 to implement the programming at the process level and used Pthreads to implement the programming at the thread level. In a HPC, we tested our hybrid parallel algorithm using three LiDAR datasets of different volumes. We discussed the relationship between the number of portioned data blocks and performance of our algorithm and also analyzed how combining processes with threads could achieve the best performance. After that, we used a larger dataset to test the performance of our algorithm; furthermore, we analyzed the effect of dynamic scheduling strategy based on computation quantity and the effect of asynchronous parallel strategy between the threads, respectively. Finally, we compared the performance of the hybrid process/thread parallel algorithm (HPA) proposed in this paper with that of the traditional process parallel algorithm (PPA) and analyzed the differences between them, explaining those reasons in detail.

Through the above-mentioned experiments, we reached three conclusions: (1) compared with PPA, the HPA proposed in this research improved the converting efficiency from LiDAR points to DEM by making full use of the advantages in process-paralleling and the advantages in thread-paralleling; (2) the PPDB was parallelizable and the dynamic scheduling strategy based on computation quantity combined with PPDB achieved a better load balancing; and (3) the asynchronous parallel strategy between threads further reduced the impact of reading LiDAR points on the algorithm's efficiency.

Acknowledgments: This work is supported by the Science and Technology Project of Qingdao (Grant No. 16-6-2-61-NSH).

Author Contributions: Yibin Ren designed and implemented the algorithm, did the experiments, wrote the paper. Associate professor Zhenjie Chen and Professor Yong Han conducted the experiments, helped Yibin Ren to analyze experimental results. Professor Ge Chen read and modified the paper. Yanjie Wang drew the result figures of experiments.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Jones, K.H. A comparison of algorithms used to compute hill slope as a property of the DEM. *Comput. Geosci.* **1998**, *24*, 315–323. [CrossRef]
2. Floriani, L.D.; Magillo, P. Digital Elevation Models. In *Encyclopedia of Database Systems*; Liu, L., ÖZSU, M.T., Eds.; Springer: Boston, MA, USA, 2009; pp. 817–821.
3. Lloyd, C.D.; Atkinson, P.M. Deriving ground surface digital elevation models from LiDAR data with geostatistics. *Int. J. Geogr. Inf. Sci.* **2006**, *20*, 535–563. [CrossRef]

4. White, S.A.; Wang, Y. Utilizing DEMs derived from LIDAR data to analyze morphologic change in the North Carolina coastline. *Remote Sens. Environ.* **2003**, *85*, 39–47. [CrossRef]
5. Ma, R.J. DEM generation and building detection from Lidar data. *Photogramm. Eng. Remote Sens.* **2005**, *71*, 847–854. [CrossRef]
6. Hu, X.Y.; Li, X.K.; Zhang, Y.J. Fast Filtering of LiDAR Point Cloud in Urban Areas Based on Scan Line Segmentation and GPU Acceleration. *IEEE Geosci. Remote Sens.* **2013**, *10*, 308–312.
7. Ma, H.C.; Wang, Z.Y. Distributed data organization and parallel data retrieval methods for huge laser scanner point clouds. *Comput. Geosci.* **2011**, *37*, 193–201.
8. Wang, Y.; Chen, Z.; Cheng, L.; Li, M.; Wang, J. Parallel scanline algorithm for rapid rasterization of vector geographic data. *Comput. Geosci.* **2013**, *59*, 31–40. [CrossRef]
9. Chen, C.; Chen, Z.; Li, M.; Liu, Y.; Cheng, L.; Ren, Y. Parallel relative radiometric normalisation for remote sensing image mosaics. *Comput. Geosci.* **2014**, *73*, 28–36. [CrossRef]
10. Guan, Q.; Kyriakidis, P.C.; Goodchild, M.F. A parallel computing approach to fast geostatistical areal interpolation. *Int. J. Geogr. Inf.* **2011**, *25*, 1241–1267. [CrossRef]
11. Zhao, L.; Chen, L.; Ranjan, R.; Choo, K.R.; He, J. Geographical information system parallelization for spatial big data processing: A review. *Clust. Comput.* **2016**, *19*, 139–152. [CrossRef]
12. Liu, J.; Zhu, A.X.; Liu, Y.; Zhu, T.; Qin, C.Z. A layered approach to parallel computing for spatially distributed hydrological modeling. *Environ. Modell. Softw.* **2014**, *51*, 221–227. [CrossRef]
13. Guan, X.; Wu, H. Leveraging the power of multi-core platforms for large-scale geospatial data processing: Exemplified by generating DEM from massive LiDAR point clouds. *Comput. Geosci.* **2010**, *36*, 1276–1282. [CrossRef]
14. Huang, F.; Liu, D.; Tan, X.; Wang, J.; Chen, Y.; He, B. Explorations of the implementation of a parallel IDW interpolation algorithm in a Linux cluster-based parallel GIS. *Comput. Geosci.* **2011**, *37*, 426–434. [CrossRef]
15. Han, S.H.; Heo, J.; Sohn, H.G.; Yu, K. Parallel Processing Method for Airborne Laser Scanning Data Using a PC Cluster and a Virtual Grid. *Sensors-Basel* **2009**, *9*, 2555–2573. [CrossRef] [PubMed]
16. Danner, A.; Breslow, A.; Baskin, J.; Wilikofsky, D. Hybrid MPI/GPU interpolation for grid DEM construction. In Proceedings of the International Conference on Advances in Geographic Information Systems, Redondo Beach, CA, USA, 6–9 November 2012; pp. 299–308.
17. Huang, F.; Bu, S.; Tao, J.; Tan, X. OpenCL Implementation of a Parallel Universal Kriging Algorithm for Massive Spatial Data Interpolation on Heterogeneous Systems. *ISPRS Int. J. Geo-Inf.* **2016**, *5*, 96. [CrossRef]
18. Cappello, F.; Richard, O.; Etiemble, D. Understanding performance of SMP clusters running MPI programs. *Future Gener. Comput. Syst.* **2001**, *17*, 711–720. [CrossRef]
19. Yang, C.T.; Huang, C.L.; Lin, C.F. Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Comput. Phys. Commun.* **2011**, *182*, 266–269. [CrossRef]
20. Wu, X.; Taylor, V. Performance modeling of hybrid MPI/OpenMP scientific applications on large-scale multicore supercomputers. *J. Comput. Syst. Sci.* **2013**, *79*, 1256–1268. [CrossRef]
21. Prasad, S.K.; Mcdermott, M.; Puri, S.; Shah, D.; Aghajarian, D.; Shekhar, S.; Zhou, X. A vision for GPU-accelerated parallel computation on geo-spatial datasets. *Sigspatial Spec.* **2015**, *6*, 19–26. [CrossRef]
22. Lee, V.W.; Kim, C.; Chhugani, J.; Deisher, M.; Kim, D.; Nguyen, A.D.; Satish, N.; Smelyanskiy, M.; Chennupaty, S.; Hammarlund, P. Debunking the 100X GPU vs. CPU myth:an evaluation of throughput computing on CPU and GPU. *Acm Sigarch Comput. Arch. News* **2010**, *38*, 451–460. [CrossRef]
23. Zuo, Y.; Wang, S.; Zhong, E.; Cai, W. Research Progress and Review of High-Performance GIS. *J. Geo-Inf. Sci.* **2017**, *19*, 437.
24. Huang, F.; Zhou, J.; Tao, J.; Tan, X.; Liang, S.; Cheng, J. PMODTRAN: A parallel implementation based on MODTRAN for massive remote sensing data processing. *Int. J. Digit. Earth* **2016**, *9*, 819–834. [CrossRef]
25. Chatzimilioudis, G.; Costa, C.; Zeinalipouryazti, D.; Lee, W.C.; Pitoura, E. Distributed in-memory processing of All K Nearest Neighbor queries. In Proceedings of the IEEE International Conference on Data Engineering, Helsinki, Finland, 2016; pp. 1490–1491.
26. Dong, B.; Li, X.; Wu, Q.; Xiao, L.; Li, R. A dynamic and adaptive load balancing strategy for parallel file system with large-scale I/O servers. *J. Parallel Distrib. Comput.* **2012**, *72*, 1254–1268. [CrossRef]
27. Qian, C.; Dou, W.; Yang, K.; Tang, G. Data Partition Method for Parallel Interpolation Based on Time Balance. *Geogr. Geo-Inf. Sci.* **2013**, *29*, 86–90.

28. Qi, L.; Shen, J.; Guo, L.; Zhou, T. Dynamic Strip Partitioning Method Oriented Parallel Computing for Construction of Delaunay Triangulation. *J. Geo-Inf. Sci.* **2012**, *14*, 55–61. [CrossRef]
29. Ismail, Z.; Khanan, M.F.A.; Omar, F.Z.; Rahman, M.Z.A.; Salleh, M.R.M. Evaluating Error of LiDar Derived DEM Interpolation for Vegetation Area. *Int. Arch. Photogramm. Remote Sens. S* **2016**, *XLII-4/W1*, 141–150. [CrossRef]
30. Guan, X.; Wu, H.; Li, L. A Parallel Framework for Processing Massive Spatial Data with a Split-and-Merge Paradigm. *Trans. GIS* **2012**, *16*, 829–843. [CrossRef]
31. Zou, Y.; Xue, W.; Liu, S. A case study of large-scale parallel I/O analysis and optimization for numerical weather prediction system. *Future Gener. Comput. Syst.* **2014**, *37*, 378–389. [CrossRef]
32. Lu, G.Y.; Wong, D.W. An adaptive inverse-distance weighting spatial interpolation technique. *Comput. Geosci.* **2008**, *34*, 1044–1055. [CrossRef]

© 2017 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).