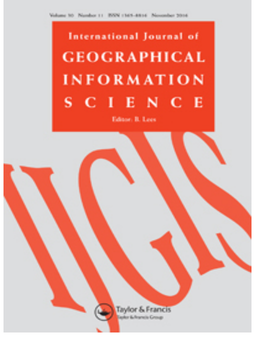

International Journal of Geographical Information Science

ISSN: 1365-8816 (Print) 1362-3087 (Online) Journal homepage: http://www.tandfonline.com/loi/tgis20

Enabling point pattern analysis on spatial big data using cloud computing: optimizing and accelerating Ripley's K function

Guiming Zhang, Qunying Huang, A-Xing Zhu & John H. Keel

To cite this article: Guiming Zhang, Qunying Huang, A-Xing Zhu & John H. Keel (2016) Enabling point pattern analysis on spatial big data using cloud computing: optimizing and accelerating Ripley's K function, International Journal of Geographical Information Science, 30:11, 2230-2252, DOI: 10.1080/13658816.2016.1170836

To link to this article: http://dx.doi.org/10.1080/13658816.2016.1170836

Published online: 11 Apr 2016.

INTERNATIONAL JOURNAL OF GEOGRAPHICAL INFORMATION SCIENCE, 2016
VOL. 30, NO. 11, 2230–2252
http://dx.doi.org/10.1080/13658816.2016.1170836

Enabling point pattern analysis on spatial big data using cloud computing: optimizing and accelerating Ripley's *K* function

Guiming Zhang[a], Qunying Huang[a], A-Xing Zhu[b,c,d,e,a] and John H. Keel[f]

[a]Department of Geography, University of Wisconsin-Madison, Madison, WI, USA; [b]Key Laboratory of Virtual Geographic Environment, Nanjing Normal University, Ministry of Education, Nanjing, China; [c]State Key Laboratory Cultivation Base of Geographical Environment Evolution, Nanjing, China; [d]State Key Laboratory of Resources and Environmental Information System, Institute of Geographical Sciences and Natural Resources Research, Chinese Academy of Sciences, Beijing, China; [e]Jiangsu Center for Collaborative Innovation in Geographical Information Resource Development and Application, Nanjing, China; [f]Division of Information Technology, University of Wisconsin-Madison, Madison, WI, USA

ABSTRACT

Performing point pattern analysis using Ripley's *K* function on point events of large size is computationally intensive as it involves massive point-wise comparisons, time-consuming edge effect correction weights calculation, and a large number of simulations. This article presented two strategies to optimize the algorithm for point pattern analysis using Ripley's *K* function and utilized cloud computing to further accelerate the optimized algorithm. The first optimization sorted the points on their *x* and *y* coordinates and thus narrowed the scope of searching for neighboring points down to a rectangular area around each point in estimating *K* function. Using the actual study area in computing edge effect correction weights is essential to estimate an unbiased *K* function, but is very computationally intensive if the study area is of complex shape. The second optimization reused the previously computed weights to avoid repeating expensive weights calculation. The optimized algorithm was then parallelized using Open Multi-Processing (OpenMP) and hybrid Message Passing Interface (MPI)/OpenMP on the cloud computing platform. Performance testing showed that the optimizations effectively accelerated point pattern analysis using *K* function by a factor of 8 using both the sequential version and the OpenMP-parallel version of the optimized algorithm. While the OpenMP-based parallelization achieved good scalability with respect to the number of CPU cores utilized and the problem size, the hybrid MPI/OpenMP-based parallelization significantly shortened the time for estimating *K* function and performing simulations by utilizing computing resources on multiple computing nodes. Computational challenge imposed by point pattern analysis tasks on point events of large size involving a large number of simulations can be addressed by utilizing elastic, distributed cloud resources.

ARTICLE HISTORY
Received 19 October 2015
Accepted 21 March 2016

KEYWORDS
Point pattern analysis; Ripley's *K* function; optimization; MPI/OpenMP; geospatial cloud computing

CONTACT A-Xing Zhu ✉ azhu@wisc.edu

The underlying C++ computer program and test data sets for this article can be accessed at https://goo.gl/0nBi7y.

1. Introduction

Point pattern analysis is the study of the spatial arrangement of point events (i.e., a set of locations over which certain geographic phenomenon of interest occurs) in space (Diggle 1983, Illian *et al*. 2008). It identifies the pattern (i.e., dispersed, clustered, or random) underlying the spatial distribution of the geographic phenomenon and thus can inform formulating hypotheses to investigate the phenomenon-generating mechanisms and processes. Point pattern analysis is a useful analytical tool that has been widely applied in ecology, epidemiology, crime pattern analysis, economics, etc. (Fotheringham *et al*. 2000, Law *et al*. 2009). Among the many methods that have been developed to approach point pattern analysis (see Fotheringham *et al*. 2000, Illian *et al*. 2008, Burt *et al*. 2009 for reviews), Ripley's *K* function has advantages over other alternatives. For instance, the quadrat analysis method is sensitive to the parameter *quadrat size* used to define grids within which the number of points is counted. Different quadrat sizes might lead to different conclusions on the same point data set (Fotheringham *et al*. 2000, Burt *et al*. 2009). The kernel estimation method requires a similar parameter: *bandwidth*. The determination of quadrat size or bandwidth mostly relies on experience and therefore is rather subjective. Ripley's *K* function, on the other hand, does not require such parameters. The nearest neighbor distance method considers only the nearest neighbor and ignores spatial dependencies between points beyond the nearest neighbor and thus reflects only the shortest scale of spatial variation. Ripley's *K* function, however, provides an estimate of spatial dependence over a much wider range of scales and considers all the distances between point events in the study area.

Spatial big data are now ubiquitous (Shekhar *et al*. 2012, Evans *et al*. 2013) and point pattern analysis on such spatial big data sets is needed for scientific or pragmatic purposes. With the emergence and prosperity of social media, location-based services, citizen science projects, etc., the general public are generating and contributing an unprecedentedly large volume of volunteered geographic information (Goodchild 2007). Much of this data can be perceived as point events. Examples are tourist attractions from OpenStreetMap (Haklay and Weber 2008), birding checklists from eBird (Wood *et al*. 2011), and geo-tagged posts from social media such as Twitter. Point pattern analysis on such data sets is of interest to many audiences. Ornithologists model the geographic distribution pattern of bird species using eBird records (Wood *et al*. 2011). Amateur birders identify birding hotspots by examining spatial pattern of other birders' birding sites. Researchers mine spatial pattern of people's tweeting locations in disasters to provide decision support for disaster relief (Gao *et al*. 2011).

Ripley's *K* function approach for point pattern analysis is computationally challenging especially on point events of large sizes. Existing software packages for point pattern analysis with Ripley's *K* function (e.g., Splancs and Spatstat in R; see Rowlingson and Diggle 1993, Baddeley and Turner 2005) are limited to handle only hundreds or thousands of points. In this big data era, point data sets however, usually include millions of points (e.g., eBird checklist). The acceleration of Ripley's *K* function is urgently needed to enable point pattern analysis on spatial big data.

High-performance computing (HPC) has been used to address the computational challenges posed by GIScience applications (Yang *et al*. 2010, Wright and Wang 2011, Wang 2013). Parallel programming models such as OpenMP (Open Multi-Processing), MPI (Message Passing Interface), CUDA (Compute Unified Device Architecture), and MapReduce are widely adopted to make full use of the computing power of multiprocessor CPUs (central processing units) and massively parallel GPUs (graphic processing units) to accelerate geoprocessing and spatial analysis (Mineter *et al*. 2000, Stojanovic and Stojanovic 2013, Gao *et al*. 2014, Pijanowski *et al*. 2014, Tang *et al*. 2015). Yet few studies focused on utilizing HPC for point pattern analysis although a massive parallelization method was developed recently to accelerate *K* function using the many-core architecture GPUs (Tang *et al*. 2015). Computation tasks of estimation *K* function on a large point data set were scheduled to many GPU cores to compute *K* function in parallel. Monte Carlo simulations were distributed to GPUs interconnected in a HPC cluster to accelerate the statistical significance testing (Tang *et al*. 2015). However, building an HPC system, especially a GPU-based HPC cluster, is costly and time-consuming. A large financial investment and several weeks or even months are required to purchase the servers and configure the hardware and software while building up a middle-scale HPC system (Huang *et al*. 2013a). Consequently, only a few organizations can afford or have access to these expensive computing facilities. Besides, such systems are also difficult and expensive to maintain and operate.

In recent years there has been an explosion of interest in using cloud computing to access computing resources (Yang *et al*. 2011, Huang *et al*. 2013b). Cloud computing, an emerging computing paradigm with the capability of provisioning on-demand computing resources, overcomes some shortcomings of traditional HPC. Cloud computing releases the user from cost and effort on purchasing and maintaining the physical infrastructure. It is built on virtualization technology and this enables cloud service providers (e.g., Amazon EC2, Google App Engine, and Microsoft Azure) offer users elastic and scalable computing resources in a pay-as-you-go manner. Users request computing resources that match the computing task at hand, and could be flexibly scaled up or down as computing task changes. These features make cloud computing very appealing and it has been utilized for geoprocessing and spatial big data analytics (Wang *et al*. 2009, Li *et al*. 2011, Shekhar *et al*. 2012). Applications include retrieving and indexing spatial data (Wang *et al*. 2009), processing intensive floating car data for urban traffic surveillance (Li *et al*. 2011), running coupled atmosphere-ocean climate models (Evangelinos and Hill 2008), and supporting dust storm forecasting (Huang *et al*. 2013a). Yet to the best of our knowledge there is no study that has been conducted to exploit cloud computing for point pattern analysis on spatial big data.

This study exploited cloud computing to accelerate point pattern analysis with Ripley's *K* function. A series of strategies were designed to optimize the algorithm for estimating *K* function. The optimized algorithm was then parallelized in a cloud computing environment using two programming models: OpenMP and hybrid MPI/OpenMP. The overall goal was to enable point pattern analysis on spatial big data with Ripley's *K* function accelerated utilizing cloud computing. Section 2 introduced Ripley's *K* function and analyzed the algorithmic complexity of estimating *K* function. Section 3 presented the basic idea for optimizing the algorithm for estimating *K* function, implementations of the optimizations, and parallelization of *K* function. A description of the cloud computing platform utilized to test the algorithm was presented in Section 4. Effectiveness of the optimizations and efficiency of the parallelization were then evaluated in Section 5. This article ends with conclusions in Section 6.

2. Ripley's *K* function for point pattern analysis

For an isotropic, stationary process, Ripley's *K* function is defined as (Ripley 1988):

$$K(h) = \frac{1}{\lambda}E(h) \tag{1}$$

where λ is the intensity of point events and $E(h)$ is the expected number of point events within distance *h*. *K* function is estimated by:

$$\hat{K}(h) = \frac{A}{n \cdot n}\sum\sum_{i \neq j}\frac{I_h(d_{ij})}{w_{ij}} \tag{2}$$

where *A* is the area of the study area; *n* is the total number of points; d_{ij} is the distance between points *i* and *j; $I_h(d_{ij}) = 1$* if $d_{ij} \leq h$, otherwise $I_h(d_{ij}) = 0$; w_{ij} is a weighting function that corrects edge effect. *K* function without edge effect correction is biased and should not be used for data analysis (Baddeley and Turner 2005). Ripley's isotropic correction was adopted for edge effect correction in this study, which defines w_{ij} as the proportion of circumference that lies within the study area on a circle of radius d_{ij} centered on point *i* (Ripley 1988). Isotropic edge effect correction weights were calculated using the discrete Green formula (Baddeley *et al.* 2015, A. Baddeley, personal communication, 26 March 2015). If a point pattern follows the property of complete spatial randomness (CSR), $\hat{K}(h)$ is expected to be πh^2. Thus, equivalently, whether a point pattern is CSR or not can be determined by examining:

$$\hat{L}(h) = \sqrt{\hat{K}(h)/\pi} \tag{3}$$

A point pattern is CSR if $\hat{L}(h) = h$ (i.e., the expected $L(h)$), clustered if $\hat{L}(h) > h$, and dispersed if $\hat{L}(h) < h$.

A large number of simulations are required to obtain empirical distribution of the statistics to test whether an observed point pattern is significantly different from CSR (Diggle 1983, Fotheringham *et al.* 2000). Monte Carlo and bootstrapping are two common approaches for simulations. The Monte Carlo approach generates many realizations of CSR (e.g., 1000 realizations) and the expected $L(h)$ is estimated on each CSR realization (Baddeley and Turner 2005). Confidence intervals constructed based on $L(h)$ estimated on these CSR realizations are then compared to the $\hat{L}(h)$ estimated on the observed point pattern to determine whether its departure from CSR is statistically significant. The bootstrapping approach, which was adopted in this article, takes a different strategy. It repeatedly resamples the point events randomly with replacement and generates bootstrap samples (e.g., 1000 bootstrap samples) of the underlying population of the observed point pattern (Fotheringham *et al.* 2000). $\hat{L}(h)$ is estimated on each bootstrap sample and confidence intervals of $\hat{L}(h)$ are then compared to the L (h) expected on CSR to determine its statistical significance.

Point pattern analysis using *K* function is computationally intensive. For a particular distance h_0 under consideration, it takes $O(n^2)$ point-wise comparisons to compute $\hat{L}(h_0)$ where *n* is the number of points. In each point-wise comparison, the edge effect correction weight is calculated based on geometric arrangement of the two points and the study area boundary. The complexity of computing $\hat{L}(h_0)$ then becomes *O*

($m{\cdot}n^2$) where m is the number of vertices on the study area boundary. $\hat{L}(h)$ is estimated over a series of distances and the complexity becomes $O(v{\cdot}m{\cdot}n^2)$ where v is the number of distinct distances. Simulations required for statistical significance testing adds to the complexity. Let s be the number of simulations runs, the complexity of point pattern analysis using K function becomes $O(s{\cdot}v{\cdot}m{\cdot}n^2)$. As a result, point pattern analysis using Ripley's K function on large point data sets in the study areas of complex geometric shape (i.e., the boundary is composed of many vertices) at various spatial scales is extremely slow. Ripley's K function needs to be accelerated to enable point pattern analysis on spatial big data.

3. Optimizing and parallelizing *K* function

3.1. *The basic idea*

Computing $\hat{L}(h_0)$ at a given distance h_0 requires nested traversals over the n points. The outer traversal goes through each of the n points. For each point P_i traversed, another traversal (i.e., the inner traversal) over the n points is needed to count the number of points within distance h_0 from P_i (i.e., number of points located in a circle centered at P_i of radius h_0). Obviously, in this inner traversal, checking points that are way further than distance h_0 from P_i is not necessary; examining only those points that could potentially be in the circular neighborhood of radius h_0 around P_i is sufficient. Admittedly, if the n points are not in any order, it is indeed impossible to determine whether a point is within circular neighborhood of P_i unless its distance to P_i is calculated and compared to h_0. However, if points are properly sorted on their x and y coordinates, the inner traversal can be confined to a rectangular area around P_i. In this case, unnecessary distance calculation between P_i and points outside the rectangular area can be avoided (Figure 1). In other words, computation involved in each inner traversal is largely reduced. The complexity of point pattern analysis using K function, $O(s{\cdot}v{\cdot}m{\cdot}n^2)$, could be optimized to $O(s{\cdot}v{\cdot}m{\cdot}n{\cdot}n')$ where n' is the average number of points checked in an inner traversal. Foreseeably, n' can be much smaller than n especially for a short distance h_0.

In the inner traversal, it is actually not simply counting the number of points. The edge effect correction weight needs to be computed for each point P_j in the circular neighborhood of P_i. Isotropic correction defines the weight w_{ij} as the proportion of circumference within the study area on a circle centered on P_i of radius d_{ij}. Computing the weight involves complicated geometric operations involving P_i, P_j, and the boundary of the study area. This is computationally expensive if the polygonal study area is of a complex shape. However, it is noteworthy that a weight, once computed, could be reused to avoid repeating the expensive computation (Figure 1). An edge effect correction weight can be reused in two cases. In the first case, there might be points other than P_j that are of distance d_{ij} to P_i (i.e., these points are on the same circle centered on P_i of radius d_{ij}). The edge effect correction weights for these points are by definition equal to w_{ij}, the edge effect correction weight for P_j. In the second case, edge effect correction weights can be reused in subsequent bootstrapping simulations that are required for statistical significance testing. A bootstrap sample consists of n points that are randomly resampled, with replacement, from the n original points. In estimating

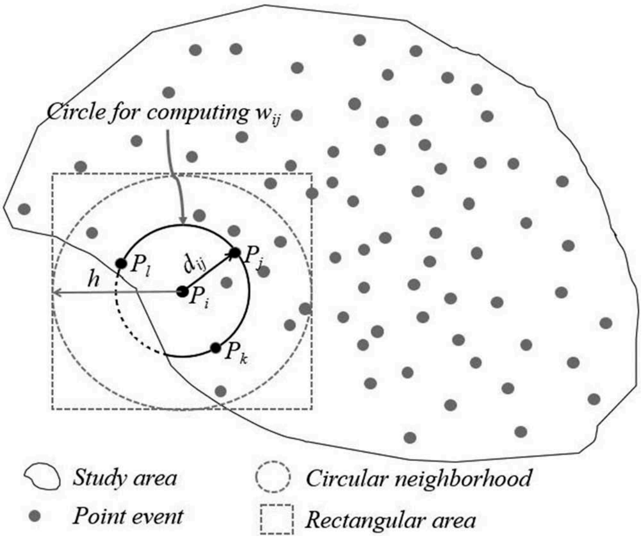

Figure 1. Optimizations to the algorithm for estimating *K* function. By sorting points using their *x, y* coordinates, inner traversal was confined to the rectangular area around P_i. Edge effect correction weights for P_j, P_k, and P_l should be identical by definition. Thus w_{ij} was reused when computing weights for P_k, and P_l.

$\hat{L}(h)$ on a bootstrap sample, the edge effect correction weights for points in circular neighborhood of P_i can be reused, instead of being computed again, as they have been computed in the initial run to estimate $\hat{L}(h)$ on the original points. By reusing edge effect correction weights, the complexity of bootstrapping simulations, $O(s{\cdot}v{\cdot}m{\cdot}n{\cdot}n')$ could be optimized to $O(s{\cdot}v{\cdot}n{\cdot}n')$. The factor *m* (i.e., the number of vertices on the boundary of the study area) disappeared. The complexity no longer depends on the shape of the study area in bootstrapping simulations. It should be noted that the second case of reusable weights only holds true for bootstrapping simulations (which was adopted in this study) but not for Monte Carlo simulations. Nevertheless, even if Monte Carlo simulations were adopted for significance testing, the first case of reusable weights still holds true both for the initial run to estimate $\hat{L}(h)$ on the original points and for Monte Carlo simulations. In this regard, reusing edge effect correction weights is applicable for optimizing point pattern analysis using *K* function in general. Yet, it is expected that reusing the weights would benefit bootstrapping simulations more than Monte Carlo simulations in terms of computation speedup.

There are two levels of parallelism in point pattern analysis using *K* function (including initial estimation and subsequent simulations) that can be exploited to accelerate the process. At the first level, in the initial run to estimate $\hat{L}(h)$ or in one simulation run, the outer traversal over the *n* points can be done in parallel because counting the number of points in circular neighborhood of any point, as well as computing edge effect correction weights for those points, is independent from that of another point in the outer traversal. At the second level, each simulation run is independent from each other and thus can be conducted in parallel.

3.2. *Optimizations*

3.2.1. *Sorting points*

The first optimization strategy was designed to confine the inner traversal to the rectangular neighborhood of any point *i*. To achieve this goal, points were sorted on their *x* and *y* coordinates before estimating *K* function on the points. The sorting was done in two steps. First, the *x* coordinate range of the *n* points was divided into equi-width bins, and the number of bins is specified by the user. Each point was put into the bin corresponding to its *x* coordinate. Second, points in each bin were sorted on their *y* coordinates. The *vector* container in C++ standard library was used as an in-memory data structure to hold the points and the *sort* function provided by the standard library was utilized to efficiently sort the points. Given the sorted points, when computing $\hat{L}(h_0)$ at a given distance h_0, each inner traversal for P_i was limited to a roughly rectangular area which is the minimum bounding box for the circle centered at P_i and with h_0 as the radius. The inner traversal started at P_i, and points in the same bin as P_i were then examined along both the increasing *y* direction and the decreasing *y* direction. Traversal in either direction stopped once the absolute *y*-coordinate difference between a point under examination and P_i exceeded h_0, or if the point with the minimum or the maximum *y*-coordinate in the bin was examined. The traversal then expanded to neighbor bins that could potentially contain points that were distance h_0 away from P_i on *x*-axis (i.e., the largest possible absolute *x*-coordinate difference between any point in a neighbor bin and P_i was no further than h_0). Similarly, traversal within each qualified neighbor bin was then conducted by selecting a random point as the starting point. If the starting point was further than distance h_0 on the *y*-axis from P_i, traversal proceeded along the *y* direction towards P_i, otherwise traversal proceeded along both the increasing *y* direction and the decreasing *y* direction. Criteria to stop traversal within each bin remained the same.

3.2.2. *Reusing edge effect correction weights*

The second optimization strategy was designed for reusing edge effect correction weights. Edge effect correction weights for points in the circular neighborhood with respect to P_i were stored in a hash table associated with P_i. The hash table was filled in with $<d_{i\cdot}, w_{i\cdot}>$ entries where $d_{i\cdot}$ was the distance from P_i to a point in its neighborhood; $w_{i\cdot}$ was the edge effect correction weight for that point. The key in the hash table for this entry was a hashed value of the distance $d_{i\cdot}$. Suppose it was needed to compute edge effect correction weight w_{ik} for P_k in the neighborhood of P_i. First, the distance d_{ik} was calculated and compared to existing $<d_{i\cdot}, w_{i\cdot}>$ entries in the hash table associated with P_i. The entry $<d_{ix}, w_{ix}>$ in which d_{ix} was the closest to d_{ik} was returned. Let Δd be the minimum distance difference at which difference in edge effect correction weights can be ignored. If $|d_{ik} - d_{ix}| \le \Delta d$, then $w_{ik} = w_{ix}$ (i.e., the weight was reused). Otherwise, w_{ik} was computed based on Ripley's isotropic correction and a new entry $<d_{ik}, w_{ik}>$ was inserted into the hash table associated with P_i. Here Δd was a distance tolerance that served to quantize the distance. This tolerance was of necessity because there is a low probability that the two points will have exactly the same distance to P_i. Using a larger Δd could reduce the number of edge effect weight calculations. Using a smaller Δd, however, will produce a more accurate *K* function. In determining the appropriate value for Δd, as demonstrated in Section 5.4.1, the dimension of the study area needs to be taken into account.

Upon completion of the initial run to estimate K function on the n points, every possibly needed <$d_{i\cdot}$, $w_{i\cdot}$> entries should have already been computed and stored in the hash table associated to P_i. In subsequent bootstrapping simulations, any weight in need was directly looked up from the hash table conveniently. The time complexity of either insertion or search operation in a hash table is $O(1)$ (Cormen 2009), meaning that inserting a <$d_{i\cdot}$, $w_{i\cdot}$> entry into the hash table or looking up a <$d_{i\cdot}$, $w_{i\cdot}$> entry from the hash table only took constant time that did not depend on the number of entries in the hash table.

3.3. *Parallelization*

The OpenMP programming model and the hybrid MPI/OpenMP programming model were adopted to exploit parallelism exposed in the optimized algorithm for point pattern analysis using K function.

3.3.1. *OpenMP-based parallelization*

The optimized algorithm was parallelized using OpenMP to exploit the first-level parallelism. The shared-memory programming model OpenMP was a logical choice for parallelizing K function because the massive point-wise comparisons involved in computing $\hat{L}(h)$ requires that the full point data set be accessible to the algorithm over the course of computation. Given the full point data set in a shared memory, computation on each of the points encountered in the outer traversal was independent from one another (i.e., the first-level parallelism) and thus was parallelized utilizing multicore CPUs in the OpenMP programming model. Specifically, the compiler directive '*#pragma omp parallel for*' was used to parallelize the outer level of the two nested for-loops iterating through the points (i.e., the outer traversal), dispatching computation on each point (i.e., partial weighted count) to threads running on multicore CPUs. The array data structure was used to record partial counts corresponding to distinct distances over which $\hat{L}(h)$ was estimated. Each thread had a local array to maintain the partial counts in order to avoid race conditions where two or more threads were attempting to update the same item in a globally shared array. Upon completion of the partial computation, partial counts saved in local arrays were aggregated to compute $\hat{L}(h)$. These steps were followed in both the initial estimation and simulations. In this OpenMP-based parallelization, only computing resource on a single computing node can be utilized.

3.3.2. *Hybrid MPI/OpenMP-based parallelization*

MPI can utilize computing resources distributed across different computing nodes. The optimized algorithm was parallelized using the hybrid MPI and OpenMP programming model to exploit both levels of parallelism. On top of the OpenMP-based parallelization on each node, computation on different nodes was coordinated using MPI by synchronizing executions and passing data between nodes whenever necessary. Specifically, in the initial run to estimate K function, computation on the n points (i.e., n outer traversal) was divided among computing nodes. Simulation runs were also divided among computing nodes. Scheduling information, indicating which points in the initial estimation and which simulation runs each node was responsible for, was consistently determined based on MPI process rank (i.e., node ID). Computation in the initial run on different nodes was synchronized using '*MPI_Barrier*'. Once all nodes had reached this synchronization point, partial results on different nodes were then aggregated using '*MPI_Reduce*' to compute $\hat{L}(h)$. Simulation runs on different nodes were conducted independently and no synchronization was needed. The amount of data transferred between computing nodes was significantly reduced in the above implementation. Every node kept a copy of the data files (i.e., the points and the study area) on its local hard drive. Upon invoked, each computing node directly read data from the files. Thus transferring a large amount of data (i.e., coordinates of the points and the study area) across nodes was avoided. At the simulation stage, each node performed simulation runs independently and saved results directly to files on its local hard drive.

4. Cloud computing platform

The proposed optimizations and parallelization were implemented and tested on a private cloud platform because of the various advantageous features provisioned by cloud computing (as discussed in Section 1). The cloud platform was built upon open-source cloud solution – Eucalyptus (Nurmi *et al*. 2009). Several characteristics of Eucalyptus make it a logical choice for this study. The design of Eucalyptus targets infrastructure commonly found within academic and laboratory settings. It is portable, modular, and easy to deploy atop existing computing resources (Nurmi *et al*. 2009). Eucalyptus also has good virtual machine (VM) isolation and security strategies, and its application programming interfaces are compatible with one of the popular public clouds – Amazon EC2 (Huang *et al*. 2013c). Correspondingly, third-party plug-ins, such as Hybridfox, which were originally developed to access EC2 cloud services, can be directly used to manipulate Eucalyptus resources. There were no significant performance differences in CPU, memory, and I/O of VMs created and managed by Eucalyptus and by other open-source cloud solutions, such as OpenNebula and CloudStack (Huang *et al*. 2013c).

The underlying computing infrastructure includes three computing nodes. Each node has 48 GB memory and dual eight-core CPU of 2.60 GHz. All nodes are connected through local area networks (LANs with 10 Gbps). A VM with 16 virtual CPU cores with the CPU speed of 2.6 GHz, 12 GB memory and 4 MB cache running Ubuntu 12.04 operating system was created. The GNU compiler collection and MPICH2 were installed as OpenMP runtime and MPI runtime, respectively. This VM was then imaged and another VM was created by launching a new instance from the image (i.e., the two VMs had the same configuration). Experiments for evaluating effectiveness of the optimizations and for testing performance of the OpenMP parallel algorithm were performed on only one VM. Experiments testing performance of the hybrid MPI/OpenMP parallel algorithm were performed on these two interconnected VMs.

5. Experiments

5.1. *Experiment design*

Experiments were designed to evaluate effectiveness of the optimizations and efficiency of the parallelization. The necessity of edge effect correction and the associated computational cost was assessed in Section 5.2. Effects of each of the two optimizations were evaluated separately in Section 5.3 and 5.4 respectively. Effectiveness of the two optimizations as a whole was evaluated in Section 5.5. The sequential *K* function algorithm was used for these evaluations. Efficiency of the OpenMP-based parallelization and the hybrid MPI/OpenMP-based parallelization was evaluated in Section 5.6. Point pattern analysis using *K* function was then performed on a real-world data set using the optimized and parallelized algorithm in Section 5.7.

Random points generated within an experimental study area were used in experiments evaluating effectiveness of the optimizations and efficiency of the parallelization. The expected *K* function on a random point pattern is known (see Section 2). Thus the estimated *K* function can be directly compared to the expected *K* function to assess the correctness of the optimizations. The experimental study area was the contiguous U.S. The boundary of the contiguous U.S. was extracted from the 2014 version of cartographic boundary shapefiles at a scale of 1: 2 million (U.S. Census Bureau 2015). Random point patterns of varying sizes over the study area were generated as experimental point data sets (Figure 2(a)). The real-word data set was the eBird checklist records reported by amateur birders (Wood *et al*. 2011, eBird 2015). Each record had an explicit geographic location expressed in longitude and latitude indicating where the birding session was conducted. The records within the contiguous U.S. in summer months of 2012 were extracted (228,858 points in total) (Figure 2(b)). In all experiments, unless otherwise stated, *K* function was estimated at 2000 distinct distances at an equal interval in the range of 0 to 700 km (i.e., the maximum distance at which *K* function was estimated).

In each experiment, execution time (i.e., elapsed time, or wall clock time) for estimating *K* function and the average execution time of one simulation were recorded. To evaluate the effectiveness of the optimizations, speedup factor *sf* was defined as an indicator of how much acceleration the optimizations achieved:

$$sf = T_{original}/T_{optimized} \tag{4}$$

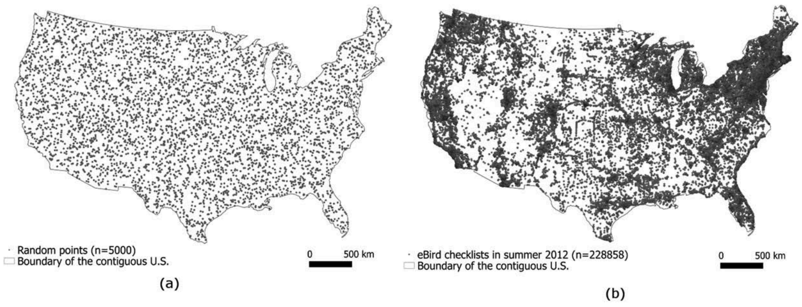
Random points (n=5000)
Boundary of the contiguous U.S.
0 500 km
(a)
eBird checklists in summer 2012 (n=228858)
Boundary of the contiguous U.S.
0 500 km
(b)

Figure 2. (a) Random points generated within the contiguous U.S. (b) The eBrid checklists within the contiguous U.S. in summer months of 2012.

where $T_{original}$ and $T_{optimized}$ are the execution time of the original algorithm and the optimized algorithm, respectively. To evaluate the efficiency of the parallelized algorithm, speedup ratio *sr* was used an indicator of how much acceleration the parallelization achieved (Wilkinson and Allen 2004):

$$sr = T_{sequential}/T_{parallel} \tag{5}$$

where $T_{sequential}$ and $T_{parallel}$ are the execution time of the sequential algorithm and the parallel algorithm. The execution times mentioned above did not include time spent on reading in data from files or writing outputs to files, and therefore allows for comparison of the execution time of the algorithms.

5.2. *Cost of edge effect correction*

The computational cost of edge effect correction was expected to be proportional to the geometric complexity of the study area (i.e., number of vertices on the study area boundary). Point pattern analysis was performed on random point data sets using study areas of different geometric complexity. Study areas of decreasing complexity were constructed by simplifying the original polygon with increasing simplifying tolerances. This resulted in simplified polygons with decreasing number of vertices on the boundary (Polygons 1 through 4 in Table 1). Random point data sets (n = 5000) were generated within the original polygon as well as within each of the four simplified polygons (i.e., five random point data sets in total). Point pattern analysis was performed on each random point data sets using the corresponding polygon for edge effect correction. In addition, Point pattern analysis was performed on the random point data set that were generated within Polygon 4 but using the bounding box of the points for edge effect correction.

The estimated $\hat{L}(h)$ was expected to be equal to h on random point data sets. Figure 3 showed that edge effect correction using the actual polygonal study area (i.e., Polygon 4) resulted in an estimation of $\hat{L}(h)$ that aligned to the expected line $\hat{L}(h) = h$, whilst edge effect correction using the bounding box resulted in an estimation of that was consistently above the expected line. Overestimation occurred if only the bounding box of the points was used for edge effect correction. It would certainly lead to a wrong conclusion that these points were clustered but in fact they occurred randomly. Thus, calculating edge effect correction weights using the bounding box was insufficient for edge effect correction. The actual polygonal study area was necessary for edge effect correction in order to estimate an unbiased K function.

Figure 4 showed the execution time for estimating K function over 5000 random points and the average execution time of one bootstrapping simulation using polygonal study areas of different complexity for edge effect correction. The execution time

Table 1. Study areas of different level of geometric complexity that were used in the experiments.

Study area	Simplifying distance tolerance (km)	Number of vertices on the boundary
Polygon 0	The original polygon	2995
Polygon 1	5	560
Polygon 2	10	302
Polygon 3	20	179
Polygon 4	50	86

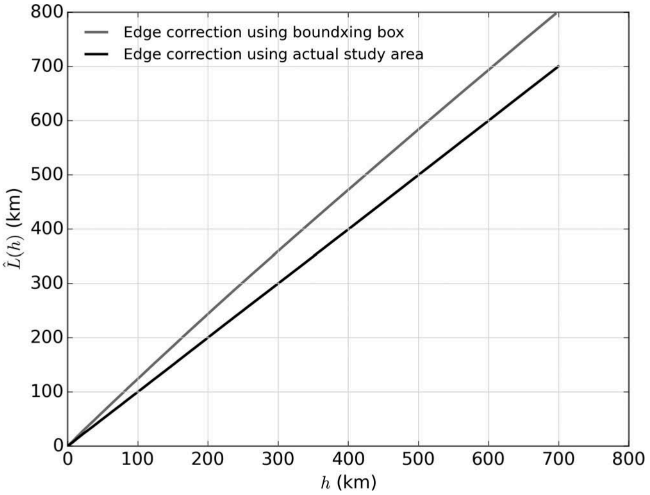

Figure 3. *K* function estimated on random points using either the bounding box or the actual polygonal study area for edge effect correction.

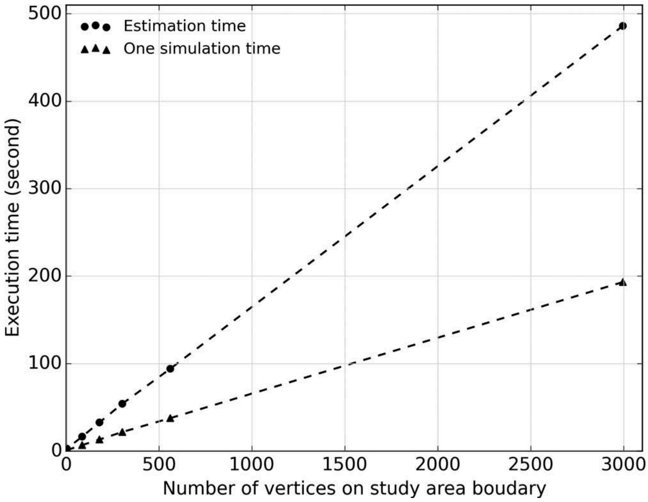

Figure 4. Execution time of performing point pattern analysis on random points using the study area of increasing geometric complexity for edge effect correction.

increased approximately linearly in the number of vertices on the study area boundary. In subsequent experiments, in order to complete the experiments in an acceptable amount of time, unless otherwise stated, random points were generated within Polygon 4 and it was used for edge effect correction in default.

5.3. *Effects of sorting points*

By sorting all points on their *x* and *y* coordinates, the search for points that were within distance *h* from a focal point can be confined to points within a rectangular neighborhood around the focal point (Sections 3.1 and 3.2). The degree to which the searching scope can be narrowed down by sorting the points depended on the maximum distance h_{max} at which $\hat{L}(h)$ was estimated. Point pattern analysis was performed on a random point data set (n = 50,000) but with varying h_{max}. All h_{max} investigated were no greater than one quarter of the dimension of the study area, as suggested by Ripley (1979). The execution time of estimating K function on the sorted points was compared to that on the unsorted points. In both cases, edge effect correction weights were not reused in order to investigate the effects brought purely by sorting points.

Results in Figure 5 showed that sorting points indeed accelerated the initial estimation and simulations. The speedup factor, defined as the ratio of execution time on unsorted points to that on sorted points, was consistently higher than 1. It was obvious that the speedup factor for either initial estimation or simulation increased decreasing h_{max}. For h_{max} greater than 500 km, the speedup factor for initial estimation and simulation were close to 1. But for smaller h_{max} such as 10 km, the estimation speedup factor was greater than 40 and the simulation speedup factor was over 70. The smaller h_{max} was, the larger degree to which sorting points helped confine the search scope around a focal point and, therefore, accelerate computation. Sorting points can effectively accelerate point pattern analysis using tasks performed at smaller spatial scales.

The time spent on sorting points was not included in the execution time. The *sort* function implemented in C++ standard library was very efficient that sorting 50,000 random points took nearly negligible time. Reading in the 50,000 random points and the study area from files, organizing them in proper data structures, and sorting the points took about 0.98 second in total. Sorting the points was done only once prior to the initial estimation.

5.4. *Effects of reusing edge effect correction weights*

Performance of the algorithm optimized by only reusing edge effect correction weights (i.e., no sorting points) was compared against performance of the original algorithm (i.e.,

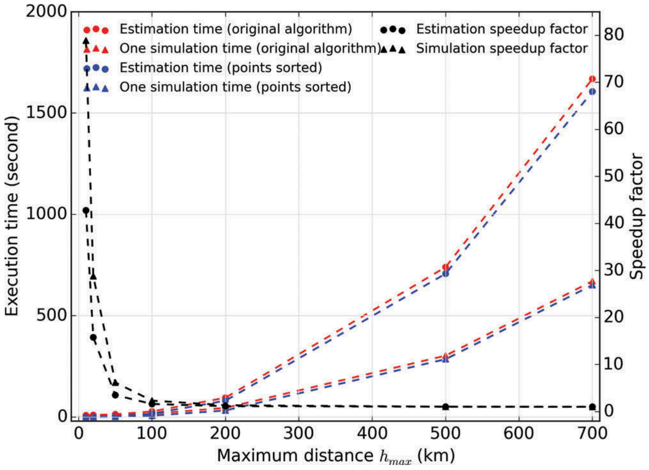

Figure 5. Execution time of performing point pattern analysis on unsorted points and on sorted points with varying h_{max}.

no optimization). In this way, the effects of reusing the weights alone can be investigated by examining the impact of various parameters in point pattern analysis using *K* function.

5.4.1. *Impact of the distance difference threshold Δd*

The distances in <$d_{i\cdot}$, $w_{i\cdot}$> entries in the hash table had to be quantized when reusing edge effect correction weights. The effect of quantization was that some weights looked up from hash tables were only approximations of exact weights (Section 3.2.2). The degree of approximation was determined by Δ*d*, the ignorable distance difference. *K* function estimated by reusing the weights could possibly differ from that estimated with exact weights calculated on the fly. Thus it is necessary to examine the accuracy of the *K* function estimated using algorithm with this optimization before embracing the computational improvement brought by this optimization. Point pattern analysis was performed on random points (n = 50,000). *K* function estimated by always computing the weights on the fly was compared to those estimated by reusing the weights with varying Δ*d*. To quantify the deviation of the *K* function estimated by reusing the weights (denoted as $\hat{L}_{\Delta d}(h)$) from that estimated without reusing the weights (denoted as $\hat{L}_0(h)$), the root mean-squared error (*RMSE*) between the two estimated *K* functions was calculated:

$$RMSE_{\Delta d} = \sqrt{\sum_{i=1}^{v} \left[\hat{L}_{\Delta d}(h_i) - \hat{L}_0(h_i)\right]^2 / v} \qquad (6)$$

where v is the number of distinct distances (e.g., 2000) over which *K* function was estimated. Results in Table 2 showed that the deviation increased with increasing Δ*d*. But as long as Δ*d* did not exceed 1000 m, the RMSE was within 1 m. Figure 6(a) showed that *K* function estimated by always computing the weights on the fly and that estimated by reusing the weights (Δ*d* = 500 m) were virtually identical.

The distance difference threshold Δ*d* was a trade-off between accuracy of the estimated *K* function and computational efficiency. As shown in Table 2, the execution time for initial estimation decreased with increasing Δ*d* because under a greater Δ*d*, more weights were reused and fewer weights were directly computed. The average execution time of one simulation remained nearly the same under different Δ*d*, the reason being that, in simulations, the weights could always be looked up from hash tables (Section 3.2.2). Although a smaller Δ*d* implied that a larger number of entries would be in the hash table associated with a point (thus required more memory space),

Table 2. Deviation of the *K* function estimated by reusing the weights from that estimated with directly computed weights.

		Execution time (second)	
Ignorable distance difference Δd (m)	$RMSE_{\Delta d}$ (m)	Estimation	Simulation*
500	0.49	34.70	7.73
1,000	0.87	29.43	7.52
3,000	4.76	19.64	6.74
5,000	11.31	18.55	6.44
10,000	32.54	15.78	6.06
20,000	91.53	14.91	5.75

* Average execution time of one bootstrapping simulation.

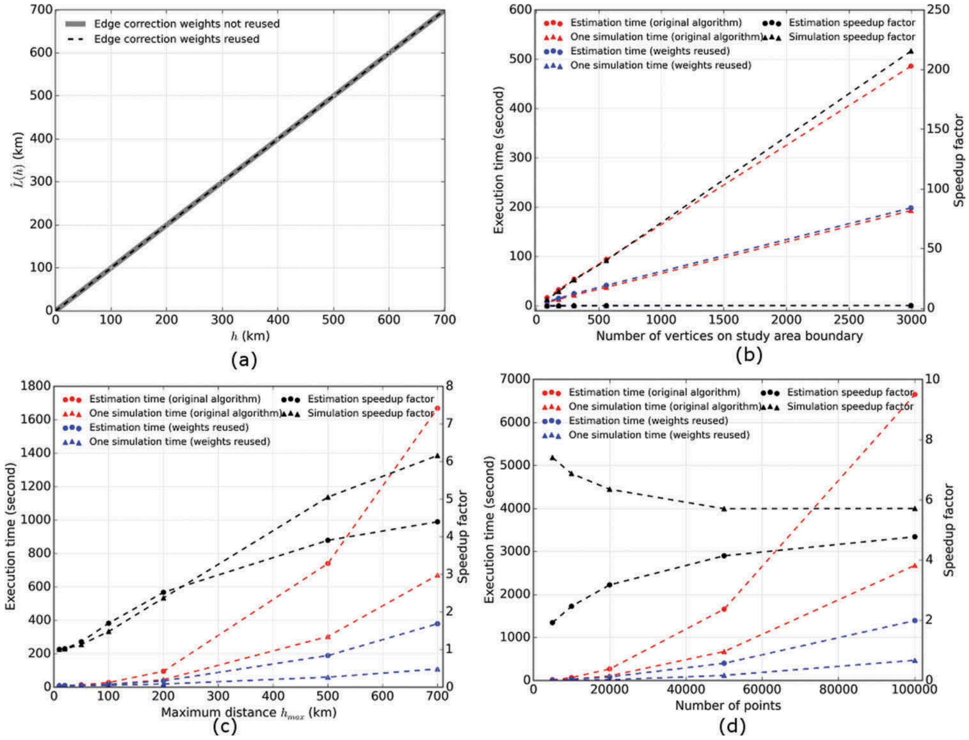

Figure 6. Effects of reusing edge correction weights. (a) *K* function estimated on random points with or without reusing edge effect correction weights. (b) Impact of the study area complexity. (c) Impact of the maximum distance h_{max}. (d) Impact of the problem size.

a hash table lookup only took constant time that was independent of the number of entries in the hash table. In determining a proper value for Δ*d*, one needs to take into account the dimension of the study area as well as the density of the points (i.e., smaller Δ*d* for denser points). For experiments in this study, Δ*d* was set to 500 m because it was small enough to ensure an accurate estimation of *K* function without losing much computational efficiency.

5.4.2. *Impact of the study area complexity*

The impact of geometric complexity of the study area on effectiveness of reusing edge effect correction weights was examined. Figure 6(b) showed the execution time for estimating *K* function on random point data sets (n = 5000) using study areas of varying geometric complexity (i.e., varying number of vertices on the boundary) for edge effect correction. Greater speedup factor was achieved with study areas of higher geometric complexity for both initial estimation and simulations. The weight calculations were more expensive with complex study areas. Thus reusing the weights achieved greater acceleration. The acceleration achieved in simulations was much greater than that in initial estimation. Although estimation speedup factors were relatively low, the absolute execution time saved by reusing the weights was considerably significant. Acceleration achieved in simulation increased much more rapidly than that in estimation with increasing study area complexity. In simulations the weights can always be looked up from hash tables because all the weights had been computed and stored in hash tables upon completion of the initial estimation. The study area complexity became irrelevant in simulations by reusing the weights. This was why the average time of one simulation remained roughly the same regardless of geometric complexity of the study areas used for edge effect correction. It was favorable to point pattern analysis tasks requiring a larger number of simulations for statistical significance testing.

5.4.3. *Impact of the maximum distance h_{max}*

The impact of the maximum distance h_{max} on effectiveness of reusing edge effect correction weights was assessed performing point pattern analysis on a random point data set (n = 50,000) with different h_{max}. Results in Figure 6(c) revealed that both the estimation speedup factor and the simulation speedup factor increased with increasing h_{max}, but the simulation speedup factor was generally higher than the estimation speedup factor.

5.4.4. *Impact of the problem size*

The impact of the problem size on effectiveness of reusing edge effect correction weights was assessed performing point pattern analysis on random point data sets of various sizes. Results in Figure 6(d) revealed that the simulation speedup factor was generally higher than the estimation speedup factor. The simulation speedup factor dropped only slightly as the problem size increased while the estimation speedup factor increased with increasing problem size.

It is noteworthy that acceleration achieved by reusing edge effect correction weights was indeed a space-time trade-off. Larger memory space was traded for shorter execution time. Besides memory required by data structures that organized the points, hash tables containing <$d_{i\cdot}$, $w_{i\cdot}$> entries definitely required additional memory. The benefit, however, was that both insertion and search operation on a hash table can be done in constant time regardless of the hash table length (Cormen 2009). Given enough memory, reusing the weights accelerated computation by replacing expensive weight calculations with cheap hash table lookups.

5.5. *Overall effectiveness of the optimizations*

The overall effectiveness was evaluated by comparing performance of the algorithm optimized by the two optimizations against performance of the original algorithm. Computational intensity of point pattern analysis using K function was largely determined by the maximum distance h_{max} and problem size.

5.5.1. *Impact of the maximum distance h_{max}*

Point pattern analysis was performed using both the original algorithm and the optimized algorithm on a random point data set (n = 50,000) with varying h_{max}. Results in Figure 7(a) showed that the speedup factors achieved by the two optimizations were slightly higher than those achieved by any one of them alone (see Figures 5 and 6(c) for comparisons). Both the estimation speedup factor and the simulation speedup factor increased with decreasing h_{max}. For smaller h_{max}, acceleration achieved by the optimizations was dictated by sorting points. For larger h_{max}, acceleration achieved was dictated by reusing the weights. Overall,

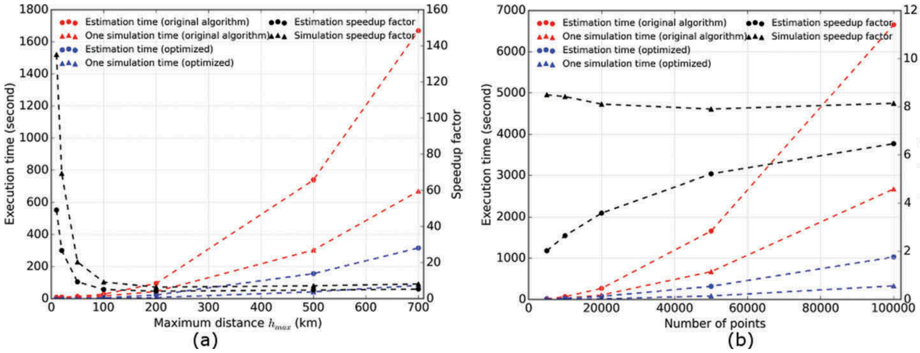

Figure 7. Overall effectiveness of the two optimizations. (a) Impact of the maximum distance h_{max}. (b) Impact of the problem size.

for large h_{max} such as 700 km, a baseline speedup factor of about 5 and 8 were achieved in estimation and simulation, respectively; For small h_{max} such as 10 km, a speedup factor of about 50 and 135 can be achieved in estimation and simulation, respectively.

5.5.2. *Impact of the problem size*

Point pattern analysis was performed using the original algorithm and using the optimized algorithm on random point data sets of various sizes. Results in Figure 7(b) showed that speedup factors achieved by the two optimizations were slightly higher than those achieved by solely reusing the weights (see Figure 6(d) for comparisons). The estimation speedup factor increased with increasing problem size. For small problem size such as 5000 points, an estimation speedup factor of about 2 was achieved; for problem size with points up to 100,000, an estimation speedup factor of about 6.5 was achieved. A constant simulation speedup factor of about 8 was achieved regardless of the problem size.

The optimization strategies effectively accelerated point pattern analysis using *K* function. Sorting points dictated the acceleration achieved in cases where the spatial scale examined was relatively small compared to the dimension of the study area. Reusing edge effect correction weights dictated the acceleration achieved in cases where the study area was geometrically complex, the spatial scale examined was large, or the size of the point events was large. A constant speedup factor of about 8 was achieved by the optimizations in simulations regardless of the size of point events. This was practically significant because a large number of simulations were required for statistical significance testing.

5.6. *Parallelization efficiency*

5.6.1. *OpenMP-based parallelization*

Performance of the optimized algorithm parallelized using OpenMP was evaluated by performing point pattern analysis on the two random point data sets (n = 50,000 and n = 100,000, respectively). Figure 8 showed the speedup ratio on varying number of CPU cores. Speedup ratio achieved in simulation was consistently higher than that achieved in the initial estimation. This was practically beneficial as a large number of simulations

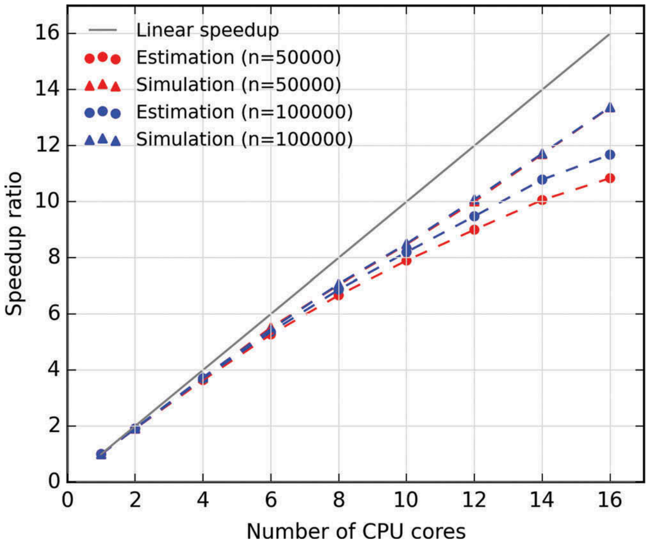

Figure 8. Speedup ratio of the OpenMP-parallel program for performing point pattern analysis on the two random point data sets.

are required for significance testing whilst the initial estimation only needs to be conducted once. The speedup ratios were only slightly lower than the ideal linear speedup. There was no significant difference in speedup ratios achieved in simulations between the two data sets. But the speedup ratios were higher for both simulation and initial estimation on the larger data set. This indicates that the OpenMP-based parallel algorithm achieved fairly good scalability with respect to both the number of CPU cores and problem size. Thus it is reasonable to expect that utilizing more CPU cores can match the computational intensity imposed by point pattern analysis tasks of large problem sizes.

The gap between the achieved speedup ratio and the ideal linear speedup was widening with the increasing number of CPU cores. This was most probably due to 'false sharing' in which multiple threads modify variables residing in the same cache line and thus seemingly independent computation on threads was slowed down because of the high cost of maintaining coherent copy of the cache line in thread-local cache (Torrellas *et al*. 1994). It also explained why the gap between the simulation speedup ratio and the ideal linear speedup was not as large as estimation speedup ratio. In the initial estimation, many updates (i.e., insertions) were performed on the hash tables. But in simulations read-only lookup was the dominant operation performed on the hash tables.

5.6.2. *Hybrid MPI/OpenMP-based parallelization*

Performance of the optimized algorithm parallelized using hybrid MPI/OpenMP was evaluated by performing point pattern analysis on the two random point data sets used in testing the OpenMP parallel algorithm. Results in Table 3 showed that the execution time of the hybrid MPI/OpenMP parallel algorithm running on one computing node was very close to that of the OpenMP parallel algorithm. There was only slight

Table 3. Execution time (in second) to perform point pattern analysis on the two random point data sets using the OpenMP-based and the hybrid MPI/OpenMP-based parallel algorithms. 16 CPU cores were utilized on each computing node.

	OpenMP		Hybrid MPI/OpenMP			
	1 computing node		1 computing node		2 computing nodes	
Number of points	Estimation	Simulation*	Estimation	Simulation*	Estimation	Simulation*
n = 50,000	29.21	6.17	32.67	6.21	18.07	6.22
n = 100,000	87.67	24.36	89.68	24.42	49.03	26.07

* Average execution time of one bootstrapping simulation.

overhead in the initial estimation and negligible overhead at the simulation stage by using MPI on top of OpenMP. Running on two nodes, the initial estimation time was much shorter than that running on one node. This was expected because the computation involved in the initial estimation was split up over more computing resources. However, the nonlinear relationship between the reduction of estimation and the number of nodes (i.e., estimation time using 2 nodes was more than half of that using 1 node) suggests additional overhead was introduced by communication and synchronization among MPI nodes. Moreover, simply dividing the points by equal number did not guarantee that the actual computation was equally dispatched among computing nodes because computation associated to each point may vary (e.g., points close to the study area boundary got involved more often in edge effect correction weights calculation).

The average time of one simulation using 2 nodes was very close but slightly higher than that of one node. This could be due to the extra communication overhead of VMs attributed by the virtualization technology (Huang *et al*. 2013c). At the same time, the time to complete a given number of simulations can be significantly reduced by utilizing multiple computing nodes. The computational intensity imposed by point pattern analysis tasks performed on large point data sets and involving a large number of simulation runs can be matched by utilizing computing resources on multiple computing nodes.

5.7 *Point pattern analysis on a real-world data set*

The optimized algorithm and its OpenMP and hybrid MPI/OpenMP parallelization were evaluated using the real-word data set – eBrid checklist data set (n = 228,858), and the results were shown in Table 4 (suppose 1000 bootstrapping simulations were required for significance testing). Compared to the original algorithm (i.e., no optimization) which

Table 4. Execution time of performing point pattern analysis on the eBrid checklist data set (228,858 points, 1000 simulations).

Algorithm	Estimation (minute)	Simulation* (minute)	Estimated total time (hour)	Speedup
Original (sequential)	973.69	393.03	6566.70	1.0
Optimized (sequential)	118.36	44.94	750.90	8.75
OpenMP (16 CPU cores)	10.53	3.48	58.22	112.79
Hybrid MPI/OpenMP (2 computing nodes, 16 CPU cores on each)	7.37	3.56	29.79	220.45

* Average execution time of one bootstrapping simulation.

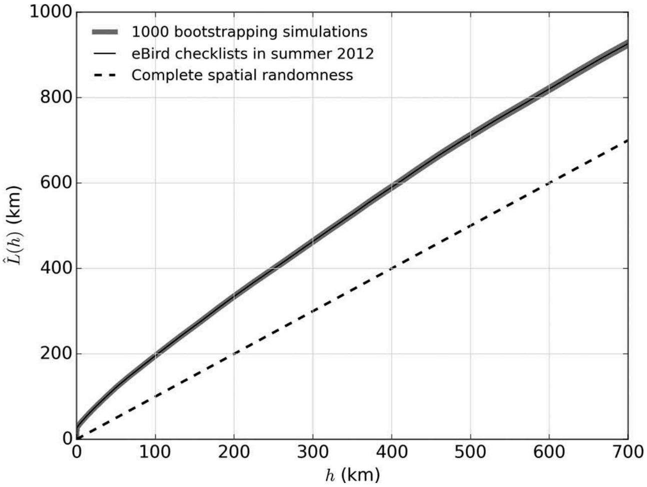

Figure 9. Results of comprehensive point pattern analysis on the eBird checklist data set. This figure shows that the point pattern underlying the eBird checklists is clustered at all spatial scales examined (i.e., 0–700 km).

could take 6566 hours to complete, the optimized algorithm could complete the task about 9 times faster. The optimized algorithm parallelized using OpenMP running on 16 CPU cores could complete the task 113 times faster. The optimized algorithm parallelized using hybrid MPI/OpenMP running on 2 computing nodes, each with 16 CPU cores, could complete the task 220 times faster.

The significant speedup in the point pattern analysis of the eBird checklist data set demonstrates the importance or the power of using parallel Ripley's *K* in spatial problem-solving. For example, to gain a more comprehensive assessment of the spatial nature of a large volume of point data (such as the eBird checklists presented above) one might need to examine the spatial pattern across different scales with smaller *h* values through larger *h* values. In the eBird checklist example, through this multiscale point pattern analysis we can compare the estimated $\hat{L}(h)$ for the real data set with the expected *K* function across many different *h* values and then draw conclusion about the spatial nature of the eBird checklist data set (Figure 9). Given what we have seen above, conducting this type of analysis using a sequential Ripley's *K* function is extremely time-consuming, even to the level which prohibits this comprehensive analysis. With the parallel implementation of Ripley's *K* function, tasks such as this type can be routinely conducted in spatial problem-solving.

6. Conclusions

This article presented two strategies (i.e., sorting points and reusing edge effect correction weights) to optimize the algorithm for performing point pattern analysis using *K* function. Theoretical analysis and empirical evaluations conducted on a cloud platform demonstrated that the optimization strategies effectively accelerated point pattern analysis using *K* function. On top of the acceleration achieved by the optimizations, parallelism exposed in the algorithm for performing point pattern analysis using *K* function was exploited using OpenMP and MPI on the cloud platform in order to enable point pattern analysis on point data sets of large size. Performance testing revealed that the OpenMP-based parallelization achieved fairly good scalability with respect to the number of CPU cores utilized. Only slight overhead introduced by using MPI on top of OpenMP. The hybrid MPI/OpenMP parallel algorithm running on multiple computing nodes much shortened the time to estimate *K* function and conduct simulations. The computational challenge imposed by point pattern analysis tasks on large point data sets involving many simulation runs can be addressed by utilizing distributed computing resources provisioned by cloud computing. Empowered by the optimized and parallelized algorithm, point pattern analysis using *K* function was accelerated utilizing cloud computing to enable point pattern analysis on spatial big data.

We expect the conclusions reached in this study remain valid in other computing environment (e.g., cloud computing provisioned by other cloud solutions, or traditional HPC clusters) for several reasons. First, the optimizations and parallelization were implemented in the cross-platform C++ programming language and thus can adapt to a different computing infrastructure very easily. The implementation only requires the OpenMP and MPI runtimes. The underlying computing infrastructure was transparent to the implementation. Second, Huang *et al*. (2013c) had shown that there were no significant performance differences in CPU, memory, and I/O of VMs created and managed by Eucalyptus and by other open-source cloud solutions. Third, Xu *et al*. (2009) had shown that the performance of parallel programs in VMs was close to that in native, nonvirtualized environment.

Using Ripley's *K* function as an example, this study demonstrated how cloud computing can be utilized to optimize and to accelerate geospatial analysis. The research results make contributions to the broader movement of advancing geospatial cyber infrastructure to geospatial cloud computing to implement the platforms required for addressing fundamental geoscience questions and application problems (Yang *et al*. 2010). By provisioning reliable and scalable geospatial service for massive users, geospatial cloud computing has the potential to engage nonexperts and citizens in scientific research (Zhu *et al*. 2015, Jiang *et al*. 2016) in order to advance public knowledge (Yang *et al*. 2011).

Acknowledgements

The work reported here was supported by grants from National Natural Science Foundation of China (Project No.: 41431177), National Basic Research Program of China (Project No.: 2015CB954102), Natural Science Research Program of Jiangsu (14KJA170001), PAPD, and National Key Technology Innovation Project for Water Pollution Control and Remediation (Project No.: 2013ZX07103006). Supports to A-Xing Zhu through the Vilas Associate Award, the Hammel Faculty Fellow Award, the Manasse Chair Professorship from the University of Wisconsin-Madison, and the 'One-Thousand Talents' Program of China are greatly appreciated. The comments from Professor James E. Burt and other members in the GIS group in the Department of Geography, University of Wisconsin-Madison are much appreciated.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

This work was supported by grants from National Natural Science Foundation of China (Project No.: 41431177), National Basic Research Program of China (Project No.: 2015CB954102), Natural Science Research Program of Jiangsu (14KJA170001), PAPD, and National Key Technology Innovation Project for Water Pollution Control and Remediation (Project No.: 2013ZX07103006). Supports to A-Xing Zhu through the Vilas Associate Award, the Hammel Faculty Fellow Award, the Manasse Chair Professorship from the University of Wisconsin-Madison, and the 'One-Thousand Talents' Program of China are greatly appreciated.

References

Baddeley, A.J. and Turner, R., 2005. Spatstat: an R package for analyzing spatial point patterns. *Journal of Statistical Software*, 12 (6), 1–42. doi:10.18637/jss.v012.i06

Baddeley, A.J., Turner, R., and Rubak, E., 2015. The SpatStat Package [online]. Available from: http://github.com/spatstat/spatstat/blob/master/src/corrections.c [Accessed 20 April 2015].

Burt, J.E., Barber, G.M., and Rigby, D.L., 2009. *Elementary statistics for geographers*. New York: Guilford Press.

Cormen, T.H., 2009. *Introduction to algorithms*. Cambridge, MA: MIT Press.

Diggle, P.J., 1983. *Statistical analysis of spatial point patterns*. Cambridge, MA: Academic Press.

eBird, 2015. [online]. Available from: http://ebird.org/content/ebird/ [Accessed 5 June 2014].

Evangelinos, C. and Hill, C., 2008. Cloud computing for parallel scientific HPC applications: feasibility of running coupled atmosphere-ocean climate models on Amazon's EC2. *Ratio*, 2 (2.40), 2–34.

Evans, M.R., *et al*., 2013. Enabling spatial big data via CyberGIS: challenges and opportunities. *In*: S. Wang and M. Goodchild, eds. *CyberGIS: fostering a new wave of geospatial innovation and discovery*. New York: Springer.

Fotheringham, A.S., Brunsdon, C., and Charlton, M., 2000. *Quantitative geography: perspectives on spatial data analysis*. Thousand Oaks, CA: Sage.

Gao, H., Barbier, G., and Goolsby, R., 2011. Harnessing the crowdsourcing power of social media for disaster relief. *IEEE Intelligent Systems*, 26 (3), 10–14.

Gao, S., *et al*., 2014. Constructing gazetteers from volunteered big geo-data based on Hadoop. *Computers, Environment and Urban Systems*. doi:10.1016/j.compenvurbsys.2014.02.004

Goodchild, M.F., 2007. Citizens as sensors: the world of volunteered geography. *Geojournal*, 69 (4), 211–221. doi:10.1007/s10708-007-9111-y

Haklay, M. and Weber, P., 2008. OpenStreetMap: user-generated street maps. *IEEE Pervasive Computing*, 7 (4), 12–18. doi:10.1109/MPRV.2008.80

Huang, Q., *et al*., 2013a. Utilize cloud computing to support dust storm forecasting. *International Journal of Digital Earth*, 6 (4), 338–355. doi:10.1080/17538947.2012.749949

Huang, Q., *et al*., 2013b. Accelerating geocomputation with cloud computing. *In*: X. Shi, V. Kindratenko, and C. Yang, eds. *Modern accelerator technologies for geographic information science*. New York: Springer, 41–51.

Huang, Q., *et al*., 2013c. Evaluating open-source cloud computing solutions for geosciences. *Computers & Geosciences*, 59 (September), 41–52. doi:10.1016/j.cageo.2013.05.001

Illian, J., *et al*., 2008. *Statistical analysis and modelling of spatial point patterns*. Hoboken, NJ: John Wiley & Sons.

Jiang, J., *et al*., 2016. CyberSoLIM: a cyber platform for digital soil mapping. *Geoderma*, 263 (1), 234–243. doi:10.1016/j.geoderma.2015.04.018

Law, R., *et al*., 2009. Ecological information from spatial patterns of plants: insights from point process theory. *Journal of Ecology*, 97 (4), 616–628. doi:10.1111/jec.2009.97.issue-4

Li, Q., Zhang, T., and Yu, Y., 2011. Using cloud computing to process intensive floating car data for urban traffic surveillance. *International Journal of Geographical Information Science*, 25 (8), 1303–1322. doi:10.1080/13658816.2011.577746

Mineter, M.J., Dowers, S., and Gittings, B.M., 2000. Towards a HPC framework for integrated processing of geographical data: encapsulating the complexity of parallel algorithms. *Transactions in GIS*, 4 (3), 245–261. doi:10.1111/1467-9671.00052

Nurmi, D., *et al*., 2009. The eucalyptus open-source cloud-computing system. *In*: F. Cappello, C. Wang, and R. Buyya, eds. *Proceedings of the 9th IEEE/ACM international symposium on cluster computing and the grid*, 18–21 May Shanghai. Washington, DC: IEEE Computer Society, 124–131.

Pijanowski, B.C., *et al*., 2014. A big data urban growth simulation at a national scale: configuring the GIS and neural network based land transformation model to run in a high performance computing (HPC) environment. *Environmental Modelling & Software*, 51, 250–268. doi:10.1016/j.envsoft.2013.09.015

Ripley, B.D., 1979. Tests of "randomness" for spatial point patterns. *Journal of the Royal Statistical Society. Series B (Methodological)*, 41 (3), 368–374.

Ripley, B.D., 1988. *Statistical inference for spatial processes*. Cambridge: Cambridge University Press.

Rowlingson, B.S. and Diggle, P.J., 1993. Splancs: spatial point pattern analysis code in S-Plus. *Computers & Geosciences*, 19 (5), 627–655. doi:10.1016/0098-3004(93)90099-Q

Shekhar, S. *et al*., 2012. Spatial big-data challenges intersecting mobility and cloud computing. *In*: *Proceedings of the eleventh ACM international workshop on data engineering for wireless and mobile access, SIGMOD/PODS '12 international conference on management of data*, 20–24 May Scottsdale, AZ. New York: ACM, 1–6.

Stojanovic, N. and Stojanovic, D., 2013. High-performance computing in GIS: techniques and applications. *International Journal of Reasoning-based Intelligent Systems*, 5 (1), 42–49. doi:10.1504/IJRIS.2013.055126

Tang, W., Feng, W., and Jia, M., 2015. Massively parallel spatial point pattern analysis: Ripley's K function accelerated using graphics processing units. *International Journal of Geographical Information Science*, 29 (3), 412–439. doi:10.1080/13658816.2014.976569

Torrellas, J., Lam, M.S., and Hennessy, J.L., 1994. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43 (6), 651–663. doi:10.1109/12.286299

U.S Census Bureau, 2015. Cartographic boundary shapefiles – Nation [online]. http://www2.census.gov/geo/tiger/GENZ2014/shp/cb_2014_us_nation_20m.zip [Accessed 3 July 2015].

Wang, S., 2013. CyberGIS: blueprint for integrated and scalable geospatial software ecosystems. *International Journal of Geographical Information Science*, 27 (11), 2119–2121. doi:10.1080/13658816.2013.841318

Wang, Y., Wang, S., and Zhou, D., 2009. Retrieving and indexing spatial data in the cloud computing environment. *In*: M.G. Jaatun, G. Zhao, and C. Rong, eds. *Proceedings of the 1st international conference on cloud computing*, 1–4 December Beijing. Heidelberg: Springer, 322–331.

Wilkinson, B. and Allen, M., 2004. *Parallel programming: techniques and applications using networked workstations and parallel computers*. 2nd ed. Upper Saddle River, NJ: Pearson Prentice Hall.

Wood, C., *et al*., 2011. eBird: engaging birders in science and conservation. *PLoS Biology*, 9 (12), e1001220. doi:10.1371/journal.pbio.1001220

Wright, D.J. and Wang, S., 2011. The emergence of spatial cyberinfrastructure. *Proceedings of the National Academy of Sciences of the United States of America*, 108 (14), 5488–5491. doi:10.1073/pnas.1103051108

Xu, C., Bai, Y., and Luo, C., 2009. Performance evaluation of parallel programming in virtual machine environment. *In: Sixth IFIP international conference on network and parallel computing*, 19–21 October Gold Coast. Washington, DC: IEEE Computer Society, 140–147.

Yang, C., *et al*., 2011. Spatial cloud computing: how can the geospatial sciences use and help shape cloud computing? *International Journal of Digital Earth*, 4, 305–329. doi:10.1080/17538947.2011.587547

Yang, C., *et al*., 2010. Geospatial Cyberinfrastructure: past, present and future. *Computers, Environment and Urban Systems*, 34 (4), 264–277. doi:10.1016/j.compenvurbsys.2010.04.001

Zhu, A.-X., *et al*., 2015. A citizen data-based approach to predictive mapping of spatial variation of natural phenomena. *International Journal of Geographical Information Science*, 29 (10), 1864–1886. doi:10.1080/13658816.2015.1058387