A MapReduce Approach to $G_i^*(d)$ Spatial Statistic

Yan Liu(1,2) Kaichao Wu(3) Shaowen Wang(1,2) Yanli Zhao(1) Qian Huang(4)
yanliu@uiuc.edu kaichao@cnic.cn shaowen@uiuc.edu zhao56@uiuc.edu skyswind@pku.edu.cn

(1)CyberInfrastructure &Geospatial Information Laboratory (CIGI)
University of Illinois at Urbana-Champaign, Urbana, Illinois, USA
(2)National Center for Supercomputing Applications (NCSA)
University of Illinois at Urbana-Champaign, Urbana, Illinois, USA
(3)Computer Network Information Center (CNIC)
Chinese Academy of Science (CAS), Beijing, China
(4)Institute of Remote Sensing and GIS
Peking University, Beijing, China

ABSTRACT

Managing and analyzing massive spatial datasets as supported by GIS and spatial analysis is becoming crucial to geospatial problem-solving and decision-making. MapReduce provides a data-centric computational model through which highly scalable spatial analysis computation can be achieved. However, it is challenging to leverage multi-dimensional spatial characteristics on the horizontally-partitioned and transparently managed MapReduce data system for improving the computational performance of spatial analysis. This paper tackles this challenge through the development of MapReduce-based computation of $G_i^*(d)$ – a spatial statistic for detecting local clustering. Without exploiting spatial characteristics, $G_i^*(d)$ computation for a particular location requires pair-wise distance calculation for all points of a given dataset. A spatial locality-based storage and indexing strategy is developed to associate spatial locality with storage locality on MapReduce platform. Based on a spatial indexing method, unnecessary map tasks can be eliminated for a MapReduce job, thus significantly improving the overall computation performance. To leverage underlying parallelism on storage nodes, an application-level load balancing mechanism is developed to produce even loads among map tasks based on adaptive spatial domain decomposition. Experiments show the effectiveness of the developed storage and indexing strategy with different distance parameter settings. Significant reduction on execution time for all-point computation is observed through the use of the application-level load balancing mechanism.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications

General Terms

Algorithms, Measurement, Performance, Design, Experimentation

Keywords

Cloud computing, data-centric computing, spatial statistics

1. INTRODUCTION

GIS and spatial analysis are ubiquitously applied in numerous fields for solving scientific problems and improving decision-making practices with significant societal impacts [1-3]. However, the deluge of spatial data poses significant computational challenges to GIS and spatial analysis methods and software that are often not designed to scale to massive spatial datasets.

Cloud computing technologies such as Google MapReduce/BigTable [4, 5] and the related open source project Hadoop [6] (referred to as MapReduce hereafter) provide a scalable parallel and distributed platform for data-driven computing. Built on a horizontally scalable (or "scale-out") [7] and distributed file system, MapReduce splits large data into evenly sized data blocks and stores them on "shared-nothing" storage nodes [4, 8]. Data parallelism is exploited by transparently moving computational tasks to storage nodes where data reside through a generic master-slave programming and execution model that consists of two elements: *map* and *reduce*.

The scalable and reliable data storage, retrieval, and processing capabilities provided by MapReduce represent both opportunities and challenges for developing scalable and efficient data-intensive spatial analysis solutions [9]. While scalable computation on massive data can be achieved by simply adding storage nodes to a MapReduce system, data partitioning in MapReduce is horizontal which means that a data block is split by default when its size exceeds a threshold. Therefore, the storage and indexing of multi-dimensional spatial data on MapReduce is a crucial issue for scalable spatial analysis because spatial characteristics of distributed data blocks need to be taken into account for achieving optimal computational performance. Furthermore, MapReduce system configurations such as for data distribution and job execution are often tuned from system perspective and not recommended to be changed by applications. Efficient domain decomposition and task scheduling techniques for spatial data analysis [10] at the level of applications, therefore, cannot be applied directly without changing MapReduce internal configurations.

This paper addresses the aforementioned challenges through the development of a MapReduce-based parallel computation method for a local spatial statistic: $G_i^*(d)$ [11]. $G_i^*(d)$ is widely used for local clustering detection. The computation of $G_i^*(d)$ is both data- and compute-intensive [12]. To compute $G_i^*(d)$ values using MapReduce, map and reduce algorithms are developed to adapt the computation onto computing partial results through map tasks based on locally available data and aggregating map output into global $G_i^*(d)$ results in reduce tasks.

Specifically, a spatial locality-based storage and indexing strategy (SLSI) is developed to exploit spatial characteristics for scalable computation by: 1) transforming spatial locality characteristics into storage locality information properly; and 2) building spatial indexes to eliminate unnecessary map function calls and corresponding data loading overhead. Application-level parallelism is further exploited to improve computation performance through a suite of adaptive spatial domain decomposition methods without changing underlying MapReduce data distribution and job scheduling policies. These methods achieve application-level load-balancing by creating evenly sized query data for every map function call.

In the remainder of this paper, section 2 describes related work on parallel and distributed $G_i^*(d)$ computation and MapReduce-based geospatial computation. Section 3 describes the details of SLSI . Section 4 evaluates the effectiveness of exploiting spatial characteristics for the improvement of computation performance. Conclusions and future work are discussed in section 5.

2. BACKGROUND AND RELATED WORK

$G_i^*(d)$ is a spatial statistic used for detect local clustering [11]. It is particularly helpful when applied to datasets for which global measures of spatial dependence, such as Geary's *c* and Moran's *I* or the *G* statistic [13], may fail to reveal the existence of important pockets of clustering. To demonstrate the generality of our computation approach, we use a standardized form of this statistic [14]. Let z_i, $i = 1, ..., n$, denote measurements of the variable of interest taken at *n* distinct locations with known spatial coordinates. A Euclidean distance *d* of interest is chosen such that most points in the dataset have at least one neighbor point within distance *d*. Then a spatial weight matrix $\{ w_{ij}(d) \}$ is defined such that $w_{ij}(d) = 1$ if location *i* is within distance *d* of location *j* (note that $w_{ii}(d) = 1$), and $w_{ij}(d) = 0$ otherwise. If we let $W_i^* = \sum_j w_{ij}(d)$, $S_{1i}^* = \sum_j w_{ij}^2(d)$, and μ and σ^2 denote the standard sample mean and sample variance, then the value of $G_i^*(d)$ is calculated as:

$$G_i^*(d) = \frac{\sum_j w_{ij}(d) z_j - W_i^* \mu}{\sigma\{[(nS_{1i}^*) - W_i^{*2}]/(n-1)\}^{1/2}} \qquad <1>$$

where μ and σ are usually known or pre-computed. Therefore, the computation of $G_i^*(d)$ depends on efficient determination of $\{ w_{ij}(d) \}$, which involves pair-wise distance calculation.

Parallel computing of $G_i^*(d)$ can exploit two types of application-level parallelism: *d*-parallelism and spatial parallelism [12]. This paper focuses on the spatial parallelism embedded within spatial data. A parallel and distributed $G_i^*(d)$ method is developed in [12] to use Grid computing resources for efficient computation of $G_i^*(d)$. The method is based on a spatial computational domain theory [10, 15] which is developed to provide a general guidance for achieving high-performance and scalable spatial analysis computation [16]. Although the data-centric MapReduce programming model is different from conventional Grid and cluster computing models, the generality of the spatial computational domain theory and associated computational intensity analysis make it possible to analyze the computational intensity of our MapReduce solution and develop spatially explicit strategies to improve the computational performance on MapReduce.

MapReduce has been used in GIS and spatial analysis for spatial data repository construction, spatial query processing, and GIS workflow support. Zhang *et.al* [17] leverages MapReduce in compute-intensive spatial query processing. HDFS [18] is used to store spatial objects. A spatial query is served by using map tasks to search for matched objects on storage nodes and using reduce tasks to intersect MBRs (minimum bounding rectangle) for final aggregation of query results. Cary *et.al* [9] applied MapReduce to construct R-Tree on massive spatial vector data by creating a set of small R-trees in parallel and using space-filling curves to group spatially close objects into the same partition. MapReduce is used to efficiently build R-tree index. Chen *et.al* [19] used MapReduce as a parallel GIS workflow execution environment which is able to identify and simultaneously execute multiple jobs without dependencies in GIS workflows.

Closely related to our work, Stupar *et.al* [20] uses the Hadoop distributed platform for large-scale *k*-nearest neighbor (KNN) query processing of high-dimensional datasets. A set of hash values are calculated by using locality sensitive hashing (LSH)-based sampling to index data points using dimensional characteristics. The values are stored in a spatial indexing table to approximate the distance between query and stored datasets. Only matched subsets of data are involved in the map function. In our work, a spatial indexing strategy is also developed for computational performance improvement by building nearest cell distance matrix. While the hash table in Stupar *et.al* 's work is based on an approximation mechanism that affects matching accuracy, our spatial indexing strategy is an exact solution to guarantee the correctness in selecting neighboring cells within distance *d* in a given spatial domain.

3. MAPREDUCE-BASED COMPUTATION

Without losing generality, our MapReduce-based computation of $G_i^*(d)$ exploits data parallelism within a given point dataset *D*, and is conceptualized as follows:

- The spatial domain representing *D* is decomposed into *m* cells c_k, $k = 1, \ldots, m$, where $D = \bigcup c_k$. Each cell contains a set of points. The decomposition is denoted as $C = \{c_k\}$.
- $\{c_k\}$ is stored as *l* data blocks on MapReduce by a partitioning, denoted as $P = \{p_i\}$, where $i = 1, \ldots, l$; p_i is a data block and $p_i \subset \{c_k\}$, $k=1, \ldots, m$. Here we assume for any cell c_k, its size $|c_k|$ is less than the maximum size of a data block (64MB for a Hadoop HDFS block and 256MB for a Hahoop HBase table region).
- Each point in dataset *D* is represented as a tuple <*x*, *y*, *a*>, where *x* and *y* are the coordinates and *a* is a vector of point attributes. The *z* value in <1> is a function of *a*: $z = f(a)$.

This conceptual formalization is necessary because partitioning in MapReduce is not directly controllable by application developers. Given a set of points of interest $Q \subseteq D$ (referred to as a *query* *cell* hereafter), { $w_{ij}(d)$ } is derived for each point $i \in Q$ and then each $G_i^*(d)$ value is calculated using <1>. The calculation of mean and variance of *z* values also occurs at the decomposition step. A straightforward computation of single-point $G_i^*(d)$ value would involve all the cells to derive { $w_{ij}(d)$ }, if no prior knowledge on the distance between point *i* and other points in other cells is available. To compute all-point $G_i^*(d)$, let $Q = D$. Figure 1 illustrates the steps needed to compute all-point $G_i^*(d)$.

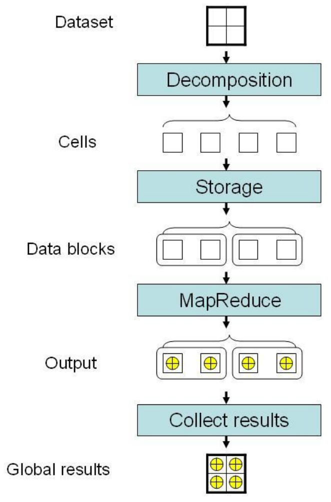

Figure 1. Workflow for all-point $G_i^*(d)$ computation

3.1 MapReduce-based $G_i^*(d)$ computation

In compute-centric parallel models [21], when there is a need to derive $w_{ij}(d)$ for cell c_k, either communication or replication needs to be engaged to make points in c_k accessible by a processor if c_k is not available locally to the processor. In contrast, MapReduce is a data-centric computing model that moves computing to where data are hosted. As a result of data partitioning, computation is also parallelized into map tasks, each of which is responsible for processing a subset of data blocks located on a storage node. In the case of $G_i^*(d)$, dataset *D* is partitioned by MapReduce into evenly sized data blocks and stored on its distributed file system. For a decomposition *C* consisting of a set of data cells (controlled at the application level), a partitioning *P* (controlled by MapReduce) creates a set of evenly sized data blocks, each consisting of a subset of *C*. The assignment and replication of *P* on specific storage nodes are managed inside of MapReduce without any interface to applications. Application developers are provided with two types of programming interface functions known as map and reduce. The map function is given an input as *InputSplit* [6] and computes partial results based on local data specified in the input. The output of a map function is a set of <key, value> pairs that are processed by a reduce function. The reduce function aggregates partial results into global results. Once map and reduce functions are defined, the computation is executed as a job that dispatches map and reduce tasks to storage nodes. One map task often works on a data block located at an individual storage node and processes data with multiple map function calls.

Our MapReduce computation includes two steps: calculation of Z_{ij}, W_{ij}, and S^2_{ij} at the level of cells as partial results, and calculation of aggregated global results. Table 1 lists the pseudo code of our MapReduce algorithm for all-point $G_i^*(d)$ computation. The *job* algorithm reads all |*C*| cells of a spatial domain as query cells and composes one MapReduce job for each query cell. A MapReduce job is executed by MapReduce's task scheduling system by sending map tasks to storage nodes and organizing map task output into one or more reduce tasks. The map function takes a query cell and a local data cell as input parameters, and computes partial results of Z_{ij}, W_{ij}, and S^2_{ij} for all pairs of <*i*, *j*> in the two cells, if the distance between *i* and *j* is within *d*. Output is indexed by coordinates in each query cell. The reduce function then aggregates partial results from all storage nodes and computes global $G_i^*(d)$ value for every point within each query cell.

Table 1. $G_i^*(d)$ MapReduce algorithms

```
Algorithm: Job
Input: $Q = R$; $d$: distance parameter
Output: G: $G_i^*(d)$ value set on file system
1.  $J = Ø$;
2.  foreach $q$ in $Q$
3.      $J = J ∪ \{ job(q, d) \}$;
4.  Submit $J$ to MapReduce;
5.  foreach $j$ in $J$
6.      wait($j$);

Algorithm: Map
Input: $q$: query cell; $c$: local cell; $d$: distance parameter
Output: $M$: { <key, value> }
1.  foreach $i$ in $q$
2.      key = <$ID_q$, $x_i$, $y_i$, $d$>;
3.      $Z_i = W_i = S^2_i = 0$;
4.      foreach $j$ in $c$
5.          if (distance($i$, $j$) $\le d$)
6.              $Z_i$ += $Z_j$; $W_i$ += 1; $S^2_i$ += 1;
7.      $M = M ∪$ { <key, < $Z_i$, $W_i$, $S^2_i$> > };

Algorithm: Reduce
Input: $M$: {<key, < $Z_i$, $W_i$, $S^2_i$> >}, intermediate result set
Output: G: $G_i^*(d)$ value set for $Q$
1.  $tZ_i = tW_i = tS^2_i = 0$;
2.  foreach <key, < $Z_i$, $W_i$, $S^2_i$> > in $M$
3.      $tZ_i$ += $Z_j$; $tW_i$ += $W_i$; $tS^2_i$ += $S^2_i$;
4.  $g = (tZ_i - tW_i * \mu) / \{\sigma * sqrt[n * tS^2_i - (tW_i)^2] /(n-1)]\}$;
5.  $G = G ∪$ { <key.$x$, key.$y$, key.$d$, $g$> };
```

Based on Hadoop, our implementation uses HBase to store decomposed data cells for data query and efficient storage (based on empty cell compression and column-wise storage). Each dataset is stored in HBase as a table that is physically stored as a set of regions (data blocks) distributed on region servers (storage nodes). A decomposed cell becomes one row in a HBase table. Each cell is processed by a map function call using the default HBase table row InputSplit. The organization of a dataset as a set of cells enables application-level control of data storage access, which is important to exploit spatial parallelism for achieving desirable computational performance.

3.2 Computational intensity analysis

A primary advantage of using MapReduce is to exploit its capability for scaling data processing by simply increasing the number of storage nodes. In $G_i^*(d)$ computation, the computing complexity of map function is $O(|q| * |c|)$, where q and c stand for query and local data cells, respectively. This is because computation cost of each partial result for *Zi, Wi, and* S^2i is a constant, making distance calculation the only varying factor. The size of q and c, in worst case, is bounded by data block size limit on MapReduce regardless the size of a particular dataset. Since distance matrix between points in query cell and local cell is not needed for partial result computation in map function, the memory and I/O requirement for a map function is thus $O(|q|+|c|)$, instead of $O(|q| * |c|)$.

The execution time of the job algorithm (Table 1) is dependent on the size of datasets because the number of jobs created increases as the overall number of query cells increases. Note that in compute-centric models, distance calculation between query cell i and data cell j can be reused when they switch roles: j as query cell and i as data cell. On MapReduce, such reuse is difficult because cell i and j may not be on the same data block.

Our implementation as specified in Table 1 assumes that, for each query point i, all of the points within a dataset need to be considered. Within the map function that processes a local data cell, it is evident that all points in the local data cell must be computed for pair-wise distance measurement to detect if a point is within d-distance of point i in a query cell. But, as illustrated in figure 2, data cells outside of distance d do not contribute to the computation of $G_i^*(d)$. Figure 2 shows our strategy of exploiting spatial locality to group spatially close points together into one data cell. This way, spatial locality is preserved into storage locality and provides potential to eliminate unnecessary distance calculations for improving computational performance.

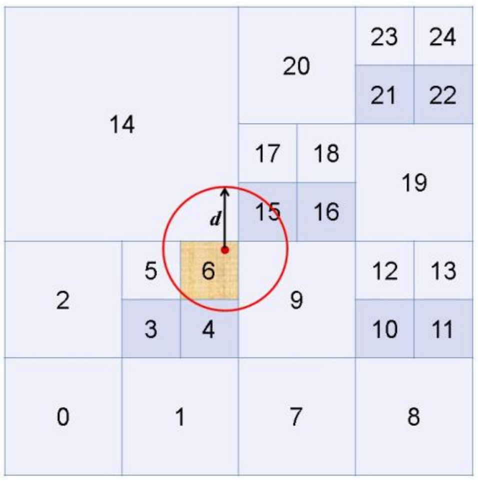

Figure 2. $G_i^*(d)$ calculation for a point. The circle designates the search range for all points within d. The spatial domain is decomposed into 25 cells using a region quadtree and indexed using Z-curve [22]. Cells 4,5,6,9,14,15 should be considered in computation, and cell 6 is the query cell.

In principle, spatial locality can be exploited on MapReduce at three levels progressively: intra-cell, inter-cell, and inter-block. At intra-cell level, spatially close points are grouped within a data cell. The computational cost can be reduced if it is possible to decide whether this data cell contributes to the $G_i^*(d)$ computation of a given query cell in map function. At inter-cell level, keeping spatially close cells together in a data block does not reduce the total computation to be conducted by map functions because the number of data cells involved in the computation of a query cell does not change upon the ordering of cells on storage. Total computation time can be reduced if spatial locality information among cells can be leveraged by MapReduce job scheduling to dispatch jobs with prior knowledge on how much time each map task takes (note that each map task processes all data cells on a data block). However, change of MapReduce system configurations is needed to exploit this inter-cell level of spatial locality. It is not recommended because such a change may affect system scalability and availability. The inter-block level is not applicable because the distribution and replication of data blocks to physical locations are managed internally by MapReduce system and cannot be controlled by applications. Consequently, we focus on exploiting intra-cell level spatial locality by addressing which data cells do not need to be considered in computation before MapReduce jobs start through the development of a spatial locality-based storage and indexing mechanism (SLSI).

3.3 Spatial locality-based storage and indexing

SLSI is designed to achieve two objectives: 1) keep points in a data cell spatially close; and 2) detect whether a data cell should be included for MapReduce computation in advance. The first objective is achieved by using spatial decomposition methods (specifically, quadtree-based decomposition) [23], also illustrated in figure 2.

Our solution to achieve the second objective is based on the following observation: for any point i in query cell q and point j in local data cell c, the distance between i and j is larger than d if the nearest distance between q and c is larger than d. The nearest distance between q and c, referred to as *nearest cell distance*, is defined as the shortest distance between any two points on query and local cell boundaries, respectively. For example, in figure 2, the nearest distance between cell 1 and 17 is the distance between the upper-right corner of cell 1 and lower-left corner of cell 17. Based on this observation, a nearest cell distance matrix can be built at the time a dataset is decomposed to calculate and store the nearest cell distances as a spatial index.

SLSI stores the spatial index on HDFS. Before a MapReduce job starts, the main program of job algorithm (Table 1) looks up the nearest cell distance matrix to retrieve those cells that are within distance d to the query cell q. The IDs of these cells are passed to a MapReduce job as an input filter that works at the time of input data loading in map tasks by generating the input data to map function as the intersection of cells locally available on data block and cells whose IDs are specified the input filter .

It is worth noting that using nearest cell distance is a conservative measure to detect whether a data cell is needed for the computation of a query cell q and a specific d. Some points in a selected cell that is within d-distance of q may still fall out of d-distance from q. But no points within d are ignored. Moreover, decomposition parameters such as point density threshold in quadtree-based decomposition affect the effectiveness of SLSI because different parameter values change cell boundaries. In principle, finer decomposition will improve the accuracy of using the nearest cell distance matrix, but requires more preprocessing computation.

3.4 Load balancing through adaptive spatial domain decomposition

Without changing underlying scheduling policies on Hadoop, spatial decomposition methods are developed to exploit the parallelism of map tasks.

There are two sources of parallelism on MapReduce based on its scaling characteristics: multiple map and reduce tasks can be executed in parallel *across* multiple storage nodes and *within* each storage node. The second parallelism assumes that a storage node has multiple processors (e.g., multi-core [24] or GPGPU [25]) on which multiple map or reduce tasks from one or more MapReduce jobs can be executed concurrently. The reduce function of $G_i^*(d)$ computation is not computationally intensive. So we focus on exploiting parallelism for map tasks. In all-point $G_i^*(d)$ computation, the number of data cells generated at decomposition step determines the number of MapReduce jobs, because data cells are read as query cells in job algorithm (see Table 1). For a data block *p* located on a storage node with *M* processors, at most *M* map tasks from multiple jobs can be executed simultaneously on the node. Since each map task on a storage node needs to process the same set of data cells local to the task, the cost of these map tasks is proportional to the size of query cell $|q|$ (note that the complexity of map function is $O(|q| * |c|)$. Illustrated in figure 3, if the size of query cells is not evenly distributed, large size query cells may affect the total execution time of all-point $G_i^*(d)$ computation. This is true both across and within storage nodes, especially in a proprietary MapReduce cluster environment. Without changing underlying MapReduce scheduling algorithms, application-level effort can be made to make query cell sizes even to minimize the side effect of long running jobs prolonging total execution time, therefore balancing the load of map tasks on storage nodes.

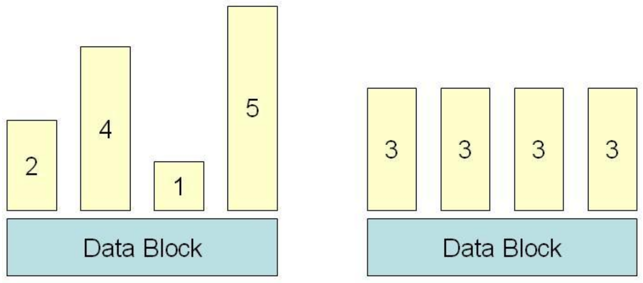

Figure 3. Effects of query cell size (i.e., number of points in a query cell) on multi-job execution time (totally 12 query points). Left: a dataset is decomposed into 4 cells of size 2, 4, 1, 5. Right: evenly decomposed cells of size 3. Left-side execution time is bounded by the job with query cell size 5.

As a result, an application-level load-balancing mechanism is developed using a quadtree decomposition algorithm [23]. At decomposition step, the maximum number of points allowed in a cell is specified as a parameter to decompose a spatial domain into cells with similar number of points, therefore producing query cells with similar sizes.

4. EXPERIMENTS

Computational experiments were conducted to evaluate our MapReduce-based $G_i^*(d)$ computation method. The correctness of $G_i^*(d)$ values computed were verified by crosschecking with the results from the $G_i^*(d)$ code published in [12]. The Hadoop environment used is provided by the Computer Network Information Center (CNIC) of the Chinese Academy of Science (CAS). Hadoop version 0.20.2 and HBase 0.20.3 are configured on 8 storage nodes. Each node is an Intel Xeon L5420 2.5GHz 8-core machine with 32GB memory and two 146GB hard disks. The operating system is CentOS 5.4 x86_64.

4.1 Dataset

A synthetic dataset was generated to evaluate computational performance (figure 4). As shown in [13], data with globally-uniform distribution exhibits stationarity. A representative spatial dataset is usually non-stationary in terms of spatial dependency and heterogeneity. Therefore, the dataset with an irregular distribution was used in our experiments and contains 1 million points that are located in a fixed square region of [0, 50] x [0, 50]. These points belong to seven clusters whose locations are uniformly distributed within the spatial domain. Within each cluster, points follow a normal distribution. Each point is characterized by location information <*x*, *y*> and an attribute vector from which *z* value is derived. Total data size is 2.56GB.

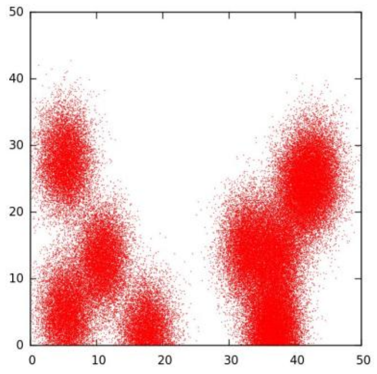

Figure 4. A clustered dataset with seven "hot spots"

4.2 Performance evaluation

Two sets of experiments were conducted to evaluate: 1) the effectiveness of the SLSI strategy for preserving spatial locality in data storage and filtering unnecessary data cells in the computation of map tasks; and 2) the impact of spatial data decomposition methods on all-point $G_i^*(d)$ computation performance.

4.2.1 Effectiveness of SLSI

$G_i^*(d)$ computation was conducted on the same dataset using two different ways: with or without SLSI applied. In the case without SLSI, spatial indexing-based input filter is not used for MapReduce jobs. Performance differences are measured from two aspects: data loading cost and map computation. The effect of distance parameter *d* is also evaluated. The dataset was decomposed into 256 cells using a region quadtree.

Figure 5 shows the difference in average data loading cost with and without the use of SLSI. The all-point $G_i^*(d)$ computation creates 256 MapReduce jobs. The number of input cells for map tasks in each MapReduce job is recorded and the average number of loaded input cells per MapReduce job is reported. With SLSI, a MapReduce job can safely skip a data cell if the nearest cell distance from the query cell is larger than *d*. As a result, the number of data cells loaded for each map function call is reduced, which also results in the reduction of map output size, input sizes of reduce tasks, and the number of pair-wise distance calculations. For example, when *d* was 5, only 36.97 of 256 data cells were loaded averagely for each MapReduce job, resulting in approximately 84.4% savings on data I/O for map task execution (i.e., saving 48.98MB map input data to load and 46.62MB map output data). Without SLSI, every data cell in a data block had to be considered for distance calculations because no prior knowledge is available about whether a data cell is within *d*-distance of a query cell. Therefore, the number of loaded input cells for map tasks is a constant without SLSI applied.

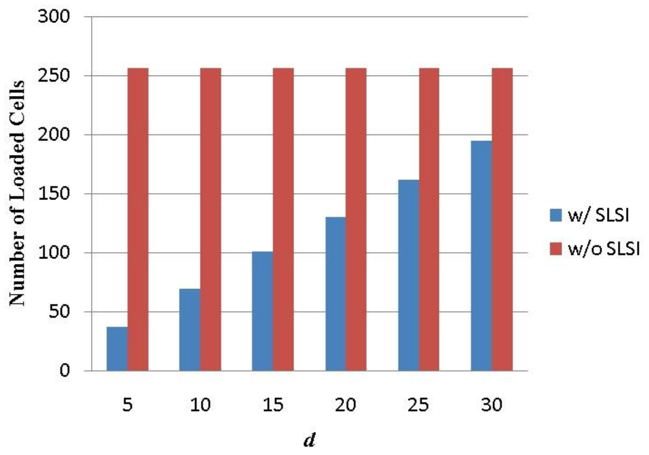

Figure 5. Average data loading performance

The performance gain obtained from using SLSI is more significant when the value of *d* is small. This is because more data cells fall out of distance *d* when *d* is small. As *d* increases, the benefit of SLSI drops because the MapReduce job includes more data cells in the computation of partial results.

Figure 6 shows the execution time with SLSI applied in $G_i^*(d)$ computation. In all-point $G_i^*(d)$ computation, multiple MapReduce jobs are created and executed in parallel (one job for each query cell), which complicates the execution picture of each individual MapReduce job and make it difficult to identify whether the change of execution time is due to the use of SLSI or the runtime interference among MapReduce jobs. Therefore, only a single MapReduce job is created for execution time measurement by randomly selecting a query cell (cell id = 150; 5251 points). As shown in figure 6, as *d* increases, longer execution time is observed because more cells are loaded as input cells for map tasks and involved in computation. A little variation on execution time is observed when *d* equals to 15, which can be explained by runtime scheduling of MapReduce. Specifically, in Hadoop, there are three types of map tasks [6]: 1) data-local map tasks that run on the storage node where data are stored; 2) rack-local map tasks that run on the same rack with the storage node where data reside; and 3) neither data-local nor rack-local. Our experiment was conducted on a single rack, so there were no type 3 tasks. Depending on runtime loads on a storage node, the scheduling module of MapReduce may send a map task to a less loaded node on the same rack. The communication involved in loading data through network contributes to the slowdown of the execution of rack-local map tasks. In test runs at *d*=15, the number of rack-local map tasks created by MapReduce was more than that at other *d* values, thus slowing down the computation.

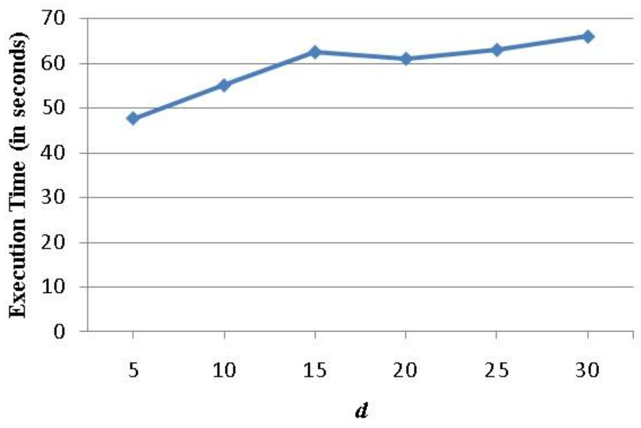

Figure 6. MapReduce job execution time with SLSI on a randomly selected query cell

4.2.2 *Effective load-balancing through spatial domain decomposition*

This experiment evaluates the performance of all-point $G_i^*(d)$ computation by comparing two decomposition methods: 1) regular quadtree decomposition, and 2) adaptive quadtree decomposition. SLSI is not used in this experiment to eliminate the effect of data cell filtering. Therefore, the number of data cells loaded in map tasks is same in both cases where 256 quads (cells) are created. For the regular quadtree decomposition, a 16*16 grid is generated. For the adaptive quadtree decomposition, the maximum number of points allowed in each quad is set to 9520 in order to create the same number of quads (figure 7), but with more balanced number of points in each quad, as illustrated in Table 2. The value of *d* was set to be 10. Figure 8 shows the performance gained by evenly distributing points into cells, as a benefit of adaptive decomposition. On average, total execution time is reduced by 15.43% through the use of adaptive decomposition. Note that the reduction on execution time is mainly due to the balanced query cell sizes which result in balanced computing time on each data block distributed among storage nodes for the parallel execution of MapReduce jobs.

Table 2. Distribution of the number of points in data cells

Number of points in cell	Number of cells	
	Adaptive decomposition	Regular decomposition
0	2	72
1-1000	20	78
1001-2000	25	17
2001-5000	134	26
5001-10000	75	25
10001-20000	0	28
20001-	0	10

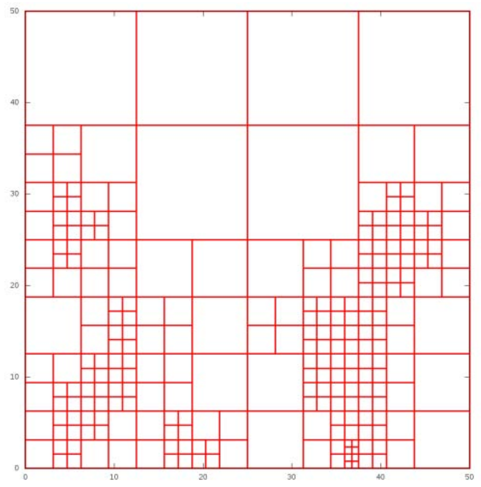

Figure 7. Adaptive quadtree decomposition (256 quads, the maximum number of points allowed in a quad is 9520)

The sensitivity of computational performance against underlying storage orders was also evaluated. Three orders were examined: 1) space-filling curve (Z-order in particular); 2) random order; and 3) interleaving the storage of decomposed cells based on the number of storage nodes (i.e., 8) [26]. From figure 8, the computational performance of both regular and adaptive decomposition methods is affected by storage orders to small extents. As discussed in section 3.2, the decomposition method chosen at decomposition step decides the number of data cells generated and the number of points in each data cell. At storage step, different storage order affects the distribution of decomposed data cells among data blocks. Therefore, a data block may contain different data cells based on different designated storage order. However, although the distribution of data cells on data blocks may change where a data cell is loaded and computed, it does not change the total number of data cells to be loaded for calculating $G_i^*(d)$ values for a query cell. Furthermore, data blocks have similar size and are stored in a balanced way by MapReduce. Therefore, data loading cost do not differ significantly on different storage orders. As long as storage nodes are homogeneous in architecture and system configurations, which is the case in our experiments, the overall computational cost by all map function calls do not change. Collectively, the total execution times on the MapReduce system for all-point $G_i^*(d)$ computation using different storage orderings do not differ significantly. Variations on execution time shown in figure 8 are due to runtime dynamics in the parallel execution of MapReduce jobs and variations on individual data blocks.

5. CONCLUDING DISCUSSION

This paper employed the MapReduce framework to develop a scalable solution to the computation of a local spatial statistic: $G_i^*(d)$ based on the analysis of the computational intensity of $G_i^*(d)$. The computational performance of the solution is significantly improved by exploiting spatial characteristics embedded in spatial data. Through the investigation of the relationship between spatial locality and storage locality, a spatial locality-based storage and indexing mechanism is developed to eliminate unnecessary computation of map tasks through spatial domain decomposition-based spatial data storage and the application of a spatial index based on a cell distance matrix. In

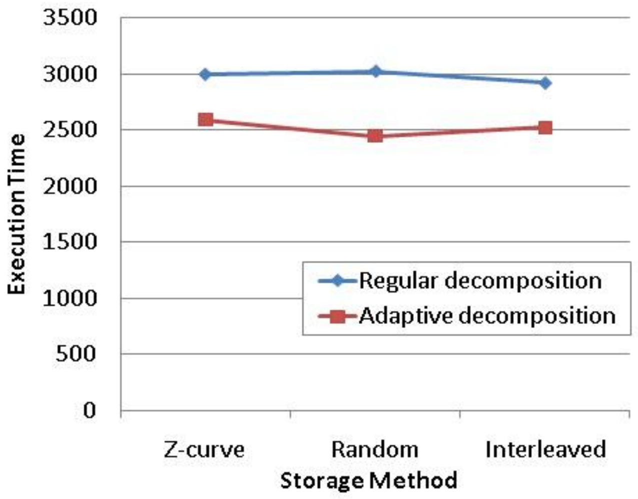

Figure 8. Average execution time (in seconds) with different decomposition methods and different storage orders

addition, an application-level load balancer is developed to produce evenly sized map task loads and, thus, facilitate more efficient computation of all-point $G_i^*(d)$ computation that involves multiple MapReduce jobs.

As for future work, scalability tests will be conducted on large MapReduce clusters for massive data processing. Data I/O performance on larger datasets will be evaluated to fine-tune our algorithms to optimize query data I/O performance. Although the strategies developed in this paper for computational performance improvement do not depend on underlying MapReduce system policies, we will investigate the impact of tuning MapReduce data assignment and replication, as well as job scheduling algorithms. The MapReduce algorithm and spatially-explicit strategy developed in this paper focus on all-point $G_i^*(d)$ computation. In reality, query-based usage pattern is more common. Users are interested in a particular part of spatial domain. We will study the impact of query-based $G_i^*(d)$ computation on MapReduce to achieve specified quality of query response time, especially in a shared MapReduce environment in which multiple queries are served simultaneously. Also included in our future work, other spatial analysis methods will be investigated using MapReduce..

6. ACKNOWLEDGMENTS

This research was supported in part by the National Science Foundation through the award BCS-0846655. The Hadoop cluster used in our experiments is supported by the Computer Network Information Center (CNIC) of the Chinese Academy of Sciences.

7. REFERENCES

[1] (NRC), N. R. C. 2010. *Understanding the Changing Planet: Strategic Directions for the Geographical Sciences*. Retrieved Jun, 2010, from http://www8.nationalacademies.org/onpinews/newsitem.aspx?RecordID=12860.

[2] Skomoroch, P., Weil, K. and Gorman, S. 2010. Spatial Analytics Workshop. *Where 2.0 conference* (San Jose, California, March 30, 2010).

[3] Burrough, P. A. and McDonnell, R. 1998. *Principles of Geographical Information Systems, Second Edition*. Oxford University Press, New York, NY, USA.

[4] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R. E. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2 (2008), 1-26. DOI= http://doi.acm.org/10.1145/1365815.1365816.

[5] Dean, J. and Ghemawat, S. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107-113. DOI= http://doi.acm.org/10.1145/1327452.1327492.

[6] Apache. 2010. *The Apache Hadoop Project*. Retrieved Jun 12, 2010, from http://hadoop.apache.org/core/.

[7] Maged, M., Jose, E. M., Doron, S. and Robert, W. W. 2007. Scale-up x Scale-out: A Case Study using Nutch/Lucene. *International Parallel and Distributed Processing Symposium (IPDPS)* (Long Beach, CA, USA, March 26-30, 2007), 441-441.

[8] Greer, R. 1999. Daytona and the fourth-generation language Cymbal. *Proceedings of the 1999 ACM SIGMOD international conference on Management of data* (Philadelphia, Pennsylvania, United States, 1999). ACM, 304242, 525-526. DOI= http://doi.acm.org/10.1145/304182.304242.

[9] Cary, A., Sun, Z., Hristidis, V. and Rishe, N. 2009. Experiences on Processing Spatial Data with MapReduce. *Proceedings of the 21st International Conference on Scientific and Statistical Database Management* (New Orleans, LA, USA, 2009). Springer-Verlag, 1561669, 302-319. DOI= http://dx.doi.org/10.1007/978-3-642-02279-1_24.

[10] Wang, S. and Armstrong, M. P. 2009. A theoretical approach to the use of cyberinfrastructure in geographical analysis. *International Journal of Geographical Information Science* 23, 2 (2009), 169 - 193.

[11] Getis, A. and Ord, J. K. 1992. The analysis of spatial association by use of distance statistics. *Geographical Analysis* 24, 3 (1992), 189-206.

[12] Wang, S., Cowles, M. K. and Marc P. Armstrong. 2008. Grid computing of spatial statistics: using the TeraGrid for Gi*(d) analysis. *Concurrency and Computation: Practice and Experience* 20, 14 (2008), 1697-1720.

[13] Getis, A. and Ord, J. K. 1996. *Loacl Spatial Statistics: An Overview*. In *Spatial Analysis: Modelling in A GIS Environment*, P. Longley and M. Batty Ed. Cambridge: Geoinformation Interntional, 261-277.

[14] Ord, J. K. and Getis, A. 1995. Local spatial autocorrelation statistics - Distributional issues and an application. *Geographical Analysis* 27, 4 (1995), 286-306.

[15] Wang, S. and Armstrong, M. P. 2005. A Theory of the Spatial Computational Domain. *Proceedings of GeoComputation 2005 (CDROM)* (Ann Arbor, MI, USA, August 1-3, 2005).

[16] Wang, S. 2010. A CyberGIS Framework for the Synthesis of Cyberinfrastructure, GIS, and Spatial Analysis. *Annals of the Association of American Geographers* 100, 3 (2010), 1-23.

[17] Zhang, S., Han, J., Liu, Z., Wang, K. and Feng, S. 2009. Spatial Queries Evaluation with MapReduce. *The 8th International Conference on Grid and Cooperative Computing* (Lanzhou, Gansu, China, August 27-29, 2009), 287-292.

[18] Hadoop. 2010. *The Hadoop Distributed File System: Architecture and Design*. Retrieved Jun 12, 2010, from http://hadoop.apache.org/core/docs/r0.16.4/hdfs_design.html.

[19] Chen, Q., Wang, L. and Shang, Z. 2008. MRGIS: A MapReduce-Enabled High PerformanceWorkflow System for GIS. *The 3rd International Workshop on Scientific Workflows and Business Workflow Standards in e-Science (SWBES)* (Indianapolis, IN, 2008).

[20] Stupar, A., Michel, S. and Schenkel, R. 2010. RankReduce - Processing K-Nearest Neighbor Queries on Top of MapReduce. *The 8th International Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR 2010)* (Geneva, Switzerland, July 23, 2010).

[21] Jordan, L. E. and Alaghband, G. 2002. *Fundamentals of Parallel Processing*. Prentice Hall Professional Technical Reference.

[22] Sagan, H. 1994. *Space-filling Curves*. Springer-Verlag, New York, NY.

[23] Wang, S. and Armstrong, M. P. 2003. A quadtree approach to domain decomposition for spatial interpolation in Grid computing environments. *Parallel Computing* 29, 10 (2003), 1481-1504. DOI= 10.1016/j.parco.2003.04.003|ISSN 0167-8191.

[24] Yoo, R. M., Romano, A. and Kozyrakis, C. 2009. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)* (2009). IEEE Computer Society, 1680796, 198-207. DOI= http://dx.doi.org/10.1109/IISWC.2009.5306783.

[25] He, B., Fang, W., Luo, Q., Govindaraju, N. K. and Wang, T. 2008. Mars: a MapReduce framework on graphics processors. *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (Toronto, Ontario, Canada, 2008). ACM, 1454152, 260-269. DOI= http://doi.acm.org/10.1145/1454115.1454152.

[26] Shafer, J., Rixner, S. and Cox, A. L. 2010. The Hadoop distributed filesystem: Balancing portability and performance. *2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2010)* (White Plains, NY, March 28-30, 2010, 2010), 122-133. DOI= 10.1109/ISPASS.2010.5452045.