MapReduce algorithms for GIS Polygonal Overlay Processing

Satish Puri, Dinesh Agarwal, Xi He, and Sushil K. Prasad
Department of Computer Science
Georgia State University
Atlanta - 30303, USA
Email: spuri2, dagarwal2, xhe8@student.gsu.edu, sprasad@gsu.edu

***Abstract*—Polygon overlay is one of the complex operations in computational geometry. It is applied in many fields such as Geographic Information Systems (GIS), computer graphics and VLSI CAD. Sequential algorithms for this problem are in abundance in literature but there is a lack of distributed algorithms especially for MapReduce platform. In GIS, spatial data files tend to be large in size (in GBs) and the underlying overlay computation is highly irregular and compute intensive. The MapReduce paradigm is now standard in industry and academia for processing large-scale data. Motivated by the MapReduce programming model, we revisit the distributed polygon overlay problem and its implementation on MapReduce platform. Our algorithms are geared towards maximizing local processing and minimizing the communication overhead inherent with shuffle and sort phases in MapReduce. Experimental results validate the method and show its effectiveness as good speedup and scalability are both achieved in large-scale overlay processing.**

I. Introduction

Computational geometry algorithms are used in different fields. One of the complex geometric algorithms used in GIS setting is polygonal overlay, which is the process of superimposing two or more polygon layers to produce a new polygon layer. In GIS, overlay operation is a key analysis function used in combining the spatial and attribute data of two input map layers. Typically, two different thematic maps of the same area are taken and overlaid one on top of the other to form a new map layer. Polygon overlay is a type of map overlay but on GIS polygonal (also known as "vector") data where the spatial input data is represented in the form of points, lines and polygons and it is much more complex than raster data computation.

Real world data in GIS is stored in two formats - raster and vector. Raster data stores the information as pixels in an image. Vector data represents geographic features as points, lines, and polygons. Vector data is represented in different file formats such as GML files, shapefiles, etc. In this work, we concentrate on polygon overlay over vector-based GIS data. In general, map overlay supports different operators such as Union, Intersection, Xor, etc.

In some cases when large volumes of spatial data is deployed for overlay analyis (see Figure 1), it becomes a time consuming task, which sometimes is also time sensitive. GIS scientists use desktop based sequential GIS systems for overlay tasks. The desktop GIS software typically takes hours to perform overlay for large data sets which makes it useless for real time policy decisions. For emergency response in the US, for example, disaster-based consequence modeling is predominantly performed using HAZUS-MH, a FEMA-developed application that integrates current scientific and engineering disaster modeling knowledge with inventory data in a GIS framework. Depending on the extent of the hazard coverage, data sets used by HAZUS-MH have the potential to exceed the capacity of standard desktops for detailed analysis, and it may take several hours to obtain the results.

In this paper, we present the adaptation and implementation of a polygon overlay algorithm (Algorithm 1), which originates from the area of Geographical Information Systems. We describe a system to execute distributed version of this algorithm on a Linux cluster using Hadoop MapReduce framework. Parallelizing polygonal overlay with MapReduce has its own set of challenges. First, MapReduce focuses mainly on processing homogeneous data sets, while polygon overlay has to deal with two heterogeneous data sets. Second, the polygon overlay operations are complex and time-consuming, and vector objects are generally larger and more complex than words or URL strings in common MapReduce applications. Third, partitioning vector data evenly to yield good load balance is non-trivial. Our parallel system is based on R-tree data structure which works well with non-uniform data. Our system carries out end-to-end overlay computation starting from two input GML (Geography Markup Language) files, including their parsing, employing the bounding boxes of potentially overlapping polygons to determine the basic overlay tasks, partitioning the tasks among processes, and melding the resulting polygons to produce the output GML file. We describe the software architecture of our system to execute the algorithm and discuss the design choices and issues. We analyze and report *Intersection* overlay operation in our paper since it is the most widely used and representative operation. Nevertheless, our system can be extended to other operations as well without any change. Our rigorous instrumentation of the timing characteristics of various phases of the algorithm point out portions of the algorithm which are easily amenable to scalable speedups and some others which are not. We have experimented with two data sets with uniform and skewed load distribution and achieved upto 22x relative speedup with larger

[1]This work is partially supported by NSF CCF1048200 project

Source	Example Type	Description	File Size
US Census	Block Centroids	Block centroids for entire US	705 MB
	Block Polygons	2000 Block polygons for the state of Georgia	108 MB
	Blockgroup Polygons	2000 Blockgroup polygons for the state of Georgia	14 MB
GADoT	Roads	Road centerlines for 5-county Atlanta metro	130 MB
USGS	National Hydrography Dataset	Hydrography features for entire US	13.1 GB
	National Landcover Dataset	Landcover for entire US	3-28 GB
JPL	Landsat TM	pan-sharpened 15m resolution	4 TB
Private	LIDAR	LIDAR point clouds 1-4 pts/sq. ft	0.1-1 TB

Fig. 1: Example GIS data sets and Typical File Size Ranges

data set using 64 CPU cores and a relative speedup upto 13x with smaller data set using 32 CPU cores.

We have implemented overlay algorithms in MapReduce in three different forms, namely, i) with a single map and reduce phase, ii) with chaining of MapReduce jobs, and iii) with a single map phase only. We also show performance comparison among these three different versions. In the literature, we found bottom-up construction of R-tree by bulk loading strategies implemented using MapReduce. In this paper, we present a top-down construction of R-tree in a distributed setting using a grid based approach. Our specific technical contributions include

- porting the MPI based spatial overlay system [1] to Hadoop MapReduce platform,
- an overlay algorithm with chaining of MapReduce jobs,
- a distributed top-down construction of R-tree using MapReduce and an overlay algorithm based on it, and
- an overlay algorithm with map phase only and without spatial indexing.

The rest of this paper is organized as follows: Section II reviews the literature briefly and provides background on GIS data, R-tree data structure, general polygon clipper (GPC) library, MapReduce framework and expresses polygon overlay as a weighted graph problem. Section III describes four (including naive algorithm) MapReduce based algorithms. Our experimental results and other experiments are in IV. Section V concludes this paper.

II. Background and Literature

A. Data Types in GIS

In GIS the real world geographic features are prominently represented using one of the two data formats: raster and vector. Raster form stores the data in grids of cells with each cell storing a single value. Raster data requires more storage space than vector data as the representation cannot automatically assume the inclusion of intermediate points given the start and end points. In vector data, geographic features are represented as geometrical shapes such as points, lines, and polygons. Polygons are composed of multiple line segments, and one line segment connects another line segment at its endpoint. Vector data is represented in GIS using different file formats such as GML, shapefile, etc. In our experiments, we use GML file format (XML-based) to represent input and output GIS data.

B. Map Overlay

Map Overlay is one of the key spatial operations in GIS. It is the process of interrelating several spatial features (points, lines, or polygons) from multiple data sets, which creates a new output vector dataset, visually similar to stacking several maps of the same region together. For instance, one map of United Sates representing population distribution and another map representing the area affected by hurricane Sandy can be overlaid to answers queries such as "What is the optimal location for a rescue shelter?". Clearly, it is often needed to combine two or more maps according to logical rules called overlay operations in GIS. A Union overlay operation, for instance, combines the geographic features and attribute tables of both inputs into a single new output. An Intersection overlay operation, similarly, defines the overlapping area and retains a set of attribute fields for each. In case of raster data format, the operation is called grid overlay and in the case of vector data format the operation is called polygon overlay. It should be noted here that the process of map overlay in case of raster data is entirely different from that of vector data and our solution deals with vector data. Since resulting topological features are created from two or more existing features of the input map layers, the overlay processing task can be time consuming.

Algorithm 1 presents the high level sequential algorithm for polygon overlay.

Algorithm 1 Sequential Polygon Overlay algorithm [1]

INPUT: Set of Base layer polygon A and set of overlay layer polygons B
OUTPUT: Set of Ouptut polygons O
for all polygon p_A in S_A **do**
  Step 1: find all intersecting polygons I from set B
  Step 2: for each p_B in I
  Step 3: compute p_O = overlay (p_A, p_B)
  Step 4: add polygon p_O to O
end for

When we perform overlay between two map layers, a polygon from base layer A can spatially overlap with zero or more polygon(s) from overlay layer B. This overlapping

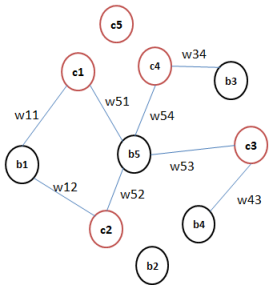

Fig. 2: Weighted Intersection Graph

relationship between polygons from two different layers can be represented in the form of intersection graph. Let us consider a graph $G = (V, E)$ where V is set of polygons, $V = \{A \bigcup B\}$ and E is the set of edges between polygons such that $\forall$ u,$v \in V$, $u \in A$ and $v \in B$, there exists an edge (u, v) in G if and only if u overlaps with v. Figure 2 shows an example of an intersection graph where b_i denotes base polygons and c_i denote overlay polygons for $1 \leq i \leq 5$. As we can see in this figure, polygon $b5$ has the highest degree and it overlaps with 4 other polygons and $b2$ and $c5$ polygons that do not overlap with other polygons, have 0 degree.

The polygonal data has high degree of irregularity. The size of polygons in terms of number of vertices varies from less than a hundred to more than a thousand. The complexity of overlay operation is captured by a weight function w. Every edge in the graph G has weight w_{ij} which denotes the weight corresponding to the connecting edge where i and j are indices of base and overlay polygon respectively.

C. R-tree Data Structure

R-tree is an efficient spatial data structure for rectangular indexing of multi-dimensional data. The basic idea is to group nearby objects and represent them with their minimum bounding rectangle in the next higher level of the tree. R-tree data structure provides standard functions to insert and search spatial objects by their bounding box co-ordinates. Searching for overlap in R-tree is typically an $O(log_m n)$ operation where *n* is the total number of nodes and *m* is the number of entries in a node. However, it can result in $O(n)$ complexity in the worst case. We use Guttman's algorithm [2] for R-tree construction and search operations. R-tree can be constructed in top-down fashion or by using bulk-loading bottom-up techniques.

D. Clipper Library

For computing basic overlay of a pair of polygons, we use the General Polygon Clipper (GPC) library which is an implementation of polygon overlay methods [3]. The input to this library is a pair of polygons and an overlay operator like Intersection, Union, Difference, and XOR. The GPC library handles polygons that are convex or concave and self-intersecting. It also supports polygons with holes. The usage of this widely used library leads to accuracy of our approach for our polygon overlay solution.

E. MapReduce Framework

The cloud computing framework based on Hadoop MapReduce provides a promising solution to solve the problems involving large-scale data sets. The input data is viewed as a stream of records comprising of key-value pairs. As the name suggests, in MapReduce there are two phases namely map phase and reduce phase. A map phase consists of map tasks where each map task can run in parallel and works on a portion of the input. A map task produces intermediate key-value pairs which are processed in parallel by reduce tasks in reduce phase. The reduce phase of a job cannot begin until the map phase ends. Under the MapReduce programming model, a developer only needs to provide implementations of the mapper and reducer. The framework takes care of task scheduling, data locality and re-execution of the failed tasks. Google′s MapReduce and Apache Hadoop are two popular implementations of MapReduce. In Hadoop, both the input and the output of the job are stored in a file-system known as Hadoop Distributed File System (HDFS).

In MapReduce, synchronization is achieved by the shuffle and sort barrier. Intermediate results generated are aggregated by reducers. Combiners perform local aggregation, and partitioners determine to which reducer intermediate data are shuffled to.

F. Overlay and Miscellaneous Approaches

For identification of overlaying map-features, different algorithms based on uniform grid, plane sweep, Quad-tree, and R-tree, have been proposed and implemented on classic parallel architectures [4]–[7]. Edge intersection detection is a subproblem of computing polygon overlay and most of the research aims to parallelize edge intersection detection phase only. Franklin et al. [4] presented the uniform grid technique for parallel edge intersection detection. Their implementation was done using Sun 4/280 workstation and 16 processor Sequent Balance 21000. Waught et al. [8] presented a complete algorithm for polygon overlay and the implementation was done on Meiko Computing Surface, containing T800 transputer processors using Occam programming language. Data partitioning in [4], [8]–[10] is done by superimposing a uniform grid over the input map layers.

After surveying the related literature, we found that there are no papers on MapReduce based polygon overlay problem to the best of our knowledge. Spatial join is similar to polygonal overlay and it is used for joining attributes from one feature to another based on the spatial relationship. MapReduce based algorithm for spatial join problem is discussed in [11]. Their strategies include strip based plane sweeping algorithm and tile-based spatial partitioning function. Tile based partitioning is used to partition the data into multiple uniform tiles and these tiles are assigned to map tasks in a round-robin fashion. Tile based partitioning may result in high overhead for replicating polygons across tiles for skewed data. Spatial analysis service system based on MapReduce programming model is proposed in [12]. A high level MapReduce algorithm for overlay analysis and R-tree index building for raster data

is also presented but the paper does not include performance evaluation. The authors in [13] and [14] propose a method of bulk-loading spatial data for R-tree generation using the MapReduce framework. We presented a parallelization scheme for polygonal overlay based on Message Passing Interface (MPI) in [1]. [15] and [16] discusses static and dynamic load balancing strategies used in parallelizing map overlay on Azure cloud platform. But these parallelization schemes are not applicaple to Hadoop-based MapReduce system since the latter lacks explicit message passing as in MPI framework (point to point send/receive) or the queues as in Azure for communication.

III. Map Reduce Algorithms

A. Problem Definition

The polygon overlay combines the input polygons from two different maps into a single new map. The input to binary map overlay are two map layers L_1 = [p_1, p_2, .. , p_n] and L_2 = [q_1, q_2, .. , q_m] where p_i and q_i are polygons represented as *x*,*y* co-ordinates of vertices. The output of the overlay operation is a third layer $L_3 = L_1 \times L_2$ = [o_1, o_2, ... , o_k] represented by *k* output polygons and this output depends on the overlay operator denoted as $\times$. Various overlay operators for example Union, Intersection etc determine how map layers are combined.

Our system has a four-step work flow which consists of input file parsing, task creation, overlay computation, and output file creation. Here, we present four different algorithms using Hadoop MapReduce framework.

File partitioning in hadoop distributed filesystem (HDFS) is based on input split size which can be 64 MB or 128 MB. Once split size is decided, hadoop internally takes care of input file partitioning. The input in our application consists of GML files which contain polygon vertices and other spatial attributes spanning multiple lines bounded by XML tags. The straightforward partitioning of such a file based on split size may lead to one polygon spanning multiple input splits which will produce incorrect result. In order to ensure proper splitting of input data, we preprocess the input file such that a polygon is contained in a single line. Even then the splitting phase can result in one line getting split at an arbitrary position, but in this case hadoop framework ensures that a line is completely contained in one input split only. A preprocessing MapReduce job can be executed to extract the polygon ID, Minimum Bounding Rectangle (MBR) and polygon vertices from each spatial object and these spatial attributes can be written in a single line for each object. Finally, the preprocessed input files can be sent to HDFS for further processing. This preprocessing phase is done offline.

B. Algorithm With both Map and Reduce Phases

The input to the Algorithm 2 is set of polygons from a base layer file and an overlay layer file which is stored in HDFS. Since there are two input files, a source tag is required to identify a polygon′s parent data set. Every mapper receives a 64 MB file split from base layer or overlay layer as input. The input file split belongs to a single map layer and each mapper simply emits the polygon as key-value pair depending upon whether the polygon belongs to base layer or overlay layer. Since, the overlap relationship or intersection graph for polygons is not known at this stage, a brute force algorithm is applied here and the polygons are emitted in such a way so that in the reduce phase a base polygon is grouped with all overlay layer polygons for intersection detection. The actual overlay computation is thus delegated to reduce phase thereby ignoring local computation in map phase and carrying forward all the polygonal data to reduce phase.

Algorithm 2 Naive MapReduce overlay algorithm

```
method MAP (id, polygon p)
{
extract tag from p
if p is from base layer then
   EMIT: (id, value p)
else
   for all base polygon b in [b1,b2, ..] do
      EMIT: (id of b, value p)
   end for
end if
}

method REDUCE (base polygon id, [c1, c2, . . ])
{
retrieve base polygon b from [c1, c2, . . ]
find intersecting polygon set M from [c1, c2, . . ]
for all c ∈ M do
   compute overlay (b, c)
   EMIT: output polygon
end for
}
```

Observation: The disadvantage of this algorithm is that it produces too many $key\text{-}value$ pairs and since intermediate $key\text{-}value$ pairs are spilled to disks and thus involves writing to files thereby degrading the overall performance. This algorithm does not perform well because of the overhead of task creation and all-to-all communication between Map and Reduce phases.

It is non-trivial to efficiently parallelize binary applications requiring two input sources in MapReduce framework. In map overlay which is a binary application, data locality may not be possible since default input file splitting may result in mapper tasks having a split from a single input source. In the absence of input chunks from both input sources in a mapper, overlay processing can not be done in the map phase itself. Instead, the data is simply forwarded to reduce phase for overlay processing which results in extra communication overhead. If a mapper possesses splits from both input sources, local overlay processing can be done but since it is possible that polygons in one mapper may overlap with polygons in other mappers, a chain of MapReduce jobs or more than one iterations are

required to perform overall overlay in a distributed fashion.

C. Chained Map Reduce Algorithm

In algorithm 3, the overall work is divided in two distinct MapReduce jobs. The first job has only a map phase in which the bounding box of base layer polygons are emitted as output in order to aid in intersection detection carried out in the second MapReduce job. The output of the first job serves as input for the second job where all the mapper tasks create a local R-tree from the bounding boxes of the base layer polygons.

Communication of polygons across different reduce tasks (during shuffle and sort phase) has its own overhead as such this algorithm is geared towards processing overlay operation locally in map phase itself. Moreover, our objective is to decrease the number of reducers in comparison to the naive algorithm.

In addition to *map* and *reduce* methods, hadoop MapReduce framework provides *initialize* or *setup* and *cleanup* methods which can be overridden by application developers. *initialize* is called once at the beginning of the map task and *cleanup* is called at the end of the map task. Instead of computing overlay in reduce phase, a combiner phase can be used for local computation in map phase itself. Instead, we implement In-Mapper Combining design pattern [17] by using *cleanup* method. The usage of this design pattern reduces the intermediate $key\text{-}value$ pairs and it is shown to perform better than using a combiner phase. Since *cleanup* is called after all the base and overlay layer polygons are read by a mapper, polygon overlay can be locally computed in a distributed fashion.

Let us consider an example with four base layer polygons (b1 to b4) and five overlay layer polygons (c1 to c5) to illustrate the algorithm with regard to the communication of polygons in map, shuffle and reduce phase of this algorithm. A base layer polygon can potentially overlay with zero or more overlay layer polygons and this overlap relationship is shown in table I as an intersection graph represented as an adjacency list.

Base layer polygon	Overlay layer polygon
b1	c3
b2	c4,c5
b3	c1
b4	-

TABLE I: Intersection graph represented as an adjacency list

The advantage of having a local R-tree in all the mapper tasks is that each mapper can independently determine those overlay layer polygons which do not overlap with any base layer polygon and as such can be discarded thereby preventing such polygons from going to shuffle and reduce phase. Moreover, in a reducer with $key\text{-}value$ pair $< k, v >$, for a given key k (base polygon id), the value v consists of only those overlay layer polygons that can potentially intersect with the base polygon with id k.

Algorithm 3 Chained Map Reduce Algorithm

JOB I
method MAP (id, polygon p)
{
Read Polygon p
if p is from base layer **then**
  extract bounding box of p
  EMIT: (id_p, bounding box of p)
end if
}

JOB II
method INITIALIZE
{
Initialize R-tree
Read bounding boxes from Distributed Cache
Insert bounding boxes to R-tree
}

method MAP (Polygon p)
{
Read p
if p is from base layer **then**
  extract id from p and store in list B
  EMIT: (id, p)
else
  parse p and store in a list C
end if
}

method CLEANUP (list B, list C)
{
for all clip polygon $p_c \in C$ **do**
  get intersecting polygon id list L from R-tree
  for all base polygon $p_b \in L$ **do**
    if p_b is present locally in B **then**
      compute overlay (p_b,p_c)
    else
      EMIT: (id_b, p_c)
    end if
  end for
end for
}

method REDUCE (base polygon id, [p_1, p_2, ..])
{
extract base layer polygon p_b from [p_1, p_2, ..]
for each overlay layer polygon p_c in [p_1, p_2, ..]
  compute overlay (p_b, p_c)
  EMIT: output polygon
}

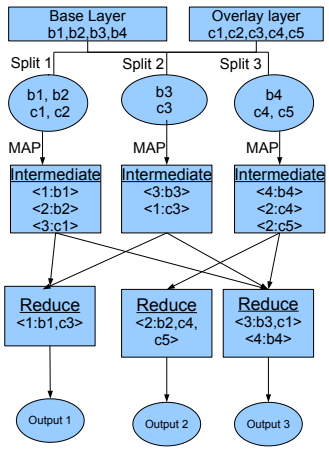

Fig. 3: Illustration of Chained MapReduce Overlay Algorithm

D. Map Phase Only Algorithm

MapReduce framework provides a facility known as DistributedCache to distribute large, read-only files. The framework copies the files on to the slave nodes before any tasks for the job are executed on that node. It is a broadcast based algorithm. In some scenarios, we can exploit an optimization when one of the input file is small enough to fit entirely in memory. The optimization results from the fact that data transfer overhead involved in transferring key,value pairs from Mappers to Reducers is avoided. In some real-world GIS application, when a small base layer file has to be overlaid on top of larger clip layer file, then this base layer file can be simply replicated on all the machines. Hadoop provides -files or -archive directive to send the input file to all the slave machines while starting a job. In algorithm 4 the base layer file is broadcasted to all the slave nodes. This algorithm has memory limitation since base layer polygons needs to be in-memory.

Each mapper loads the base layer file into its local memory and calls the map function for each tuple. It should be noted that the base layer file is not partitioned among the slave nodes as a result of which individual map tasks in this algorithm lose much of the advantage of data locality. But, overall job gains due to the potential for the elimination of reduce phase along with the elimination of the shuffle and sort after map phase. For the sake of efficient query, spatial index is created using quadtree or R-tree. Spatial index creation has its own overhead. This algorithm avoids the overhead of spatial index creation and relies on sorting of polygons to find potentially intersecting polygons.

E. Distributed R-tree based overlay algorithm

In the algorithm 3, every mapper creates an R-tree redundantly. The disadvantage of this approach is that for very large number of polygons, there may be memory constraints. In this algorithm, we partition the polygonal data in such a way that each reducer has to build an R-tree from the polygons lying in the given partition only. In this way, a distributed version of R-tree can be created in a top-down fashion starting from the root by inserting the bounding boxes iteratively to the R-tree.

Algorithm 4 Map Phase Only Overlay Algorithm

```
method INITIALIZE
{
Read base polygon list B from the distributed cache
Sort B on x-coordinate of lower bounding box of polygons
}

method MAP (id,polygon p)
{
Read p
if p is from overlay layer then
  x ← x-coordinate of upper bounding box of p
  Index n ← BinarySearch (B, x)
  L ← {b_0, b_1, .. , b_n}, b_i ∈ B
  for all polygon b_i ∈ L do
    compute overlay (b_i, p)
    EMIT: output polygon
  end for
end if
}
```

Initially, the dimension of the grid is determined which is a minimum bounding box spatially containing all the polygons from base layer and overlay layer. Then, the dimension of grid cells is computed based on the number of partitions. The number of partitions should be greater than the reduce capacity of the cluster in order to ensure proper load-balancing. A polygon may belong to one or more partitions and since we know the bounding box of all the grid cells, each mapper task can independently determine to which partition(s) a polygon belongs to. Here, each partition is handled by a single reducer. After all the mappers are done, each reducer gets a subset of base layer and overlay layer polygons. Then, an R-tree is built from the base layer polygons and for every overlay layer polygon, the intersecting base layer polygons are found out by querying the R-tree. Finally, the overlay operation is carried out using clipper library and output polygons are written to HDFS.

IV. Experimental Setup

The cluster used in the experiments is a heterogenous linux cluster containing (i) eight nodes with 8 CPU cores (ii) one node with 16 CPU cores, (iii) one node with 32 CPU cores and (iv) one node with 64 CPU cores. In our cluster all the nodes share same file system hosted at the head node. We have installed Apache's Hadoop version 1.02, which is an open source implementation of the MapReduce programming model. HDFS has a master/slave architecture consisting of a namenode which acts as a master server that manages the file system namespace and a number of datanodes that manage storage attached to the nodes. One of the node with

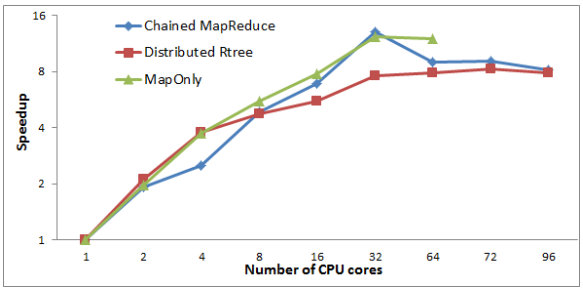

Fig. 4: Performance impact of varying worker nodes for overlay algorithms using uniform data set

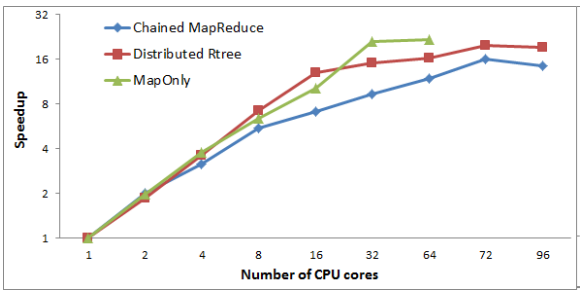

Fig. 5: Performance impact of varying worker nodes for overlay algorithms using skewed data set

8 CPU cores is configured for running namenode, secondary namenode node and job tracker. The rest of the compute nodes are configured for running datanodes and tasktrackers. We perform timing experiments with real world data sets with skewed and uniform load distribution as shown in table II.

Data set	Map layer	Size (MB)	Number of polygons
Skewed Data set	Base layer	32	4332
	Overlay layer	2300	1000000
Uniform Data set	Base layer	550	300000
	Overlay layer	770	750000

TABLE II: Description of uniform and skewed data set

Figure 4 and 5 shows the relative speedup of the three versions of overlay algorithm for uniform and skewed data sets. Since the size of skewed data set is larger than uniform data set, all of the three overlay algorithms show better speedup for skewed data set in comparison to the uniform data set. For the map-phase-only algorithm, only one of the input layer is partitioned across the mappers and the other layer is entirely in distributed cache of every mapper, as such parallelism is limited resulting in speedup getting saturated upto 32 cores only. The amount of parallelism available in the map-phase-only algorithm is directly proportional to the number of mappers that can concurrently run in the cluster. The number of mappers is equal to the number of partitions (input splits) of the input file which is thirty six ((2300 + 32)/ 64) in case of skewed data, and twelve ((770 + 550)/64) in case of uniform data. As such the skewed data shows better speedup in comparison to uniform data.

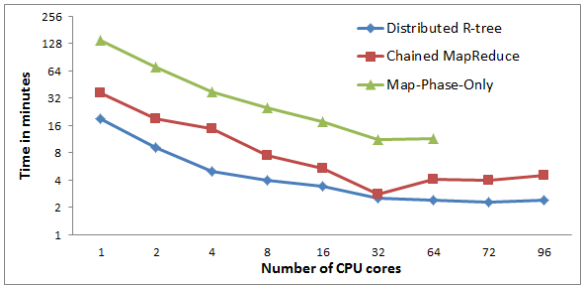

Fig. 6: Execution time for different overlay algorithms using uniform data

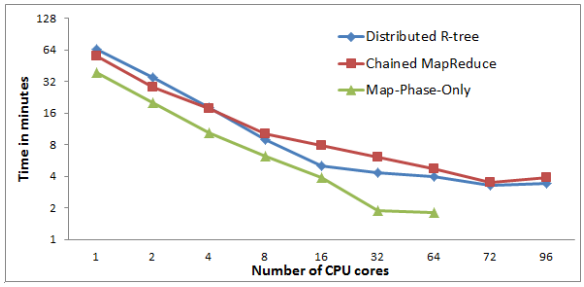

Fig. 7: Execution time for different overlay algorithms using skewed data

Figure 6 and figure 7 are the timing plots for different algorithms using uniform and skewed data sets respectively which shows that the total time taken to complete the overlay operation as the number of CPU cores increase. In both figures, the time taken to process larger skewed data set is greater than smaller uniform data sets for Chained MapReduce algorithm and Distributed R-tree algorithm. On the other hand, for map-phase-only algorithm, the time taken to process smaller uniform data set is greater than skewed data set. This anomalous behaviour can be explained by the cost of accessing larger data using distributed cache in case of uniform data. In case of skewed data set, only a 32 MB file is in-memory in all the mappers which is seventeen times smaller than the size of uniform data set which is 550 MB. Thus, we observed that a typical map-phase-only algorithm using distributed cache is suitable for those applications where a relatively smaller file has to be stored in distributed cache. Although more experiments are required to predict exactly the performance impact of file sizes in distributed cache, we observed noticeable performance degradation in case of larger files stored in the cache.

Figure 8 and figure 9 shows the breakdown of average execution times for map, shuffle and reduce phases of Chained MapReduce algorithm for the two data sets. Figure 10 and figure 11 shows the breakdown of average execution time for map, shuffle and reduce phases of distributed R-tree version for the two data sets. As we can see from the above mentioned figures, the reduce phase is the most time consuming where the bulk of overlay computation actually takes place. Map phase

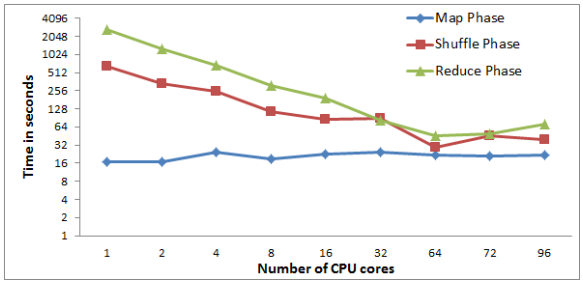

Fig. 8: Average execution time for different phases in Chained MapReduce Overlay Algorithm (Skewed data)

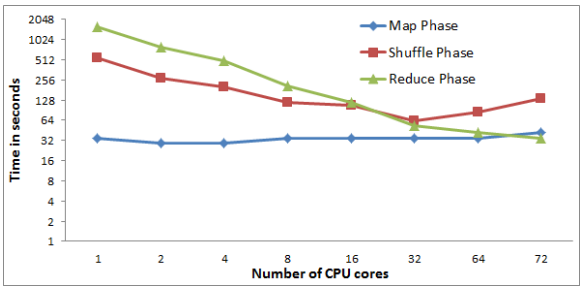

Fig. 9: Average execution time for different phases in Chained MapReduce Overlay Algorithm (Uniform data)

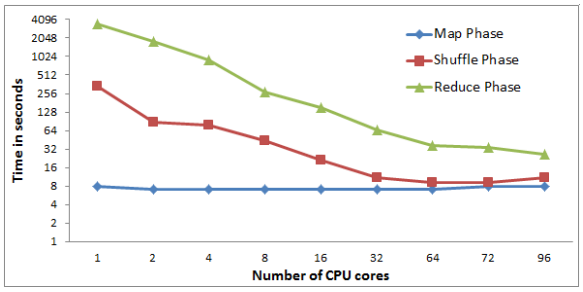

Fig. 10: Average execution time for different phases in Distributed R-tree based Overlay Algorithm (Skewed data)

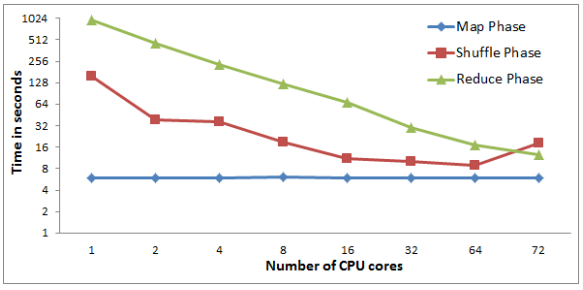

Fig. 11: Average execution time for different phases in Distributed R-tree based Overlay Algorithm (Uniform data)

consumes the least time of all the three phases. Map phase for Chained MapReduce algorithm performs local overlay computation and as such takes slightly more time than the map phase of Distributed R-tree algorithm where there is no overlay computation in map phase.

V. Conclusion

We have experimented with three MapReduce algorithms (excluding naive algorithm) using two data sets and the performance of these algorithms depend on the size and nature of the data. Even though, we have discussed polygon overlay in particular, some of the techniques discussed in this paper (related to use of distributed cache and In-Mapper combining design pattern) are applicable for parallelizing similar binary applications using MapReduce framework.

References

[1] D. Agarwal, S. Puri, X. He, and S. K. Prasad, "A system for GIS polygonal overlay computation on linux cluster - an experience and performance report," in *IEEE International Parallel and Distributed Processing Symposium workshops*, 2012.

[2] A. Guttman, *R-trees: A dynamic index structure for spatial searching*. ACM, 1984, vol. 14, no. 2.

[3] GPC clipper library. [Online]. Available: http://www.cs.man.ac.uk/~toby/alan/software/gpc.html

[4] W. Franklin, C. Narayanaswami, M. Kankanhalli, D. Sun, M. Zhou, and P. Wu, "Uniform grids: A technique for intersection detection on serial and parallel machines," in *Proceedings of Auto-Carto*, vol. 9, 1989, pp. 100–109.

[5] R. G. Healey, M. J. Minetar, and S. Dowers, Eds., *Parallel Processing Algorithms for GIS*. Bristol, PA, USA: Taylor & Francis, Inc., 1997.

[6] F. Wang, "A parallel intersection algorithm for vector polygon overlay," *Computer Graphics and Applications, IEEE*, vol. 13, no. 2, pp. 74 –81, mar 1993.

[7] H. Langendoen, "Parallelizing the polygon overlay problem using Orca," *Student Project Report, Vrije Universiteit Amsterdam*, 1995.

[8] T. Waugh and S. Hopkins, "An algorithm for polygon overlay using cooperative parallel processing," *International Journal of Geographical Information Science*, vol. 6, no. 6, pp. 457–467, 1992.

[9] S. Hopkins and R. Healey, "A parallel implementation of Franklins uniform grid technique for line intersection detection on a large transputer array," *Brassel and Kishimoto [BK90]*, pp. 95–104, 1990.

[10] Q. Zhou, E. Zhong, and Y. Huang, "A parallel line segment intersection strategy based on uniform grids," *Geo-Spatial Information Science*, vol. 12, no. 4, pp. 257–264, 2009.

[11] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu, "SJMR: Parallelizing spatial join with MapReduce on clusters," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–8.

[12] J. Zhao, Q. Li, and H. Zhou, "A cloud-based system for spatial analysis service," in *Remote Sensing, Environment and Transportation Engineering (RSETE), 2011 International Conference on*. IEEE, 2011, pp. 1–4.

[13] Y. Liu, N. Jing, L. Chen, and H. Chen, "Parallel bulk-loading of spatial data with MapReduce: An R-tree case," *Wuhan University Journal of Natural Sciences*, vol. 16, no. 6, pp. 513–519, 2011.

[14] A. Cary, Z. Sun, V. Hristidis, and N. Rishe, "Experiences on processing spatial data with MapReduce," in *Scientific and Statistical Database Management*. Springer, 2009, pp. 302–319.

[15] D. Agarwal and S. K. Prasad, "Lessons learned from the development of GIS overlay processing application on Azure Cloud platform," in *IEEE Cloud 2012 5th International Conference on Cloud Computing*, 2012.

[16] D. Agarwal, S. Puri, X. He, and S. K. Prasad. Crayons - an azure cloud based parallel system for GIS overlay operations. [Online]. Available: www.cs.gsu.edu/dimos/crayons.html

[17] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in MapReduce," in *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*. ACM, 2010, pp. 78–85.