Experiences on Processing Spatial Data with MapReduce*

Ariel Cary, Zhengguo Sun, Vagelis Hristidis, Naphtali Rishe

Florida International University
School of Computing and Information Sciences
11200 SW 8th St, Miami, FL 33199
{acary001,sunz,vagelis,rishen}@cis.fiu.edu

Abstract. The amount of information in spatial databases is growing as more data is made available. Spatial databases mainly store two types of data: raster data (satellite/aerial digital images), and vector data (points, lines, polygons). The complexity and nature of spatial databases makes them ideal for applying parallel processing. MapReduce is an emerging massively parallel computing model, proposed by Google. In this work, we present our experiences in applying the MapReduce model to solve two important spatial problems: (a) bulk-construction of R-Trees and (b) aerial image quality computation, which involve vector and raster data, respectively. We present our results on the scalability of MapReduce, and the effect of parallelism on the quality of the results. Our algorithms were executed on a Google&IBM cluster, which became available to us through an NSF-supported program. The cluster supports the Hadoop framework – an open source implementation of MapReduce. Our results confirm the excellent scalability of the MapReduce framework in processing parallelizable problems.

1 Introduction

Geographic Information Systems (GIS) deal with complex and large amounts of spatial data of mainly two categories: raster data (satellite/aerial digital images), and vector data (points, lines, polygons). This type of data is periodically generated via specialized sensors, satellites or aircraft-mounted cameras (sampling geographical regions into digital images), or GPS devices (generating geo-location information). GIS systems have to efficiently manage repositories of spatial data for various purposes, such as spatial searches, and imagery processing. Due to the large size of spatial repositories and the complexity of the applications to process them, traditional sequential computing models may take excessive time to complete. Emerging parallel computing models, such as MapReduce, provide a potential for scaling data processing in spatial applications.

* This research was supported in part by NSF grants IIS-0837716, CNS-0821345, HRD-0833093, EIA-0220562, IIS-0811922, IIP-0829576 and IIS-0534530, and equipment support by Google and IBM.

The goal of this paper is to present to the research community our experiences from using the MapReduce model to tackle two typical and representative spatial data processing problems. The first problem involves vector spatial data and the second involves raster data.

The first problem is the bulk-construction of R-Trees [1], a popular indexing mechanism for spatial search query processing. We show how previous ideas, like the ordering of multi-dimensional objects via space-filling curves, can be used to create a MapReduce algorithm for this problem. We also discuss how our solution is different from previous approaches on parallelizing the construction of an R-Tree.

The second problem processes aerial digital imagery, and computes and stores image quality characteristics as metadata. Original images may contain inaccurate, distorted, or incomplete data introduced at some phase of imagery generation; for example, a portion of an image may be completely blank. Pre-computed metadata is important in dynamic imagery consistency checking, and allows the appropriate mosaicing with better sources to improve the imagery display. This problem is naturally parallelizable since each tile can be potentially processed independently. In practice, the amount of data processed by each cluster processor depends on the file system characteristics like the minimum processing unit.

Both problems were solved and evaluated on a Google&IBM cluster supplied by the NSF Cluster Exploratory (CluE) program [2][3]. We present our experiences on using such a cluster in practice and deploying MapReduce jobs.

The key contribution of this work is as follow:

- We present techniques for bulk building R-trees using the MapReduce framework.
- We present how MapReduce can be applied to massively parallel processing of raster data.
- We experimentally evaluated our algorithms in terms of execution time, scalability and quality of the output. We provide various metrics to measure the quality of the resulting R-Tree.

This paper is organized as follows. Section 2 describes the steps in deploying MapReduce applications on the Google&IBM's cluster, as well as some physical configurations. Sections 3 and 4 present the detailed MapReduce algorithms for our two target problems. Section 5 presents experimental results of our algorithm implementations for different settings. Section 6 discusses related works. Last, Section 7 concludes our work.

2 Using MapReduce in Practice

The cluster used in this paper is provided by the Google and IBM Academic Cluster Computing Initiative [2][3]. The cluster contains around 480 computers (nodes) running open source software including the Linux operating system, XEN hypervisor and Apache's Hadoop [4], which is an open source implementation of the MapReduce programming model. Each node has half terabytes storage capacity summing up to about 240 Terabytes in total. Access to the cluster is provided through the Internet by a SOCKS proxy server. SOCKS is an Internet protocol that secures client-server communications over a non-secure network.

There are three main steps in interacting with the cluster, as shown in Figure 1. (1) Input data is uploaded into the cluster. The user uses file system shell scripts provided by the Hadoop Distributed File System (HDFS), which is an integral part of the Apache Hadoop project; HDFS is a clone project of Google's files system GFS [5]. (2) A user develops a Hadoop application and submits it to the cluster via Hadoop command. Hadoop applications are usually developed in Java, but other languages are supported, like C++ and Python. (3) After application execution is completed, the output is downloaded to the user's local site with Hadoop file system shell scripts.

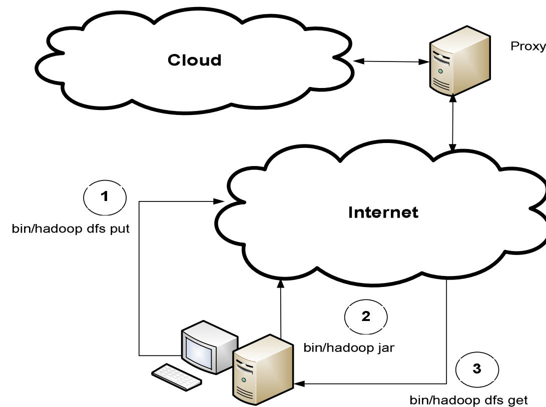


Fig. 1. Google, IBM Academic Cluster Overview

MapReduce programming model requires expressing the solutions with two functions: map and reduce. A map function takes a key/value pair, executes some computation, and emits a set of intermediate key/value pairs as output. A reduce function merges all intermediate values associated with the same intermediate key, executes some computation on them, and emits the final output. More complex interactions can be achieved by pipelining several MapReduce compounds in a workflow fashion. A data set is stored as a set of files in HDFS, which are in turn stored as a sequence of blocks (typically of 64MB in size) that are replicated on multiple nodes to provide fault-tolerance. An interested reader may refer to MapReduce Google's work [6] and open source Hadoop documentation [4] for a detailed description of MapReduce and Hadoop concepts.

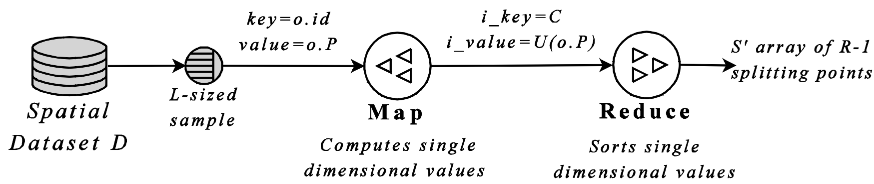
3 Building R-Tree with MapReduce

This section discusses a MapReduce-based algorithm for building an R-Tree index structure [1] on a spatial data set in parallel fashion. Let us start our description by defining the problem. Let D be a spatial data set composed of objects o_i, $i=1, .., |D|$. Each object o has two attributes $<o.id, o.P>$, where $o.id$ is the object's unique identifier and $o.P$ is the object's location in some spatial domain; other attributes are possible, but we concentrate on these only for our R-Tree construction purpose. The

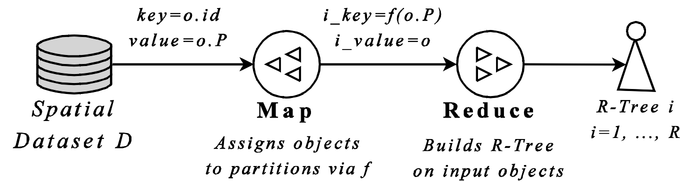
R-Tree minimum bounding rectangles (*MBRs*) are created based on the objects' spatial attribute *o.P*. Identifiers *o.id* are used as references to objects stored in the R-Tree leaves.

The proposed method consists of three phases executed in sequence, as can be seen in Figure 2. First, the spatial objects are partitioned into groups. Then, each group is processed to create a small R-Tree. Finally, the small R-Trees are merged into the final R-Tree. The first two phases are executed in MapReduce, while the last phase does not require high computational power, thus it is executed sequentially outside of the cluster.

Phase 1: *Partitioning Function Computation*



Phase 2: *R-Tree Construction*



Phase 3: *R-Tree Consolidation*

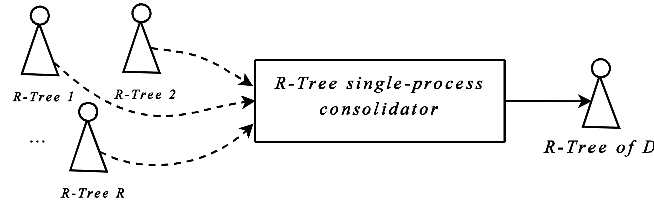


Fig. 2. Phases involved in building an R-Tree index for a data set *D* in MapReduce.

The three main phases of the algorithm are:

1 Computation of *partitioning* function *f*. The inputs for this phase are the data set *D* and a positive number *R,* which represents the number of partitions. The purpose of *f* is to assign any object of *D* into one of the *R* possible partitions. The function is computed in such a way that applying *f* on *D* yields *R* (ideally) equally-sized partitions. In practice, minimal variance in sizes is acceptable. At the same time, *f* attempts to put objects that are close in the spatial domain in the same partition. The output of this phase is a function *f* which takes as input an object location *o.P* and outputs a partition number. Note that no actual partitioning or data moving

happens at this point. The next phase utilizes f for such purpose. More details of this step are presented in Section 3.1.

2 *R-Tree construction*. During this phase, the function f calculated in the first phase is used by *Mappers* to divide D into R partitions. Then, R *Reducers* build R independent “small” R-Tree indices simultaneously on their input partitions. The output of this phase is a set of R independent R-Trees. Details of this step are presented in Section 3.2.

3 R-Tree *consolidation*. This phase combines the R individual R-Trees, built in the second phase, under a single root node to form the final R-Tree index of D. This phase can be as simple as making the R R-Trees children of a single root node, or it may require adding a few extra levels (at most one in practice) if R exceeds the capacity of a single node. Since this phase is not computationally intensive for R under a few hundreds or thousands, it is executed by a single process outside the cluster. The logic to run this phase is fairly simple, so no further elaboration will be done on this step.

3.1 Partitioning Function

The purpose of the partitioning function f is to provide a means for assigning objects of D to a pre-defined number of R partitions. We use the idea of mapping multi-dimensional spaces into an ordered sequence of single-dimensional values via space-filling curves for this purpose. This idea has been studied in the literature as a way to numbering objects in multi-dimensional spaces [7, 8]. In our present problem, we map objects’ location attribute $o.P$ into such curves. We use the Z-order curve [9] in our experiments in Section 5.1. The partition number of an object o is determined by $f(o.P)$, which evaluates to a value from the set $\{1, 2, .., R\}$. By using a space-filling curve, the partitioning function f achieves two goals:

- Generate R (almost) uniformly-sized partitions, and
- Preserve spatial locality. If two distinct objects $o1$ and $o2$ are close to each other in the spatial domain, then they are likely to be assigned to the same partition, i.e. $f(o1.P) = f(o2.P)$.

Next, we propose a MapReduce algorithm to define f.

MapReduce Algorithm

The general idea is inspired by the TeraSort Hadoop application [10], which partitions an input da*ta set* via d*ata* sampling. Given a data set D and target number of partitions R, the MapReduce algorithm runs M Mappers that collectively take L sample objects from D (that is, each Mapper samples $\frac{L}{M}$ objects) and emit their single-dimensional values $S=\{U(o_i.P), i=1, .., L\}$ given a space filling curve U. Then, a single Reducer sorts S, and determines a list S' of $R-1$ splitting points that split the ordered sequence of samples into R equal-sized partitions. Then, in general, an object o belongs to partition j if $S'[j-1] < U(o.P) \leq S'[j]$. Thus, f utilizes the splitting points in S' to assign objects to partitions.

The specific MapReduce key/value input pairs as well as outputs are presented in Table 1. Mappers read in total L samples at random offsets of their input D, and compute their single dimensional value with the space-filling curve U. The intermmediate key equals to C which is a constant, whose value is irrelevant, that helps in sending Mappers' outputs to a single Reducer. The Reducer receives the L single-dimensional values generated by Mappers, and sorts them into an auxiliary list $u_1, .., u_L$, from which R-1 elements are taken starting at the $(\frac{L}{R})$-*th* element and subsequently at fixed-length offsets $\frac{L}{R}$ to form a list S' of splitting points.

Table 1. Map and Reduce inputs/outputs in computing partitioning function *f*.

Function	Input: (Key, Value)	Output: (Key, Value)
Map	$(o.id, o.P)$	$(C, U(o.P))$
Reduce	$(C, list(u_i, i=1, .., L))$	S'

An important observation in the sampling process is that Mappers read input data from the distributed storage at block-sized amounts, which is a Hadoop distributed file system parameter specifically tuned for load balancing large files across storage nodes. Thus, all Mappers, except perhaps for the last one, will read the same amount of data, equal to the file system's block size.

The rationale of the splitting points in S' is that they provide good enough *boundaries* to sub-divide D into R partitions since they come from randomly sampled objects. Experiments in section 5.1 show very low standard deviation (under 1%) on the number of objects per partition. Formally, the function *f* is defined as follows:

$$f(o.P) = \begin{cases} 1, & U(o.P) \leq S'[1] \\ j, & S'[j-1] < U(o.P) \leq S'[j], j = 2, \dots, R-1 \\ R, & U(o.P) > S'[R-1] \end{cases} \quad (1)$$

This computation is characterized by running multiple *Mappers* (samplig data) and one *Reducer* (sorting samples), which may become a limiting factor in scalability. If the size of S becomes sufficiently large, then the TeraSort [10] approach can be used to sort its items in parallel, which makes the algorithm more scalable.

3.2 R-Tree Construction

In this phase, R individual R-Tree indices are built concurrently. Mappers partition the input data set D into R groups using the partitioning function *f*. Then, every partition is passed to a different Reducer, which independently builds an R-Tree on its input. Next, every Reducer outputs a root node of their constructed R-Trees, so R sub-trees are written to the file system at the end of this phase.

Input and output key/value pairs are shown in Table 2. Mappers read their input data in its entirety and compute objects assigned partitions via $f(o.P)$. Then, every Reducer receives a number of input objects A for which an R-Tree is built and its root emitted as output. Since f balances partitions, it is expected that all Reducers will receive a similar number of objects ($A \sim \frac{|D|}{R}$), thus executing similar amount of work

in constructing their R-Trees. However, good balancing depends on the underlying space-filling curve *U* used by *f*, and the number of sampled objects *L*. More samples help in tuning the splitting points, but incur in larger sorting time of *L* elements.

Table 2. MapReduce functions in constructing R-Trees.

Function	Input: (Key, Value)	Output: (Key, Value)
Map	(*o.id*, *o.P*)	(*f*(*o.P*), *o*)
Reduce	(*f*(*o.P*), *list*($o_{i,\ i=1,\ \ldots\ A}$))	*tree.root*

Another concern is the quality of the produced R-Trees in relation to the parameter *R*. In Section 5.1, we provide some initial insight into this direction by measuring R-Tree parameters such as area and overlap in a simplified way, and plotting their *MBRs* for visual analysis.

4 Tile Quality Computation Using MapReduce

This section discusses a MapReduce algorithm to compute the quality information of aerial/satellite imagery. Such information is useful for fast identification of defective image portions, e.g. blank regions inside a tile or a group of tiles, and subsequent dynamic image patching using better imagery available at rendering time. For a given tile, we define a pixel as "bad" if all the values of its samples are below or above some predefined value.

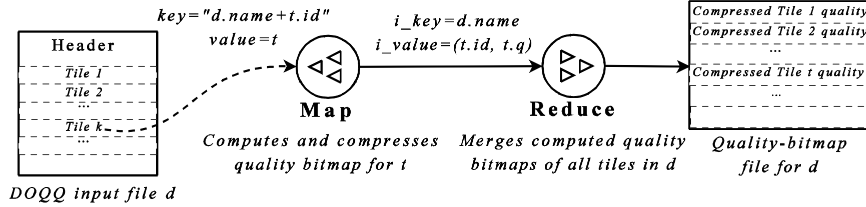


Fig. 3. Tile quality computation algorithm overview.

Image tiles are stored in customized DOQQ files [11], augmented with a descriptive header. Let *d* be a DOQQ file and *t* be a tile inside *d*, *d.name* is *d*'s file name and *t.q* is the quality information of tile *t*. More details of our data set are presented in Section 5.2. Figure 3 depicts the execution overview of our MapReduce algorithm. The algorithm runs on a tile by tile basis within the boundaries of a given DOQQ file, computing a bitmap per tile where a tile pixel is associated to a bit that is set to 1 if the pixel is deemed "bad", and 0 otherwise.

MapReduce Algorithm

Each DOQQ file is first partitioned into several splits, each of which is then processed by a separate Mapper. Splits are carefully generated by parsing tiles out of the input file until the size of all the tiles is close (little smaller) to the block size of the

underlying distributed file system or end of file is reached. In doing so, tile boundaries are preserved between different splits. Then, each Mapper will have to read at most two blocks of a file. This helps reduce data transfer time between nodes because different blocks of a file are usually stored on separate nodes. Tiles (values) inside one split are identified by *d.name* and *t.id* (keys) and combined as key/value input for Mappers.

Table 3. Input and output of map and reduce functions

Function	Input: (Key, Value)	Output: (Key, Value)
Map	(*d.name+t.id, t*)	(*d.name*, *t.q*)
Reduce	(*d.name*, *list*(t_i.*q*))	*Quality-bitmap of d*

The input and output key/value pairs for Mappers and Reducers are described in Table 3. The Mapper decompresses the JPEG tile *t*, iterates through each pixel of *t* to obtain quality information *t.q* (a bitmap, one bit per pixel) and compresses it using Run-length encoding (RLE) algorithm. After that, it emits the intermediate key/value pair with *d.name* as the key and *t.q* as the value. The Reducer merges all the *t.q* bitmaps that belongs to a file *d* and writes them to an output file, containing image quality for *d*, as shown in Figure 3.

5 Experiments

This section presents and discusses the experimental results we obtained by running the algorithms described in Sections 3 and 4 as Hadoop applications on the Google&IBM cluster presented in Section 2. All the data sets used in this section are real spatial data sets supplied by the *High Performance Database Research Center* at Florida International University [12]. At the time of experimentation, there were jobs running in the cluster from other researchers that share this resource, thus some fluctuation in the results is expected.

5.1 R-Tree Construction

Data sets and Setup

Experiments are executed on two real spatial data sets. Data set descriptions are shown in Table 4. The points in the data sets are angular coordinates in (*latitude*, *longitude*) format. In the following experiments, we use the *Z-order* space-filling curve [9] as *U* function to map the two-dimensional points into a single dimension. We used 3% of each data set as sampling size *L* (see first phase of the algorithm in Section 3). Data sets are in tabular structured format (CSV), where each line represents an object. We used Hadoop supplied functions to read objects (text lines) from the data sets. During the second phase, Reducers build their individual R-Trees in-memory (to avoid high disk latencies in maintaining the tree along object insertions), then the trees are peristed on Hadoop distributed file system.

Time Performance

This experiment consists of building R-Tree indices on the Google&IBM cluster changing the parameter *R* in phase-2, that is, the number of concurrently-built R-Trees, from 2 up to 64. As *R* varies, job completion times are measured for Mappers and Reducers as well as quality statistics on the resulting R-Trees. As a reference, we also ran a single-process R-Tree construction on a dedicated local machine with Intel Xeon E7340 2.4GHz processor and 8GB of RAM running Windows OS; we could not run the single process in the cluster since we do not have login access to individual nodes. Thus, cluster and single process times are not comparable due to dissimilar hardware.

Table 4. Spatial data sets used in experiments.

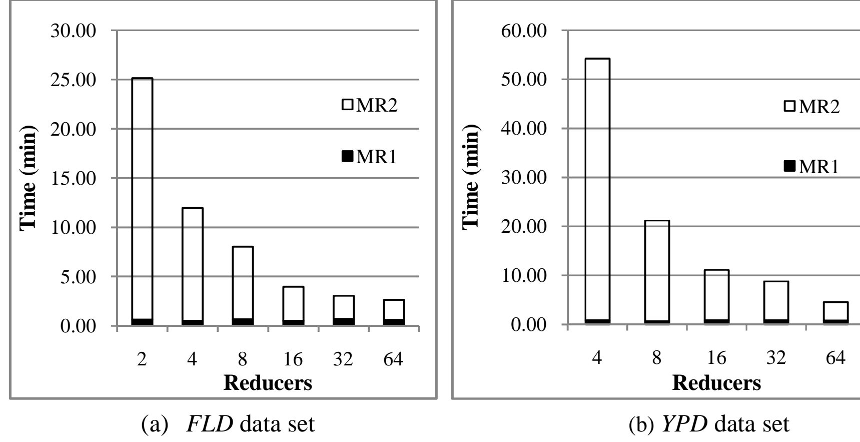
Data set	Objects (millions)	Data size (GB)	Description
FLD	*11.4*	*5*	Points of properties in the state of Florida.
YPD	*37*	*5.3*	Yellow pages directory of points of businesses mostly in the United States but also in other countries.

Table 5. MapReduce job completion times (in minutes) for the Phase 1 (*MR1*), and various Reducers (R) in Phase 2 (*MR2*) of building an R-Tree. Also, completion times for single-process (SP) constructions ran on a local machine are shown.

		MR1: ***Partitioning Function***		**MR2:** ***R-Tree Construction***		
Data set	**R**	**Map**	**Reduce**	**Map**	**Reduce**	**Total MR1+MR2**
FLD	2	0.35	0.28	0.40	24.12	25.15
	4	0.28	0.23	0.40	11.07	11.98
	8	0.47	0.22	1.73	5.62	8.03
	16	0.30	0.22	0.40	3.05	3.97
	32	0.48	0.23	0.40	1.95	3.07
	64	0.28	0.33	0.45	1.60	2.67
	SP	-	-	-	-	27.34
YPD	4	0.47	0.38	0.47	52.57	53.88
	8	0.22	0.45	0.72	25.42	26.80
	16	0.40	0.43	0.38	8.93	10.15
	32	0.40	0.43	0.42	4.65	5.90
	64	0.40	0.42	0.88	2.55	4.25
	SP	-	-	-	-	63.98

Table 5 shows MapReduce job completion times for R-Tree construction phases 1 and 2 on both spatial data sets as well as for a single-process build (SP); for *YPD* we start at *R*=4 due to memory limitations in cluster nodes for building in-memory trees with less number of Reducers. We do not include phase-3 processing times since it is of little significance compared to the other phases. Phase-1 (partitioning function computation) takes very little time, which is expected since sorting *L*=3% of objects from a data set can be quickly done in memory by the single reducer in this phase; for our largest data set *YPD*, about 1 million elements are sampled. Our *Z-order* values

are 8-byte sized elements, so around 8MB of RAM is needed to execute the sort, which is much less than the memory of each cluster node. Likewise, Mappers in phase-2 read data sequentially and execute inexpensive Z-order value computations on their inputs.



(a) *FLD* data set (b) *YPD* data set

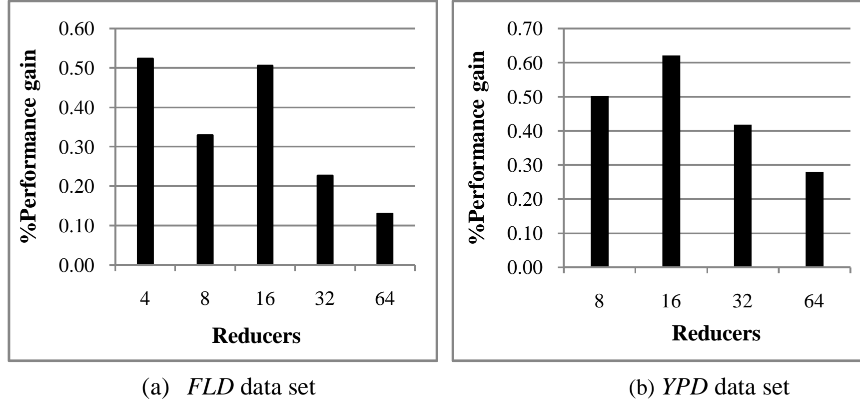
Fig. 4. MapReduce job completion times for various number of reducers in phase-2 (MR2).

The most computationally intensive part is performed by Reducers in phase-2, where the actual R-Tree constrution occurs. The fewer the number of Reducers, the longer the R-Tree construction takes, since each task receives larger number of objects. Figure 4 shows job completion times as stacked bars of the map and reduce execution times. In this figure, almost linear scalability is observed as more parallelism is induced by increasing *R* in phase-2. As expected, the improvement rate is high for few Reducers but drops as the number of Reducers increases since partitioning overheads in phase-1 (*MR1*) start becoming significant compared to R-Tree build time in phase-2 (*MR2*). In fact, for larger values of *R*, the dominating time component is given by *MR1* which, as can be seen in Table 5, is almost constant for a given data set. Thus, much less improvements are expected as *R* is increased beyond 64.

Although we cannot compare our MapReduce and single process (SP) times due to mismatch in hardware, the MapReduce parallelization certainly yields performance benefits for large-scale data sets. For example, it takes more than an hour to sequentially build the *YPD* R-Tree, while in parallel the task can be achieved in less than 5 minutes with 64 Reducers. However, the resulting R-Trees are different due to differences in object insertion sequences. Later in this section we measure and discuss R-Tree quality parameters for both cases.

Figure 5 presents percentages of performance gains in job completion times in relation to subsequent increases in the number of Reducers in the second phase of the algorithm. For example, in the *YPD* dataset, going from 4 to 8 Reducers we observe 50% decrease in job completion time, which represents linear scalability. On the other hand, going from 8 to 16 Reducers shows super-linear gains (62%). We pressume this may be due to heterogeneous nodes in the cluster (eventually the job with R=16 was

executed on faster nodes), or it may be the cluster resources were idler during that period. As discussed, as we increase the number of Reducers, performance gains are less significant because the execution time for the first phase, which has a sequential component (*Reduce*), stays almost constant.



(a) *FLD* data set (b) *YPD* data set

Fig. 5. MapReduce job percentage of performance gains as the number of reducers is increased.

Quality of Generated R-Trees

We use equations (2) and (3) below to compute the area and overlap metrics respectively for a given consolidated R-Tree with root *T*:

$$Area(T) = \sum_{i=1}^{n} Area(T_i.MBR) \tag{2}$$

$$Overlap(T) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} Area(T_i.MBR \bigcap T_j.MBR) \tag{3}$$

where *n* is the number of children (small R-Trees generated by Reducers) of *T,* and T_i is the *i*-th child node of *T*. Note that other metrics of R-Tree quality could be considered as well, e.g., consider all the nodes of the R-Tree instead of just the top level. Minimal area and overlap are known to improve search performance [13] since they increase path pruning abilities of R-Tree navigation algorithms.

Table 6 shows quality metrics on the consolidated R-Trees built for various number of Reducers and single process (SP); for reference, the U.S. Census Bureau reports Florida state land area roughly as 54,000 square miles as of 2000 [14]. As expected, we see the total MBR area and the overlap increase as the parallelism (*R*) increases because the construction of each small R-Tree is unaware of the rest of the data set, lowering the chance of co-locating neighbor objects within the same R-tree. This means that we degrade the R-Tree quality without gaining in execution time. The

latter can adversely effect performance of search algorithms, such as nearest neighbor type of queries, due to extra I/Os incurred in traversing multiple sub-trees.

Table 6. Statistics on consolidated R-Trees built by various number of *Reducers* (*R*), and single process (*SP*) construction.

		Objects per Reducer		*Consolidated* R-Tree			
Data set	***R***	**Average**	**Stdev**	**Nodes**	**Height**	**Area (sq.mi)**	**Overlap (sq.mi)**
FLD	2	5,690,419	12,183	172,776	4	132,333.9	304.4
	4	2,845,210	6,347	172,624	4	106,230.4	4,307.9
	8	1,422,605	2,235	173,141	4	103,885.8	17,261.9
	16	711,379	2,533	162,518	4	96,443.1	21,586.3
	32	355,651	2,379	173,273	3	140,028.7	80,389.1
	64	177,826	1,816	173,445	3	152,664.2	96,857.7
	SP	11,382,185	0	172,681	4	746,145.0	1,344,836.8
YPD	4	9,257,188	22,137	568,854	4	26,510,946.3	21,574,857.8
	8	4,628,594	9,413	568,716	4	23,160,080.0	20,480,729.6
	16	2,314,297	7,634	568,232	4	67,260,270.0	54,582,299.8
	32	1,157,149	6,043	567,550	4	68,626,854.9	54,008,538.5
	64	578,574	2,982	566,199	4	69,791,363.8	55,064,139.4
	SP	37,034,126	0	587,353	5	164,966,688.5	658,583,322.6

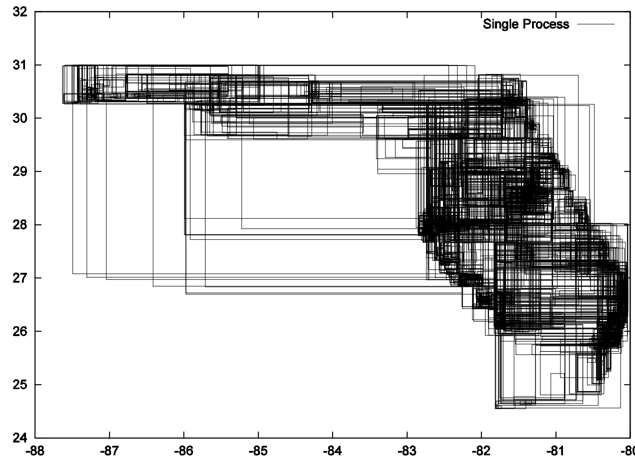


Fig. 6. MBR plotting for *FLD* data set on an R-Tree built by a single process.

For a sequential construction (*SP*), we observe these metrics are much worse, especially the overlap factor, since objects are not spatially shuffled but rather inserted in the data set original sequence. Thus, higher performance penalties are expected in *SP* constructed R-Trees. On the other hand, the tree height slightly decreases for *FLD* for *R* beyond 32 because more small trees means that each one of them may be shorter, while for *YPD* the height increases by one level for the *SP* case.

In general, small variations in tree height is less significant from a performance standpoint.

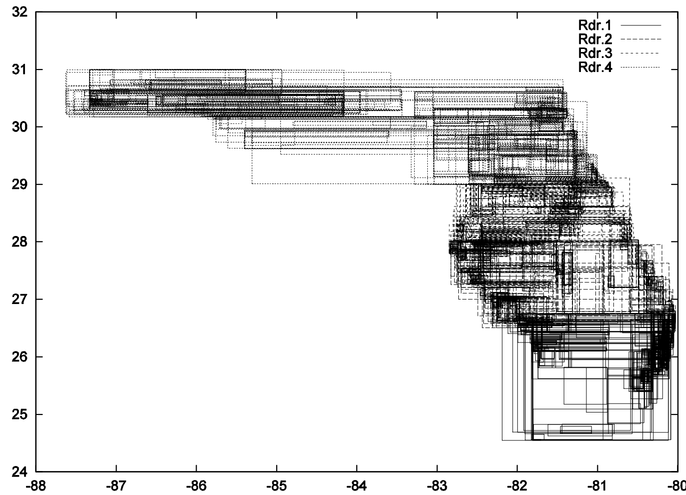


Fig. 7. MBR plotting for *FLD* data set for R-Tree built by MapReduce with *R*=4.

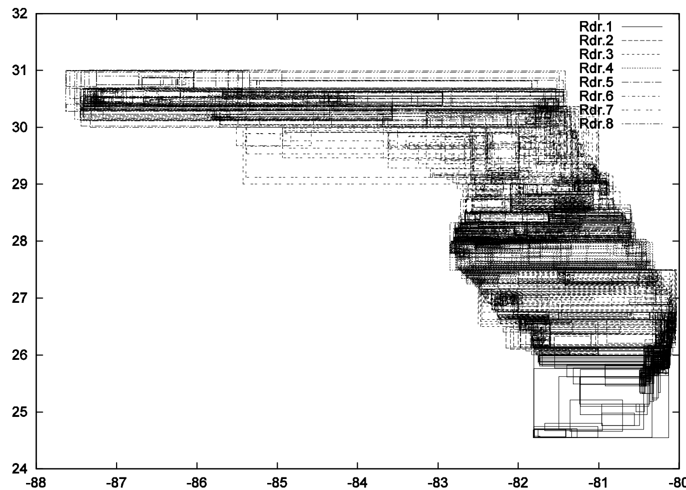


Fig. 8. MBR plotting for *FLD* data set for R-Tree built by MapReduce with *R*=8.

To visually study the effect of increasing *R* over the MBR distribution, we have plotted the MBRs of the resulting R-Trees for the case of 4 and 8 Reducers in Figures 7 and 8 respectively for the Florida state data set (*FLD*). Also the same type of graph is shown in Figure 6 for the *SP* R-Tree. In neither case is the root MBR plotted since it is common for all trees.

A few observations can be made from the MBR plottings. First, the partitioning mechanism employed in our algorithms seems to be effective in preserving spatial

locality. This results in individual Reducers indexing highly localized objects; their boundaries, however, result in multiple overlappings, which are inevitable. Second, as the number of *Reducers* is increased from 4 to 8, the plotting shape resembles more the actual shape of the Florida state; that is, *R*=8 reduces wasted areas (where no actual objects are located) as the Area statistic confirms in Table 6. In fact, Table 6 shows steady decrease in area from 2 to 16 *Reducers*; after that the area keeps on increasing. Third, when the R-Tree is built on the original sequence of objects (no object shuffling) in *SP* mode, large wasted areas are generated as can be observed in Figure 6. From a performance optimization perspective, MapReduce generated R-Trees seem to be better tuned than their single-process counterpart. Therefore, we see promising performance improvements in MapReduce generated R-Trees, which deserve closer verification.

5.2 Tile Quality

Data set and Setup

The data set used in the experiments is a 3-inch resolution aerial imagery of Miami Dade County of Florida. The size of the data set is about 52GB after compression. Imagery data is stored as compressed DOQQ file format. There are 482 compressed DOQQ files, each of which contains 4096 tiles. Each tile is 400 by 400 pixels and has 3 bytes for each pixel as the Red, Green and Blue channel. The size for each tile is 480,000 bytes uncompressed and compressed tile is about 50 KB each.

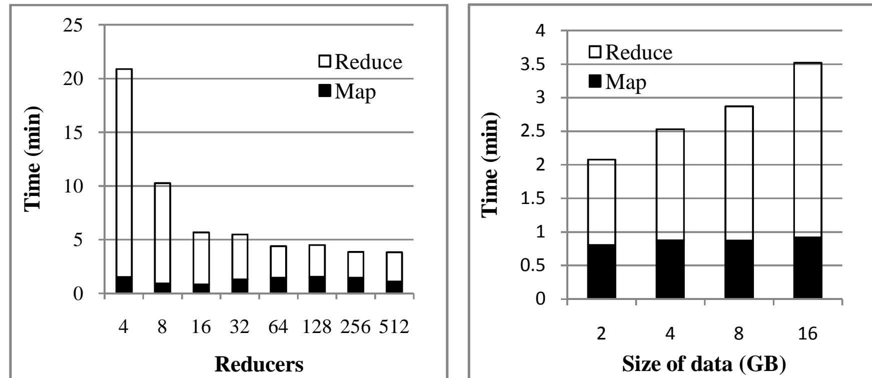
Experiments

Two experiments are carried out for this data set. The first experiment uses a subset of the data set that is a re-sampled version of the original one. It is about 20GB and has 482 files with 1024 tiles each. The size of the files ranges from several megabytes to around 80 megabytes, and the number of Reducers is varied from 4 to 512. The second experiment uses different sized subsets of the original data set. The size of the files ranges from 2GB to 16GB, and the number of Reducers is fixed at 256.

In the first experiment, the number of Mappers is also fixed, determined by the data set size. Thus, the execution time of the map phase is similar through different runs, as can be seen in Figure 9 (a). The execution time slightly fluctuates because there were other concurrent jobs running in the cluster at the same time. As the number of Reducers increases, the execution time of the reduce phase largely decreases for smaller number of reducers, and less improvements are obtained for larger number of reducers. This is because the same amount of work is now shared by more Reducers. When the number of Reducers is larger than 64, the execution time of the reduce phase stabilizes to around 2.5 minutes. This could be explained by the launching time of Reducers dominating the whole time at this point. With 64 Reducers, each of them will be writing around $482/64 \approx 8$ files. The time taken to write 8, 4 (128 Reducers) or even less files is negligible compared with the launching time of that many Reducers.

In the second experiment, Figure 9 (b), as the size of the data set increases with constant number of reducers (256), the execution time of the map phase hardly changes, which is consistent with the data parallelization provided by the MapReduce

model, that is, more Mappers are engaged in processing the data. The execution time of the reduce phase increases because there are now more files to be written with the same number of Reducers.



(a) Fixed data size, variable Reducers (b) Variable data size, fixed Reducers

Fig. 9. MapReduce job completion time for tile quality computation

6 Related Work

Space-filling Curves

The idea of using space-filling curves to map multi-dimensional spaces into a single dimension has been studied for the case of spatial databases [15, 8]; popular space-filling curves, such as Peano and Hilbert, have been studied in great level of detail. We used the Z-order curve in our experiments. This curve showed high spatial locality preservation for our experimented real data sets. Other curves can certainly be evaluated, which goes beyond our focus on the parallelization of two concrete spatial problems with MapReduce.

Parallel R-Tree Constructions

Previous works on R-Tree parallel construction have faced several intrinsic distributed computing problems such as data load balancing, process scheduling, fault tolerance, etc., for which they elaborated special-purpose algorithms. Schnitzer and Leutenegger [16] propose a Master-Client R-Tree, where the data set is first partitioned using Hilbert packing sort algorithm, then the partitions are declustered into a number of processors (via an specialized declustering algorithm), where individual trees are built. At the end, a master process combines the individual trees into the final R-Tree. Another work by Papadopoulos and Manolopoulos [17] proposed a methodology for sampling-based space partitionining, load balancing, and partition assignment into a set of processors in parallely building R-Trees. They also discuss some alternatives when the global (consolidated) index has imperfections such as different heights across individual R-Trees.

In MapReduce, these parallel computing concerns are abstracted out from the application logic, and managed transparently as part of the MapReduce framework. Further, all nodes in the cluster access a common distributed file system, with automatic fault-tolerance and load balancing support, where data locality is employed as base criterion to assign Mappers and Reducers (preferably) to nodes already containing the input data. In contrast, traditional parallel processing works assume every node has its own storage, in a shared-nothing type of architecture, where data transfer among nodes becomes an important optimization goal.

MapReduce on Spatial Data

MapReduce framework was used to solve another spatial data problem by Google [18], where they study the problem of road alignment by combining satellite and vector data. This work concentrates on the complexities of the problem, which are more challenging than the MapReduce algorithms.

Schlosser et al. [19] worked on building octrees in Hadoop for later use in earthquake simulations at large-scale. Their approach builds a tree in a bottom up fashion. The map function in the first iteration generates leaf nodes, then the reduce function coalesces homogeneous leaf nodes into a subtree. Subsequent iterations have identity functions in mappers, and successively use reduce functions to construct the final tree.

Relationship to MPI

Message Passing Interface (MPI) [20] is a specification of a language-independent communication model targeted at writing parallel programs, and it is widely used in a variety of computer cluster platforms. MPI libraries provide primitives and functionality for communication control among a set of processes. Typically, developers need to add explicit calls to synchronize processes and move data around. The key differences between MPI and MapReduce is that MapReduce exploits its simplified model to automatically parallelize tasks (Mappers and Reducers), hiding from programmers the need to worry about process communication, fault-tolerance, and scalability, which are transparently managed by key components, such as cluster management system and distributed file system, that the MapReduce framework is built-upon [6]. For example, for the R-Tree case study, the Java implementation of the Map and Reduce functions of the first phase, and Map of the second phase have each less than 40 lines of code. The Reduce function in the second phase has about 70 lines of code since it includes extra code for persisting the tree on the distributed file system and collecting build statistics. These numbers do not include application-specific routines, which are needed regardless of the parallel model.

In MapReduce, the underlying assumption is that the solution can be expressed in terms of the Map and Reduce functions working on key/value pairs. In some cases this may not be natural, such as relational joins or multi-stage processes, and can lead to inefficiencies. Then, MPI-like parallel implementations have more opportunities to address application-specific optimizations, due to its finer process control. However, high-level languages have been proposed to address this problem in MapReduce architectures by providing efficient primitives for massive data analysis combining SQL-like declarative style and MapReduce procedural programming [21][22].

7 Conclusions

In this paper, we used the MapReduce programming model to solve two important spatial problems on a Google&IBM cluster: (a) bulk-construction of R-Trees and (b) aerial image quality computation, which involve vector and raster data, respectively. The experimental results we obtained indicate that the appropriate application of MapReduce could dramatically improve task completion times. Our experiments show close to linear scalability. However, performance is not the only concern for R-Tree construction, which is sensitive to the ordering of objects in its input, but also the quality of the result. MapReduce generated R-Trees have improved quality in terms of MBR area and overlap measurements compared to the single-process construction counterpart. No such quality problem arises in the aerial image quality computation. Our experience in this work shows MapReduce has the potential to be applicable to more complex spatial problems.

References

[1] Antonin Guttman: R-Trees: A Dynamic Index Structure for Spatial Searching. SIGMOD 1984:47-57.
[2] NSF Cluster Exploratory Program: http://www.nsf.gov/pubs/2008/nsf08560/nsf08560.htm
[3] Google&IBM Academic Cluster Computing Initiative: http://www.google.com/intl/en/press/pressrel/20071008_ibm_univ.html
[4] Apache Hadoop project: http://hadoop.apache.org
[5] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung: The Google file system, SIGOPS Operating Systems Review, Volume 37, Issue 5, pp. 29-43, 2003.
[6] Jeffrey Dean, Sanjay Ghemawat: MapReduce: Simplified data processing on large clusters. In Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation, USENIX Association, Volume 6, pp. 10-10, December 2004.
[7] Tetsuo Asanoa, Desh Ranjanb, Thomas Roosc, Emo Welzld and Peter Widmayer: Space-filling curves and their use in the design of geometric data structures, Theoretical Computer Science, Volume 181, Issue 1, pp. 3-15, July 1997.
[8] Jonathan K. Lawder, Peter J. H. King: Using Space-Filling Curves for Multi-dimensional Indexing, Book Advances in Databases, Springer Berlin, Volume 1832, pp. 20-35, 2000.
[9] Morton, G. M.: A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing, Technical Report, Ottawa, Canada: IBM Ltd., 1966.
[10] Owen O'Malley: TeraByte Sort on Apache Hadoop, Yahoo!, May 2008.
[11] Doqq file format: http://egsc.usgs.gov/isb/pubs/factsheets/fs05701.html
[12] High Performance Database Research Center (HPDRC), Research Division of the Florida International University, School of Computing and Information Sciences, University Park, Telephone: (305) 348-1706, FIU ECS-243, Miami, FL 33199.
[13] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, Bernhard Seeger: The R*-tree: an efficient and robust access method for points and rectangles, Volume 19, Issue 2, pp. 322-331, 1990.
[14] U.S. Census Bureau, Florida State and County QuickFacts, http://quickfacts.census.gov/qfd/states/12000.html last revised: 25-Jul-2008.
[15] David J. Abel, David M. Mark: A comparative analysis of some two-dimensional orderings, International Journal of Geographical Information Science, Volume 4, Issue 1, pp. 21 - 31, January 1990.

[16] Schnitzer B., Leutenegger S.T.: Master-client R-trees: a new parallel R-tree architecture, In Proceedings of the 11th International Conference on Scientific and Statistical Database Management, pp. 68-77, August 1999.
[17] Apostolos Papadopoulos, Yannis Manolopoulos: Parallel bulk-loading of spatial data, Parallel Computing, Volume 29, Issue 10, pp. 1419 - 1444, October 2003.
[18] Xiaqing Wu, Rodrigo Carceroni, Hui Fang, Steve Zelinka, Andrew Kirmse: Automatic alignment of large-scale aerial rasters to road-maps, Geographic Information Systems, Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems, Article No. 17, 2007.
[19] Schlosser S. W., Ryan M. P., Taborda R., Lopez J., O'Hallaron D. R., and Bielak J.: Materialized community ground models for large-scale earthquake simulation, In Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Conference on High Performance Networking and Computing, pp. 1-12, 2008.
[20] William Gropp, Ewing Lusk, and Anthony Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge, MA, 1999.
[21] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pp 1029-1040, 2007.
[22] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pp 1099-1110, 2008.