Parallel Quadtree Implementation and Testing on Spark Platform

Syakir Farhan[1]

[1]farhan0syakir@gmail.com, Institute of Computer Science, University of Tartu

In this paper, we use spark frame to implement efficient parallel quadtree construction and manipulation algorithms. The quadtree tree is introduced to represent the distributed quadtree structure on a spark. By incorporating the geometric property into the distributed quadtree structure, we are designing a data assignment scheme that meets the multi-level adjacency preserving property (MAP). We show that the distributed data structure in conjunction with geometric constraint not only results in an effective parallel quadtree construction, but also facilitates many graphics and image processing operations on constructed quadtree, such as neighborhood finding, and border extraction. Empirical testing is performed to show that the algorithms proposed are better than non-parallel algorithms in terms of time execution.

Keyword : Quadtree | Spatial | Geolocation

Introduction

In this paper, in computational geometry, I find a fundamental problem: data structures for nearest neighbor queries. Although there are a lot of research on the topic, our research is different in the sense that to enhance performance I focus on parallelize queries. In a standard approach, I are given a set of n points stored in an array in a standard approach, and I build a data structure for efficient queries on the array. The value of n is also massive for real data, for example, there are more than 50 million edges of road networks in Japan or the USA[1].

With the growing pervasiveness of geo-positioning technologies and geo-location services, in many applications there is a huge amount of spatio-textual data available. For example, the online business directory (e.g. yellow pages) offers location information and brief business details (e.g. hotels, restaurants) in the local search system. A POI (point of interest) is a geographically grounded pushpin in the GPS navigation network that somebody can find useful or interesting, usually annotated with texture data (e.g. details and feedback of users). In addition, in many social network sites (e.g. Twitter, Flickr), a large number of geo-tagged images are accumulated on a daily basis, which users can geo-tagged. As a consequence, numerous spatial keyword query models and techniques have emerged in recent years, enabling users to effectively leverage these spatial objects ' spatial and textual data.

In the paper, I examine the issue of conducting spatial search, i.e. a request position q using parallel frame spark given a set of spatial objects. We're trying to get the nearest k items in the query. Motivating examples are given below[2].

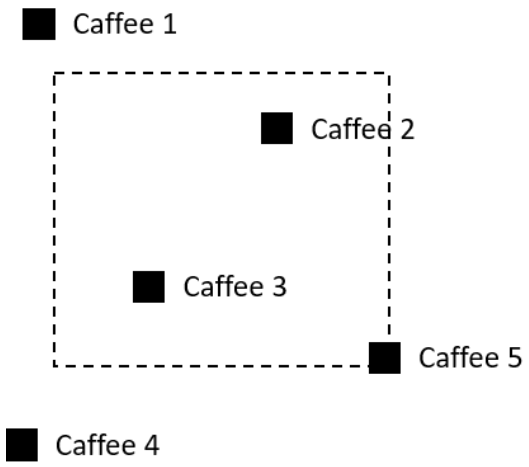

Fig. 1. Motivating Example

In the Figure. 1, Assume that there is a set of caffee (coffee shop) whose locations (represented by squares) and service lists (a set of keywords) are registered in the local search service provider's online yellow pages. When a GPS-enabled smartphone user wants to find a nearby caffee, she may send the local search. Based on the user's area location (e.g., the dashed square in Fig. 1) derived from the smartphone, caffee 2, 3 and 5 are returned by the server. Note that although caffee 1 and 4 are not in the area, they do not satisfy the constraint.

The workload of the request may vary from time to time in many real applications, and the process may experience a burst of queries (e.g. queries triggered by a specific event). In this case, if a large number of requests are handled one by one, the entire system is bad. Motivated by this, a large body of existing work has been committed to exploring how to improve the system with the parallel database processing techniques so that a large number of requests can be processed with a fair delay. Meanwhile, in a short period of time, a large volume of spatial keyword queries can be created today. For example, during dinner and lunch time, a large number of queries may be required to seek nearby restaurants[3].

With the rapid development of Big Data in recent years, more and more systems need to be applied to large clusters. The environment of programmable clusters has brought a number of challenges: Firstly, Most programs need to be rewritten in parallel, and more forms of data processing need to be handled by programmable clusters. Secondly, More important and difficult is the fault tolerance of the Clusters; Thirdly,

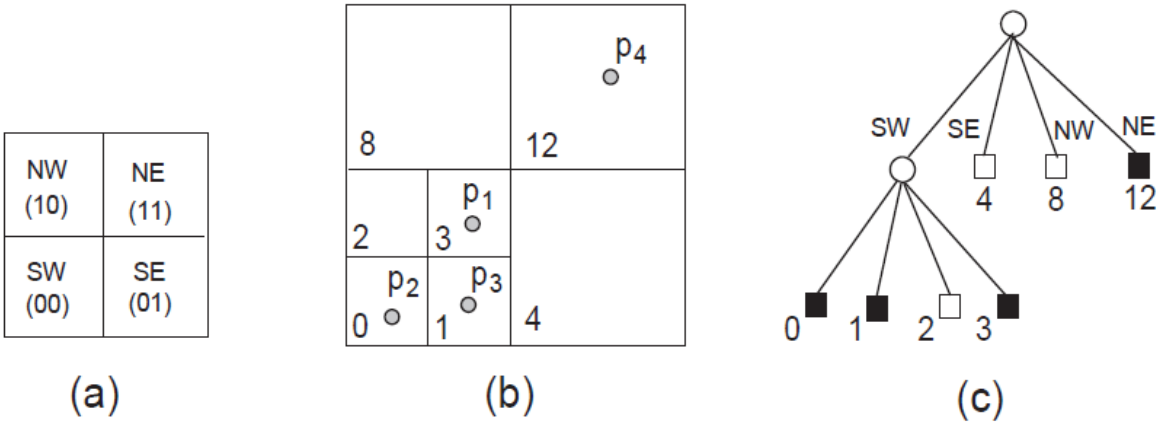

Fig. 2. Quadtree

Clusters dynamically configure computer resources among shared users, which increases application interference. Clusters computing requires a working solution to suit different calculations with the rapid increase in applications. Spark is a simultaneous computing and data analysis system developed by the UC Berkeley AMP Lab Laboratory for popular clusters. Spark offers a very good platform to integrate parallelize applications, which offers a perfect solution for this project.[2 spark]

Study

Quadtree Construction. The quadtree describes a two-dimensional partition of space by dividing the region into four equal quadrants, subquadrants, and so on, with each leaf node containing information corresponding to a particular sub-region. Each node in the tree either has exactly four children, or has no children (a leaf node). The height of quadrants following this strategy of decomposition (i.e. subdividing subquadrants as long as there are interesting data in the subquadrant for which more refinement is desired) is responsive and dependent on the spatial distribution of interesting areas in the decomposing space.

A quadtree region with a depth of n may be used to represent an image of $2^n x 2^n$ pixels where each pixel value is 0 or 1. The root node represents the whole region of the image. If the pixels are not completely 0s or 1s in any area, they are subdivided. Every leaf node in this application represents a pixel block that is all 0s or all 1s. Note the possible storage savings when these trees are used to store images; images often have many regions of considerable size with the same overall color quality. Rather than store a large 2-D array of each pixel in the image, a quadtree can capture potentially many divisive levels of the same information higher than the pixel-sized cells we would otherwise need. Pixel and image sizes limit the tree resolution and overall size.

A area quadtree can also be used as a representation of a data field with a variable resolution. The temperatures in an area, for example, can be stored as a quadtree, each leaf node storing the average temperature over the subregion it represents. If a quadtree region is used to represent a collection of point data (such as a set of cities ' latitude and longitude), regions will be subdivided until each leaf contains a total of one point or n if its capacity is n.

As illustrated in Fig. 2(a), if the quadrants resulting from the split are numbered in the order SW, SE, NW and NE. By combining the split codes in each subdivision, the code can then be generated. For illustration, Fig. 2(b) The number. 2(c) display the space partition and the corresponding tree structure of a single quadtree for a particular set of points p1, . , p4 where the codes of the leaf nodes are marked. As illustrated in Fig. 2(c), a circle and a square are used in the paper to denote the non-leaf node and the leaf node. In addition, if it is not empty, a leaf node is set to black, i.e. it contains at least one level. Otherwise, it is a white leaf node.

Remember that the quadtree structure is retained in the paper as the objects ' space partition-based signature, and thus the quadtree node level is accessible during the request processing process. Therefore, the correct node (region) can be developed based on the code and the level information. For the linear quadtree, only one dimensional index structure is used to keep the black leaf nodes on the disk.[1 il quadtree]

Quadtree Query. A request scope is an important function that you can do with a Quadtree. It's the form in general: get all the points within a spatial distance. A ride-sharing service in real-life applications wants to expose an API to provide cars within 1 km of a user's venue, or the "nearby friends" function of Facebook wants to expose a list of friends in a user's area.

In many implementations, the boundary of a quadtree area also needs to be extracted. To derive the contour of an image represented by a quadtree for examples, or to obtain the boundary codes from a quadtree [Region]. As you can see in Fig. 3, the query area is marked with red box and collected points is red dot.

For the border pixels extraction, the algorithm is trivial if the quadtree is constructed as in previous section. For every processors children whose color attribute is black, examines its four neighboring pixels. For every children who intersect with the query, recursively query on its child. Otherwise, do nothing. The algorithm requires O(logN) computation time to derive the neighbours, and it takes constant time for the border test to complete. Therefore, the time complexity of query extraction algorithm is O(logN).[4]

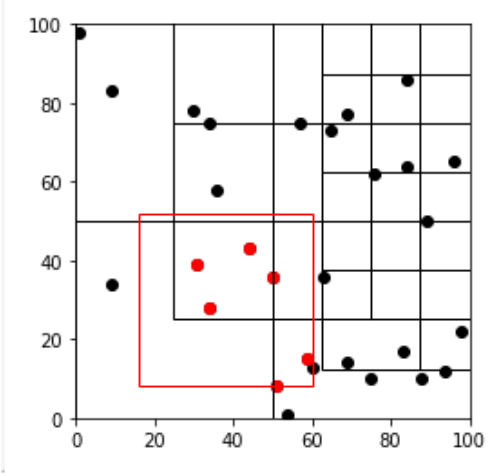

Fig. 3. Quadtree query

Parallel Quadtree Query

I will introduce a simple parallel finding algorithm in this section, and then present algorithms for perimeter calculation and border extraction using the neighbor finding technique. Assuming there are nine processors in the cluster, I aim to partition these objects into nine groups in order to significantly reduce the number of false hits. [4]

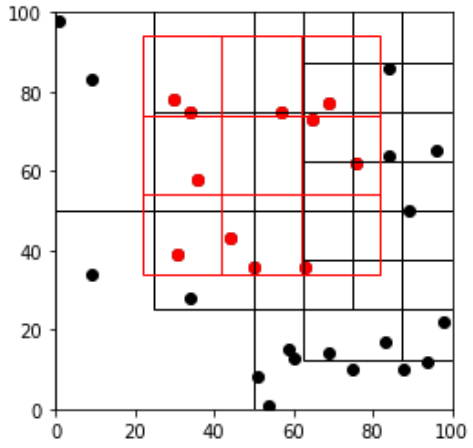

Fig. 4. Parallel Quadtree Query

Simply broadcast the tree to each processor for the parallel query. I break the query into smaller ones, as you can see in Figure 4. After split, I scatter all the smaller queries into multiple processors then for every processor will querying the the smaller query to the broadcasted quadtree. After that, every processor return all the corresponding points back.

Implementation

In this section, I will introduce implementations of parallel finding algorithm of quadtree by introducing its suppporting class.

Class Point Implementation. This is point class, represent 2d coordinate for point, therefore it is simple class consist for x and y values represent its position based on x-axis and y-axis.

```
1
2  class Point:
3      def __init__(self, x, y):
4          self.x = x
5          self.y = y
6
7      def __eq__(self,other):
8          return other.x == self.x and other.y ==
       self.y
```

Class Rectangle Implementation. This is rectangle class, represent 2d rectangle consist of x and y values for its position. For addition it have w and h value mean width and height of the rectangle. Note that I have function contains for checking if points inside of rectangle or not. Moreover I have function intersect that interact with other rectangle, by checking if between 2 rectangle intersecting or not. Note that x and y represent mid point of the rectangle. This support class have good impact for querying due to its contains and intersects functions.

```
1  class Rectangle:
2      def __init__(self, x, y, w, h):
3          self.x = x
4          self.y = y
5          self.w = w
6          self.h = h
7
8      def contains(self, point):
9          return (point.x >= self.x - self.w) and \
10             (point.x < self.x + self.w) and \
11             (point.y >= self.y - self.h) and \
12             (point.y < self.y + self.h);
13
14     def intersects(self, rect):
15         return not (rect.x - rect.w > self.x +
       self.w or \
16           rect.x + rect.w < self.x - self.w or \
17           rect.y - rect.h > self.y + self.h or \
18           rect.y + rect.h < self.y - self.h);
```

Quadtree Construction Implementation. This is the construction part. In the init I have several variable which is capacity, boundary, points, and divided. For capacity is int variable that represent how much maximum points in one box until it divided into smaller box, in this project is 4. Boundary is rectangle object that represent position and width and height of the quadtree. Points is list points inside quadtree.

For divided variable is boolean variable that represent black or white explained in the previous sections.

```
class QuadTree:
    def __init__(self, boundary, n):
        self.capacity = n
        self.boundary = boundary
        self.points = []
        self.divided = False

    def subdivide(self):
        x = self.boundary.x
        y = self.boundary.y
        w = self.boundary.w
        h = self.boundary.h
        ne = Rectangle(x + w / 2, y - h / 2, w / 2, h / 2)
        self.northeast = QuadTree(ne, self.capacity)
        nw = Rectangle(x - w / 2, y - h / 2, w / 2, h / 2)
        self.northwest = QuadTree(nw, self.capacity)
        se = Rectangle(x + w / 2, y + h / 2, w / 2, h / 2)
        self.southeast = QuadTree(se, self.capacity)
        sw = Rectangle(x - w / 2, y + h / 2, w / 2, h / 2)
        self.southwest = QuadTree(sw, self.capacity)
        self.divided = True


    def insert(self,point):
        if(not self.boundary.contains(point)):
            return False

        if len(self.points) < self.capacity :
            self.points.append(point)
            return True
        else:
            if not self.divided:
                self.subdivide()

            if self.northeast.insert(point):
                return True
            elif self.northwest.insert(point):
                return True
            elif self.southeast.insert(point):
                return True
            elif self.southwest.insert(point):
                return True


    def query(self, query, found = []):
        ...
```

In this implementation of constructing quadtree, I have two main function, which is insert and subdivide. Insert is function for inserting point into quadtree. Note that at first quadtree is a simple box. Once it exceeded its capacity it will divided into 4 region as explained before which is northeast(NE), northwest(NW), southeast(SE) and southwest(SW). Once it reach its maximum capacity it isn't insert the point to the its boundary instead its quadtree children. Subdivide function is function for handling the subdivide process by initiate its children corresponding to its position and size. Moreover it changes its divided variable to true.

Quadtree Query Implementation. For query implemenation I have one main function query and its helper. The query function have two parameters query and found. Query is rectangle object represent query area while the found is points temporary variable to store result. Since its recursive therefor its need to store it in found.

```
class QuadTree:
    def __init__(self, boundary, n):
        ...

    def subdivide(self):
       ...


    def insert(self,point):
        ...

    def join_helper(self,tmp, found):
        if tmp is not None:
            for t in tmp:
                if t not in found:
                    found.append(t)
        return found

    def query(self, query, found = []):
        if not found:
            found = []

        if not self.boundary.intersects(query):
            return []
        else:
            for p in self.points:
                if query.contains(p):
                    found.append(p)
            if (self.divided):
                tmp = self.northwest.query(query, found)
                self.join_helper(tmp, found)
                tmp = self.northeast.query(query, found)
                self.join_helper(tmp, found)
                tmp = self.southwest.query(query, found)
                self.join_helper(tmp, found)
                tmp = self.southeast.query(query, found)
                self.join_helper(tmp, found)

        return found
```

As you can see in the code, the line 20 is for initialization. In line 23, it checks itseflt if it intersect with query, if its not then it return empty list. Therefore I don't need to continue to explore its children. Otherwise, in line 25 if its not intersect then I need to continue to query its children. Lastly it returns the result points.

Split Query Implementation. In this part, it is split rectangle to multiple smaller rectangle. As you can see in line 2 it has parameter n with default 3. It means it will divided its heeight and width by 3 resulting 3x3 smaller rectangle. This

function have important role for parallelize query, which is explain in next section.

```
1
2  def split_rec(rec, n = 3):
3      x = rec.x
4      y = rec.y
5      w = rec.w
6      h = rec.h
7      new_w = w/n
8      new_h = h/n
9      xs = np.linspace(x-2*new_w,x+2*new_w,n)
10     ys = np.linspace(y-2*new_h,y+2*new_h,n)
11     rect_list = []
12 #      print(xs[1])
13     for i in range(n):
14         for j in range(n):
15             rect = Rectangle(xs[i],ys[j],new_w,
       new_h)
16             rect_list.append(rect)
17
18     return rect_list
```

Parallel Query Implementation. In this part I assume that I already intialize spark context as sc. I divide the parallel implementation to 3 stage. In stage 1 line 2 named broadcast quadtree. It simply broadcast quadtree object to n processor. In this part, spark offer very good platform to broadcast object. In next stage, stage 2 line 6, I get help by previous function split rect to split the query into smaller rectangle. After that in line 7, I parallelize the query into RDD object provided by spark. Last stage, stage 3 in line 10, I make query function for RDD object based on spark platform. In the line 11 I found that we need to import all the object Point, Quadtree and Rectangle to every processor. Inside the function, simply query it using query function. After that return all the collected points from rdd to the master processor by using collect() function from spark.
As you can see, due to spark platform the code for parallelization seems very simple. Spark already handle parallel configuration for multiple computer resources.

```
1  def parallel_query(qt,query_box):
2      #bc qt
3      bc_qt = sc.broadcast(qt)
4
5      #parallelize query
6      query_list = split_rec(query_box, 3)
7      rdd_query = sc.parallelize(query_list)
8
9      #map query
10     def query(query_box):
11         from Quadtree import Rectangle, QuadTree,
       Point
12         points = bc_qt.value.query(query_box)
13         return points
14
15     result = rdd_query.map(query)
16     return result.collect()
```

Experiment

A. Experiment Settings. All indices and algorithms were implemented in Python. Desktop PC were used for evaluation, equipped with 7th Gen Intel® Core™ i7-7700HQ Processor, 16GB RAM, and a 1TB SATA disk. I evaluate both response time.

B. Dataset. For the dataset I generate the dataset using random library from python. This is the implementation.

```
1  import random
2  N = 20000
3  r = Rectangle(0,0,N,N)
4  qt = QuadTree(r, 4)
5  for i in range(10000):
6      x = random.randint(0, N )
7      y = random.randint(0, N )
8      p = Point(x,y)
9      qt.insert(p)
```

C. Experiment variable. For the experiment I compare between two algorithms, parallel and non parallel algorithm. The number of field sizes (f) varies from 1000 to 20000. field sizes means the area of the dataset, for example if f is 1000 therefore the total area of dataset is 1000x1000 pixels.

The number of dense sizes (d) grows from 10 to 8000. Dense size means that the number of points inserted to the quadtree. The number of query sizes(q) gros from 10% to 30%. The number of query size not grow more thatn 30% because it takes to long for non parallel to compute. By default, f, d and q are set to 20000, 1000 and 25% respectively. Fig. 5 shows response time by using default paramter.

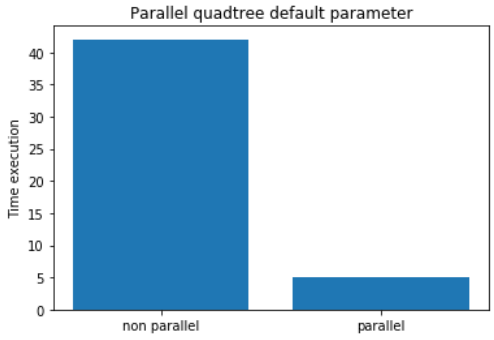

Fig. 5. Experiment by default parameter

Next experiment is varying f variable. Fig. 6 show that time execution as y-axis and field size from 1000 to 20000 as x-axis. Blue line as non parallel execution and orange line as parallel. By the graph we can see that, size of the field not necessarily make execution time longer both in non parallel and parallel. As we can see clearly that parallel application perform better for every cases.

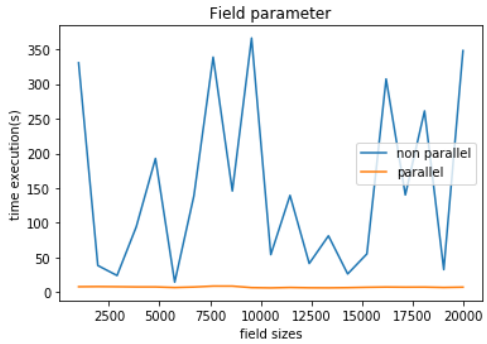

Fig. 6. Experiment varying response (k)

The next experiment ranges from of d. Fig. 7 Show timing as y-axis and dense size as x-axis from 10 to 8000. Note that denser means more points in the dataset. Similar to previous expertiment blue line as non parallel execution and orange line as parallel. From the experiment we can see that at first non parallel is faster that parallel one, but once the number of points grow, the time execution for non parallel grows faster than parallel one. Even the parallel one looks constant compare to non parallel.

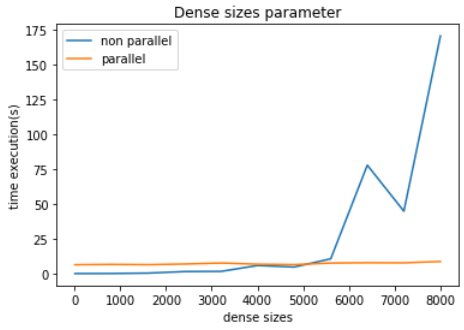

Fig. 7. Experiment varying dense (d)

Similar to previous experiment, in this experiment I varying the variable of query size. Query size from Fig. 2 is the size of red box. Bigger query size means bigger red boxes and relatively more points returned.

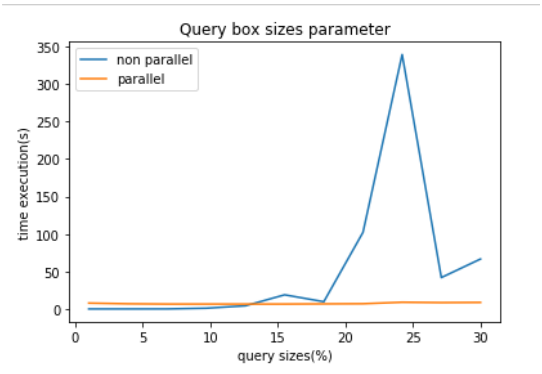

Fig. 8. Experiment varying query size (q)

In that graph we can see that at first when the query size small, the non parallel is faster. As the query size grow the non parallel execution time grow faster than parallel. Therefore it is clear that parallel method is faster and scalalble for large scale data management.

Conclusion. The problem of spatial search is important due to the increasing amount of spatio objects collected in a wide spectrum of applications. In the paper, we propose a parallel Quadtree query implementation using spark. A parallel algorithm is developed to support the spatial search by taking advantage of the Quadtree. We further experiment based on the parallel method and compare its effectiveness with non parallel quadtree query. The result of experiment is the parallelization of query enhance the performance of the system. Our comprehensive experiments convincingly demonstrate the efficiency of our techniques.

Bibliography

1. Koji Kobayashi Kazuki Ishiyama and Kunihiko Sadakane. Succinct quadtrees for road data. *Springer International Publishing AG*, 2017.
2. Yujie Zhang Zhijie Han. Spark a big data processing platform based on memory computing. *Seventh International Symposium on Parallel Architectures, Algorithms and Programming*, 2015.
3. Wenjie Zhang Xuemin Lin Chengyuan Zhang, Ying Zhang. Inverted linear quadtree: Efficient top k spatial keyword search. *IEEE Transactions on Knowledge and Data Engineering*, 28, 2016.
4. Ruen-Rone Lee Shi-Nine Yang. Parallel quadtree construction and manipulation algorithms on hypercubes. *ICCI*, 1994.