Parallel Execution of Spatial Queries on Shared Memory Machine

[EECS584 Final Project]

Qijun Jiang
University of Michigan
qijun@umich.edu

Yicong Zhang
University of Michigan
ianzyc@umich.edu

Zhizhong Zhang
University of Michigan
zhangzz@umich.edu

ABSTRACT

Spatial data is becoming more and more important nowadays as geographic applications become more and more related to human life. Spatial search is the most common operation on spatial data objects that often cover a multi-dimensional space, for example to find all cafe shops within 5 miles of current location. It is important to efficiently retrieve spatial data objects based on their multi-dimensional locations. R-tree is a good solution and has been used in many major DBMSs such as MySQL and PostgreSQL. In current MySQL R-tree implementation, it is one thread per connection serve mode and single node cluster can only support serial tree search. Based on ideas proposed on Parallel R-tree, we intend to explore different parallelization models to parallelize R-tree search process.

This paper presents the implementation of two parallel algorithms, centralized master-worker and decentralized peer-to-peer stealing schedulings, and a client-server system called MiniSpatialDB for evaluating the performance of different algorithms. The parallel algorithms are built on top of an implementation of serial R-tree. The experiment on synthetic data sets shows that the throughput has been greatly improved for both parallel algorithms, and the maximum speedup is one order of magnitude.

Keywords

Spatial Search, R-trees, Parallel R-trees, Shared-Memory, Master-Worker, Peer-to-Peer Stealing, SQL Query

1. INTRODUCTION

Spatial data, also known as geographic data, is becoming more and more important nowadays as geographic applications become more and more related to human life. For example, many people use Yelp to find the nearest restaurants and Google Map to navigate cars. Sometimes, it is not enough to use point locations to represent spatial data objects, and spatial data objects often cover a multi-dimensional space. For example, map objects like countries, cities, and facilities are often represented in two-dimensional area. Spatial search is the most common operation on spatial data objects, for example to find all cafe shops within 5 miles of current location. Spatial search has been widely used in geopraphic applications, computer aided design (CAD), and geographic information systems (GIS) [4]. Therefore, it is important to efficiently retrieve spatial data objects based on their multi-dimensional locations.

The most natural solution is to create an index on spatial locations. For one-dimensional spatial data, there are some known solutions but none of them apply to multi-dimensional spatial searching. Hash table, structure that can map keys to values, is good for searching for existence, but it does not apply to range searching. B-trees, a self-balancing tree data structure that keeps data sorted, is good for range searching, but it does not apply to multi-dimensional space. There have been a large number of structures for dealing with multi-dimensional spatial data, and R-tree is a good solution and has been used in many major DBMSs such as MySQL and PostgreSQL.

R-tree is a height-balanced tree data structure for indexing multi-dimensional spatial data objects such as spatial coordinates, rectangles or polygons. In R-tree, index records are stored in the leaf nodes and the search execution starts from the root to leaf nodes. In current MySQL R-trees implementation, it is one thread per connection serve mode and single node cluster can only support serial tree search. Based on ideas proposed on Parallel R-tree [6], we intend to explore different parallelization models to parallelize R-tree search process.

There has been many parallel paradigms such as pipelining, ring-based applications, master-worker, and cellular automata. In this project, we implemented two different parallelization models, master-worker scheduler and peer-to-peer stealing scheduler, on top of a C++ implementation of serial R-tree algorithm [5]. Master-worker has low job propagation latency and easiness to implement, while peer-to-peer has great scaling up property and no lock contention. In order to test and compare the performance, we built MiniSpatialDB, a simple client-server system to execute SQL spatial search queries. We performed runtime and speedup experiments on BigData server using synthetic data sets. The experiment result shows that the throughput has been greatly improved for both parallel algorithms, and the maximum speedup is one order of magnitude.

1.1 Paper Organization

In Section 2, we review original R-tree data structure and previous work on parallel algorithms, mainly focusing on centralized master-worker scheduling and peer-to-peer scheduling. The system design and parallel algorithm implementations are discussed in Section 3, as well as the tradeoffs between master-work and peer-to-peer. We evaluate different algorithms in Section 4, including runtime and speedup experiments on synthetic data sets testing on BigData server. Section 5 details our findings and ideas for future work. We conclude the paper in Section 7.

2. BACKGROUND & PREVIOUS WORK

In this section, we will give a brief review on original R-tree first proposed by Antonin Guttman in 1984 [4], and literature review on different parallel algorithms, mainly focusing on centralized master-worker scheduling and decentralized peer-to-peer scheduling. For centralized control, we used the most well known master-worker model [1] as a starting point. The balanced work stealing (BWS) [2], a work-stealing scheduler for multicore systems, is also being examined. Together with the implementation of BWS strategy, we considered the idea of terminating job on the condition of all threads are idle [3].

2.1 R-Tree

R-tree is a height-balanced tree data structure for indexing multi-dimensional spatial data objects such as spatial coordinates, rectangles or polygons. R-tree is similar to B-tree where index records are stored in its leaf nodes and the tree is designed so that search requires visiting only a small portion of nodes. R-tree was proposed by Antonin Guttman in 1984 and has been widely used in theoretical and practical methods [4].

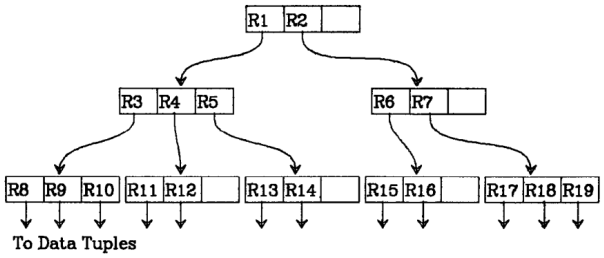

Figure 1: R-tree structure

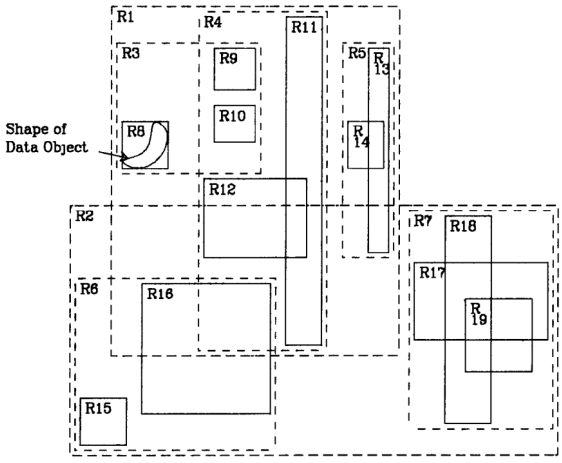

Figure 2: Rectangle representation

Here we will briefly introduce the data structure of R-tree for rectangle objects. Define M be the maximum number of entries in one node, and m be the minimun number of entries in one node, where $m \leq \frac{M}{2}$. Then every leaf (non-leaf) node has r index records (children), where $r \in [m, M]$. Each entry in non-leaf node contains the smallest rectangle that spatially covers all rectangles in the child node. All leaves appear at the same level of the tree structure. Figure 1 and 2 shows an example of the tree structure and the rectangle representation [4], where the maximum number of entries in one node is $M = 3$ and index records are $R_8 \sim R_{19}$.

The search operation on R-tree is similar to B-tree, for example to find all index records that overlap a search rectangle A. It is carried out from root to leaf, and recursively invoke search on sub-tree whose root node overlaps rectangle A until reach leaf records. In this paper, we referenced R-tree codes from online git repository, which is a C++ templated version of the R-tree algorithm [5].

2.2 Parallel Algorithms

Multiple programming models are emerging to address an increased need for dynamic task parallelism in applications for multi-core processors and shared-address-space parallel computing. The parallelism can be implemented in either centralized or decentralized schedulers. The centralized scheduler, for example master-worker, has one master to manage all tasks, and multiple workers to run their own work locally. Centralized approach is direct and fast, but the lock contention is heavy when the number of workers becomes large. Another approach, decentralized scheduler such as peer-to-peer, has only peers (or workers) and no master. The tasks are balanced among peers to reduce the lock contention, however, the propagation path may be long when the number of peers becomes large.

2.2.1 Master-Worker Scheduler

Master-worker is one of the most common parallelism schedulers, which is shown in Figure 3 below. Generally, a single master stores and manages the tasks, and also holds the result. Multiple workers get new tasks from the master when they are in idle state, and complete their tasks locally. In a shared-memory system, things are slightly different. The collection of tasks is usually a global data structure, and the result is a global variable. Thus, all of the threads can work as both workers to get tasks from global data structure, and managers to add tasks. Since every thread can access to global data structure and change the result, locks are used to protect global variables and ensure all workers will not interrupt with each other when updating global variables. Ideally, if the work is divided into subtasks to run on n cores simultaneously, the runtime can be reduced to $1/n$ of the original value, assuming all processors start and end at the same time and there are no latches or extra communications.

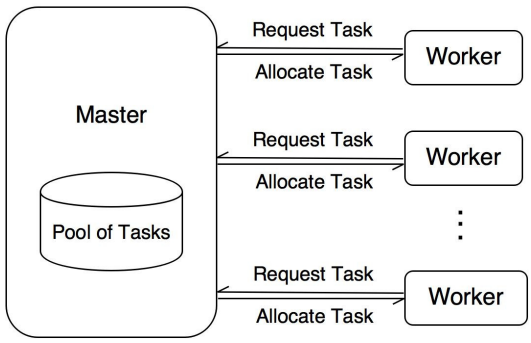

Figure 3: Master-worker scheduler

This approach allows the master to assign tasks to some worker directly without any other propagation or delays, however, it has two problems. First, the task is hard to be evenly divided into subtasks due to the existence of bound condition. It is common that some processors complete their local tasks earlier than others, and stay in idle state until the program terminates. The performance is reduced in this situation, because not all processors are been fully utilized during whole execution time. Second, the lock contention for global data structure is heavy when a large quantity of processors are involved. Multiple processors are waiting to obtain the lock without doing anything, which will compromise the overall performance.

There are some possible solutions for the first problem, for example, to balance the workload of workers dynamically. It requires a processor to push back part of its task to the global task pool, if the task is detected to be too large to finish. Unfortunately, there is no effective way to solve the second problem, because the lock contention to the global task pool is inevitable.

2.2.2 *Peer-to-Peer Scheduler*

In master-worker scheduler, as mentioned in Section 2.2.1, the global lock must be hold whenever a worker is trying to insert or pull partial jobs from the global data structure. The latching in centralized scheduler is high, and decentralized work scheduling design would prevail in terms of efficiently balancing the workload between larger number of workers.

Scale Up A typical task-parallel system consists of a pool of workers, and the number of workers represents the number of underlying computing nodes. Two well-known scheduling paradigms, *work sharing* and *work stealing* (shown in Figure 4), are used to address the problem of scheduling multithread computations among the workers. In work-sharing, whenever a new task been generated on a node, the scheduler on that node would soon start to re-distribute tasks to its peers task pool. In work-stealing, each worker maintains its own pool (stack) of tasks, and the underutilized workers take the initiative to steal work from other busy workers. Workers that create the new task pay a small overhead to enable stealing.

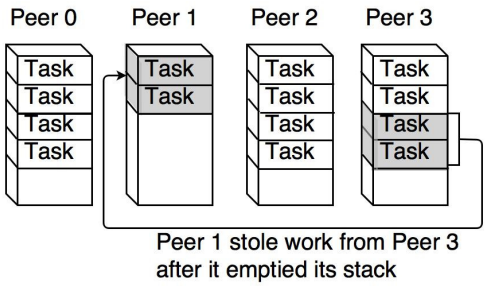

Figure 4: Peer-to-peer work stealing model

Circular Model A trivial design to achieve low locking overhead would be circular work stealing design, which is shown in Figure 5. In circular work stealing, the initial search job is sent to a randomly selected head. During the parallel search process, the communication group of each worker (say w) will contain the preceding worker (w_p) and following worker (w_f). When worker w has exhausted its local stack, it will notify the preceding worker w_p and stay in idle state while waiting for more jobs. During the normal local search, worker w will check for the idle message from its following worker w_f on a regular basis. Once the idle message is received, worker w will split half of its jobs on local stack to the idle worker w_f. As explained earlier, there is no locking overhead in circular work stealing, but the major drawback is job propagation. If all but one worker w_1 runs out the local search job, then the last worker w_n will have to wait for w_0 to firstly run some local search to expand its search job stack, and split part of them to w_1. Then w_1 to firstly run some local search to expand its search job stack, and split to w_2, w_3, $\cdots$ and so on, until w_{n-1} got some job from its preceding worker to finally generate job for splitting to w_n. In this situation, the path of job propagation is very long. Although all workers do not have to wait for locks, many idle workers will have to wait for a long time to restart local search, and it can greatly decrease the efficiency of parallelization.

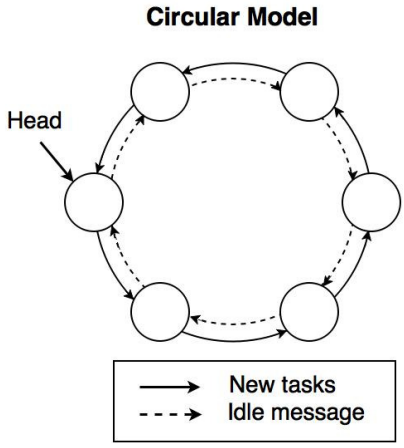

Figure 5: Circular work stealing model

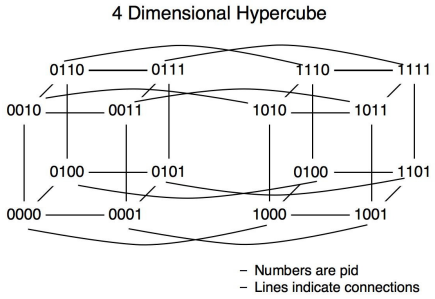

Figure 6: 4-dimensional hypercube work stealing model

Hypercube Model The distributed memory single instruction multiple data (SIMD) hypercube model, which is shown in Figure 6, can achieve a better balance between optimality of the job propagation path and minimum locking overhead of shared resource to construct the communication group [7]. It has n processors in total, where $n = 1, 2, 4, 8, 16, \cdots$ (power of 2). Each processor p_i has a processor number i in local variables, where $0 \leq i \leq n-1$. For processor p_i, the coordinates are its processor number i expressed in $\log_2 n$ bits. For example, if $n = 32 = 2^5$, then the

coordinates are expressed in 5 bits. Then for processor p_6, it has coordinates 00110 (binary in 5 digits). Two processors are connected if and only if their coordinates differ in exactly one bit. For example, 00110 is connected to 10110, 01110, 00010, 00100, and 00111. Note that it is the same as saying that p_i and p_j are connected if and only if $|i-j| = 2^k$, where k is the digit number they differ at ($k = 0, 1, 2, 3, 4$).

Though peer-to-peer scheduler solve the lock contention problem, its overhead to steal a task in the group may effect the performance since the propagation path is long. The time complexity for a worker to get new task from master is O(1), while it takes at most O(log n) time for a peer to steal its task. We need further evaluation over the implemented algorithms to find the optimize approach.

3. SYSTEM DESIGNS

In this section, we will discuss our system designs in detail. Both centralized and decentralized schedulers have their own advantages in balancing the workload to multi-processors in shared-memory machine. To find out the optimal approach for in-memory parallel R-tree, we implemented both centralized master-worker scheduling algorithm and decentralized peer-to-peer work stealing scheduling algorithm. To fully evaluate the performance of parallel R-tree search, we built a client-server system called MiniSpatialDB.

3.1 Overview

For the full functionality and evaluation of the parallel R-tree schedulers, we implemented a client-server system called MiniSpatialDB (Figure 7). The role the client plays is tp initiate SQL query execution. Whenever a new connection is created, the client sends SQL queries serially in a single transaction to the server and waits for results. Meanwhile, the client will record the start time and end time of the connection to estimate the query runtime. For server, after receiving request from client, it will store the data and execute the queries. First, the server executor will call parser to parse incoming SQL query into an instruction object that can be executed later. The server receives and executes queries continuously until it receives a termination signal to stop the connection. For the data storage, the table content and the spatial indexes are stored separately for control simplicity which will be discussed in Section 3.1.2 below.

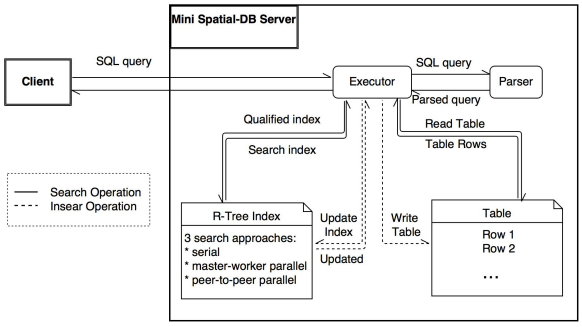

Figure 7: System Overview

3.1.1 Parser

The parser is designed to handle SQL queries and extract and pass information such as query type, table name, and spatial data object to the executor. In general, our parser has the ability to handle four standard SQL query types: `DROP TABLE`, `CREATE TABLE`, `INSERT`, and `SELECT`. For simplicity, our parser does not support all syntax for these four SQL query types, and it only supports syntax related to spatial data objects loading and spatial searching. For `CREATE TABLE`, our parser supports spatial index on geometry and one additional attribute (string or double). For `INSERT`, our parser supports inserting geomery and attribute pair. For `SELECT`, our parser supports `ST_CONTAINS` operation on spatial index and `WHERE` clause on attribute.

3.1.2 Executor

As mentioned in Section 3.1, the table content and the spatial indexes are stored separately in *Table* and *RTree* respectively. The executor is responsible for the read and write operation on *Table* and *RTree*. After a raw SQL query received from the client been parsed, the executor will pick different execution types for different SQL query types, and then apply attributes comparing functions for different arithmetic operations such as greater than $\geq$ or equal to $=$. For example, when it is an `INSERT` query, the executor will insert spatial index into *RTree* and table attributes into *Table*. For `SELECT` query, the executor will search for qualified indexes in *RTree* and scan the corresponding attributes. Here we have three different search algorithms: serial, master-worker and peer-to-peer stealing.

3.2 Master-Worker R-tree

Master-worker scheduler contains a master to send the jobs, and multiple workers to complete the assigned tasks. As mentioned in Section 2.2.1, in a shared memory, each processor can take the responsibility of both the master and the worker. The tasks pool of the master is stored as a global data structure and every worker can perform operations over the pool. Load balancing and lock contention are two most significant reasons to affect the performance. For load balancing, the divergence of workload in each thread should be low. That means we expect to avoid idle state during execution and minimize the difference of the execution time for each thread. It requires suitable load balancing strategies. For lock contention, excessive access to global data structures should be avoided.

3.2.1 Depth-First Search and Breadth-First Search

In serial R-tree search algorithm, the Depth-First Search (DFS) is utilized since it requires less memory space. However, DFS is not a good choice for the master to generate new tasks since their size may vary a lot. It is not worthwhile to assign to workers tiny tasks that can be completed in a few cycles. Thus, Breadth-First Search (BFS) is more suitable for global queue generation since it produces tasks with similar size. As shown in Algorithm 1, our program takes a combination of both DFS and BFS. We have one processor, which acts as the master temporarily to perform BFS for a few levels until enough tasks are contained in the global queue. After that, all processors start working by copying new tasks from global queue and run DFS locally.

3.2.2 Task Push-Back

Due to the existence of constraints, the size of a single task may vary a lot. It happens when some workers finish their tasks and get new tasks from global while the other workers

are still running old tasks. The tasks in global queue are consumed and the global queue size decreases. When the size of global queue is less than the total number of workers (or N), workers will put part of the local tasks back to the master to rebalance the work. The reason to use N as the update criteria is that when there are more than N tasks, the global queue will not be empty even if all workers finish their local tasks and request for new tasks. It ensures that all workers keep running until the search is completed.

Algorithm 1 Master-Worker Scheduler

```
1: Input:    root // root node of R-tree
2:           box  // coordinates of the search box
3: Output:  globalResult
4: Initialization: globalQueue := {root};
5:                 globalResult := 0;
6:                 localResult := 0;
7:                 localStack := {};
8: while size of globalQueue < number of threads do
9:     expand(globalQueue);
10: end while
11: in parallel
12: while globalQueue not empty do
13:     move one task from globaQueue to localStack;
14:     while localStack not empty do
15:         if is MAX-th cycle then
16:             update(globalQueue);
17:         end if
18:         n := one node in localStack;
19:         if n is internal and overlaps with box then
20:             expand(localStack);
21:         else if n is leaf and contained in box then
22:             localResult += check(n);
23:         end if
24:     end while
25:     globalResult += localResult;
26: end while
27: end parallel
```

- MAX: constant, used to reduce the frequency to access globalQueue
- expand(): function, take out one internal node and add its qualified children to the task storage structure
- update(): function, if the size of global queue is less than the number of threads, move part of local stack to the global queue for balancing the workload among workers. Otherwise do nothing.
- check(): function, check the non-spatial attribute of the leaf node. If the attribute are qualified, return 1, otherwise return 0.

3.2.3 Global Access Cut-Down

Cuting down the access to global queue is the main way to reduce the lock contention. The program makes it with two policies. One policy is that the global queue gives away large tasks earlier. The data structure used is priority queue, in which the node in higher tree level has higher priority and will be popped out earlier. A higher level tree node stands for a larger subtree, which means it is a longer task. Running longer tasks at first can reduce the communications between master and worker, and therefore reduce the lock contention. Also, a `check_counter` limits the period to update undersize global queue. It makes sure at most one check to the global queue by per processor every `MAX_CHECK_COUNTER` local cycles, which reduces the access to global queue. However, lock contention is still inevitable for new task requests and global queue update. The problem may become significant when the number of processors are large enough.

3.3 Peer-to-Peer Stealing R-tree

In our project, we also implement a decentralized runtime/online scheduler that maps tasks to processing elements dynamically at runtime. The peer-to-peer work scheduling model is a suitable candidate to boost the R-tree search procedure by decentralizing the global work scheduler.

3.3.1 Receiver initiated vs sender initiated

We mentioned work sharing and work stealing mode in decentralized system (Section 2.2.2). In our project, we found work-stealing schedulers more appealing. First of all, work-stealing is considered to be more efficient: the busy worker pays only a small overhead both to enable stealing on task creation and to execute tasks that are not stolen. The majority of the overhead is shouldered by underutilized peers whose cores are idle anyway. Secondly, the memory consumption is bounded throughout the search process, because the size of local stack is proportional to the height of search tree. Work-stealing has many advantages over work-sharing, but the implementation of a work-stealing system is more complicated than that of a work-sharing scheduler. For example, more constrained requirements will be needed on maintaining the idle status of worker (idle/running), and the definition of terminating condition can be complicated for each worker.

3.3.2 Grouping and Job Propagation

Another important decision in the design of peer-to-peer work stealing scheduler is peer groups. Peer group is the definition of the communication group among peer workers. Each worker node communicates with peers in communication group when it performs load balancing functionality. It matters a lot because to a large extent, the decision about the construction of communication group can affect the speed of partial job propagation (affects idle status period) and shared resource occupation (affects locking overhead).

Using previous design of master-worker scheduler as an example, the communication group for each worker node only consists of the global queue. The shared global queue is frequently occupied by every worker node, hence the expected locking overhead can be considerable. On the other hand, since all the partial jobs are pushed to and pulled from the global queue directly, the job propagation path is optimal and the idle status period of each worker is minimized. In peer-to-peer work scheduler, we want to maintain the optimality of the job propagation path and reduce the locking overhead of shared resource.

3.3.3 Load Balancing Strategy

We decide to use the distributed memory SIMD hypercube model (Section 2.2.2) to construct the communication group, since it has better balancing comparing to circular model.

In the hypercube model, the communication group of each worker consists of the peers that are connected to it, i.e., the workers with processor id that differs exactly in one digit in coordinate representation. There are $\log_2 n$ workers in total in its communication group, where n is the total number of processors. Here we summarize the algorithm below:

1. Each worker is associated with a flag to identify the idle/running status, and a lock to protect its data structure. If the worker is running out of local search stack, it will set the flag to be idle and request more jobs from peers in its communication group.

2. When a worker is doing search on its local stack, it regularly obtains lock of peers in its communication group and checks their flags. If a peer is idle, it splits part of its jobs to the peer and unset the flag to avoid other workers' splitting. After the checking and possible splitting, it releases the lock of the current peer.

3. When a worker turns idle, it starts to periodically check its flag until the flag been unset by its peer. When the flag is unset, its stack is filled by the jobs split from its peer, and it can continue the local search.

In the SIMD hypercube model, the lock for each worker is only shared between its peers in the communication group. Since the size of the communication group is only $\log_2 n$, the shared resources will not be occupied by too many workers ($\log_2 128 = 7$). On the other hand, the overlapping of each worker's communication group helps providing an upper bound $\log_2 n$ for the job propagation path. Whenever a worker or some workers turn into idle status, they (it) will not have to wait for long to receive new jobs to restart local search. To sum up, hypercube model works well in our peer-to-peer work stealing scheduler, and it provides low locking overhead and short job propagation path.

4. PERFORMANCE TESTS

We conducted all experiments on BigData server provided by instructor. The server is running RHEL 7.1, has around 1TB of RAM , and has 8 processors with 14 cores each. OpenMP has been used to utilize the number of cores on each single node cluster to parallel search the R-tree. We can simply set the environment variable in OpenMP `export OMP_NUM_THREADS = c` where c is the number of cores to use. We created synthetic spatial data objects (rectangles) and spatial search queries for evaluating the performance of parallel spatial search, and we conducted runtime and speedup experiments with both serial and parallel algorithms. The experiment result shows that: the throughput has been improved for both parallel algorithms (master-worker 15x better and peer-to-peer 12x better); the maximum speedup is 15-20 for master-worker and 8-15 for peer-to-peer, in our synthetic data sets.

4.1 Data Generation

We generated synthetic spatial data objects (rectangles) for evaluating the performance of spatial search. In general, the data generation process is in a uniformly distributed manner. We generated random rectangles within an area, where the center of rectangles is uniformly distributed and the size of rectangles is normally distributed based on mean and variance. These data objects mimic the map objects in the real environment, for example cafe shops, restaurants and hotels in a city. The uniform distribution is used to maximize the R-trees usage and simplify the search process. We want the number of data objects within a search area to be large, so there is a reasonable number of work in R-trees search and we can test on the speed up by Parallel R-trees. By providing a uniform distributed data objects, we can simply control the size of search area to estimate the number of rectangles returned. For each spatial object, we also have an additional attribute, a double value uniformly chosen between 0 and 100, for future spatial search which will be discussed in Section 4.2.

To obtain the dataset mentioned above, our data generation program takes the following parameters which are shown in Table 1.

Table 1: Data Generation Parameters

N	number of random rectangles generated
$\|A\|$	size of the area to cover all rectangles
$\|a\|$	size of single random rectangle

We first choose the size and generate a square area with size $|A|$. Then we choose uniformly distributed N random points within the square area. For each point, we expand it to a rectangle that has average size of $|a|$ and add an additional attribute, a double value uniformly chosen between 0 and 100.

4.2 Query Generation

We generated SQL queries to run on both MySQL and MiniSpatialDB (sequential and parallel). `INSERT` queries are based on the spatial data objects generated above, with an additional attribute (double or string) for further spatial search. `SEARCH` queries are generated according to the user-specified search area size and selectivity on the additional attribute. Our system supports `ST_CONTAINS` operations from MySQL 5.7, which searchs for whether geometry objects contain each other. The additional attribute is significant. First, it is common to seach for spatial objects not only on spatial locations but also other information, for example find all restaurants within a search area and customer ratings are above 4. Second, the existence of additional attribute requires checking on each "candidate" object even with spatial index on spatial locations, which has strong potential to be carried out a parallel speedup.

To obtain the queries mentioned above, our query generation program takes the following parameters which are shown in Table 2.

Table 2: Query Generation Parameters

Q	number of search queries generated
$p\%$	percentage of search area size compared to the whole area size $\|A\|$
$s\%$	selectivity on additional attribute
c	number of cores used in parallel mode

We first create `LOAD` queries to insert all sptial objects generated in Section 4.1 above. Next, we create `SEARCH` queries. First, we obtain randomly distributed M points within the whole area A. For each point, we expand it to a rectangle that has average size of $|A| \times p\%$, user specified search area size. For each search rectangle, we generate corresponding

search query also with `WHERE` clause on additional attribute $val < s$. This mimics the selectivity, because there are estimately $s\%$ rectangles that have val below s. For example if $s = 50$, the search query will return estimately 50% rectangles from ones that meet the first spatial index requirement.

We generated datasets by setting $N = 10^7$, $|A| = 10^{10}$, $|a| = 10^3$ and $Q = 10^3$. We choose different values for p, s and c to conduct different experiments both on MySQL and MiniSpatialDB.

4.3 Runtime Experiment

We created one synthetic search query set ($Q = 1000$, $s = 50\%$) to compare the performance of serial and two parallel algorithms in terms of average runtime per query as increasing the search area size. We also included test result on MySQL for the same queries. The average runtime for a single query on MySQL is relatively large compared to test results on MiniSpatialDB, because our system ignores many aspects a commercial DBMS has such as recovering, logging, and replications which is runtime consuming. The test result on MySQL is only referenced here to see the trend of runtime as increaing the search area size. As mensioned in Section 4.2, we can use p, the percentage of search area size compared to the whole area size $|A|$, to estimate the amount of work for search process in R-trees. Our test scale for the percentage of search area size is from 1% to 5%.

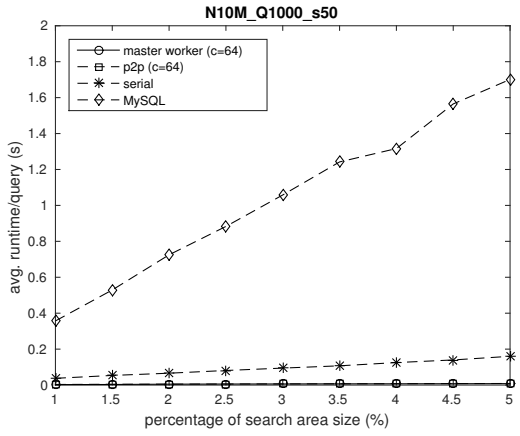

Figure 8: Runtime Experiment Result (Full)

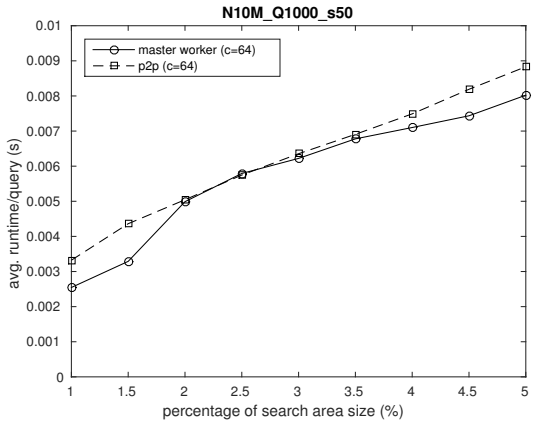

Figure 9: Runtime Experiment Result (Part)

Figure 8 shows the runtime experiment results for all cases: MySQL, serial R-tree, master-worker R-tree and peer-to-peer R-tree. As the search area size increases, the average runtime for single search query is nearly linear for all of them (Figure 9 gives a closer look for two parallel R-tree). This linear result is reasonable, because the search execution time is considered propotional to the number of nodes returned by search algorithm.

Figure 9 shows the runtime experiment result only for two parallel algorithms. Here we used 64 cores for both parallel algorithms, because we found out both of they perform well on 64-core mode in speedup experiments which will be shown in Section 4.4 below. We also compare the throughput for serial and parallel algorithms which is shown in Figure 10. The throughput is greatly improved for both parallel algorithms, where master-worker is around 15x and peer-to-peer is around 12x better compared with serial solution.

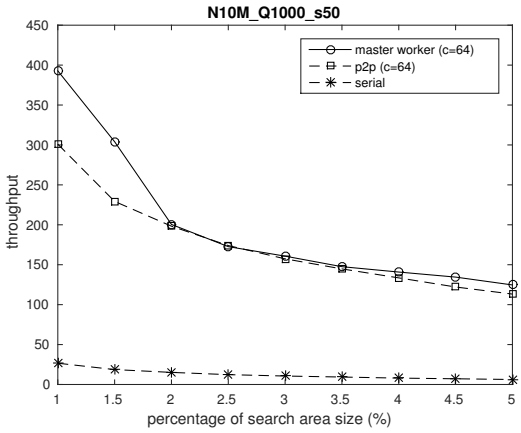

Figure 10: Speedup Experiment Result

4.4 Parallel Speedup Experiment

In this section, we will experiment on the speedup performance on parallel algorithms (master worker and peer-to-peer) compared to serial R-tree. Here the speedup is defined by the execution time in serial R-tree divided by the execution time in parallel R-tree.

$$speedup = \frac{serial\ execution\ time}{parallel\ execution\ time} \tag{1}$$

We created four synthetic search query sets, shown in Table 3, to test the speedup performance on different number of cores used. Our test scale for the number of cores used is $c = [1, 2, 4, 8, 16, 32, 48, 64, 80, 96]$.

Table 3: Parameter Settings: Speedup

Name	Q	p	s
$Q1000.p1.s50$	1000	1%	50%
$Q1000.p3.s50$	1000	3%	50%
$Q1000.p5.s50$	1000	5%	50%
$Q1000.p1.s30$	1000	1%	30%

Figure 11 shows the speedup of two different parallel algorithms for the four synthetic datasets given in Table 3 for

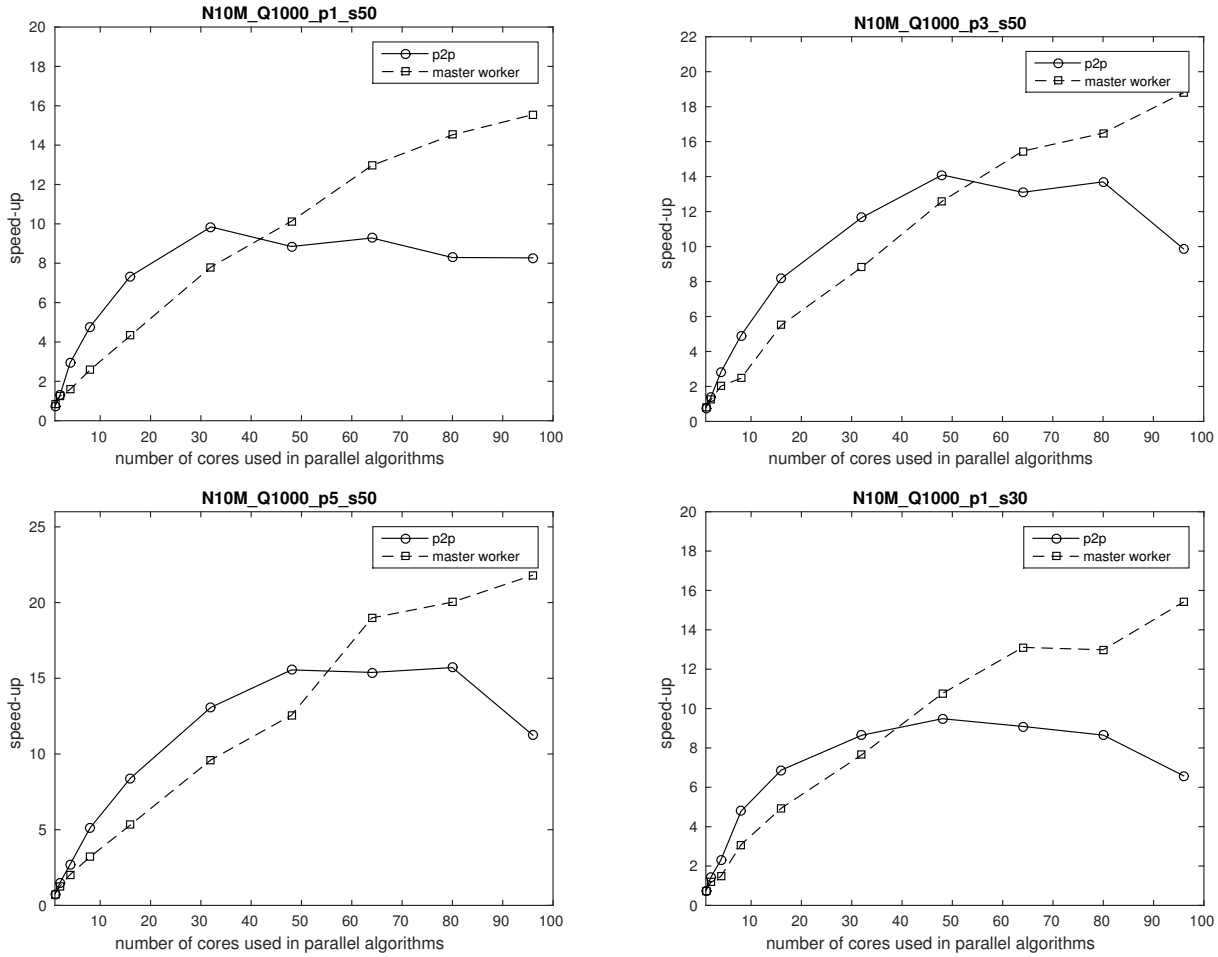

Figure 11: Speedup Experiment Result

increasing numbers of cores used. In general, as the number of core used increases, the speedup for master-worker is nearly linear, and the speedup for peer-to-peer first increases near linearly then stops increasing when c is beyond a certain number ($c > 64$). When c reaches around 64, the system efficiency (speedup divided by c) goes down very quickly for peer-to-peer. As for maximum speedup, master-worker can provide 15-20 times and peer-to-peer can provide 8-15 times speedup for four synthetic sets.

For master-worker design, as the number of cores that try to access the shared global queue goes up, the interference among different cores become significant. It can be mostly attribute to the busy waiting of cores that are trying to push/pull partial jobs to/from the central controller: global queue. However, due to the limitation of the test environment (114 cores max.), we could not see significant system efficiency slow-down. Based on our analysis, we can expect that, as more cores being added to the system (maybe 200), the locking overhead in the master-worker system will greatly increase and the system efficiency goes down.

For peer-to-peer stealing design, it is expected be outperform master-worker due to the scaling up property. In peer-to-peer stealing, the workers do not have to refer to the centralized controller for partial job assignment. Our test result achieves this expectation for small c ($c < 32$). However, our test result also shows that the speedup performance of peer-to-peer is worse than master-worker when c becomes large ($c > 64$). Our best guess is that the speedup performance decrease is due to the long job propagation path when c gets large, and the starting overhead might cause a bigger inefficiency in the functionality of the load balancing.

5. DISCUSSION & FUTURE WORK

Our experiments have shown that parallelizing the `ST_CONTAINS` search operation on R-tree can provide up to 15-20 times (master-worker) and 8-15 times (peer-to-peer stealing) speedup in general spatial query processing. We can optimistically expect the parallelization of other spatial search operations, like intersection search (`ST_INTERSECTS`), would also help boost the spatial index functionality. Based on the experiment result, we find out there are several areas for further exploration:

1. Up to now, there have been many variations of R-tree such as R* tree, R^+ tree and Hilbert R-tree, which provides even more functionalities. It is promissing if parallelism scheduling can apply to their implementations and improve the performance.

2. There are many major databases that are using R-tree

for spatial indexing such as MySQL and PostgreSQL, and they are currently using only serial algorithm for search operation. Future research could be experimenting on those databases to see whether parallelism can make a difference.

3. In this project, we mainly explored two work scheduling models, master-worker and hypercube peer-to-peer stealing. They have their own advantages in terms of reducing lock contentions and minimizing job propagation path. Yet, there are many other work scheduling paradigm proposed. It is worthwhile to implement them and analyze their advantages in boosting the R-tree search operation.

4. The key to analyze different scheduling models is to find a proper metric or method for the measurement of the locking contention cost and long job propagation cost. With such kind of measurement, we can better understand the inefficiency in the system, and choose the most appropriate scheduling model for each algorithm accordingly.

6. CONCLUSIONS

In this project, we understand the implementation of the original R-tree data structure and the serial search operation. To achieve an improvement for spatial search performance, we explored and implemented two parallel algorithms: master-worker scheduling and peer-to-peer stealing scheduling. To evaluate the performance of parallel algorithms, we built a client-server system called MiniSpatialDB and performed runtime and speedup experiments on BigData server using synthetic data sets generated in a random manner. The experiment result shows that: the throughput has been improved for both parallel algorithms (master-worker 15x better and peer-to-peer 12x better); the maximum speedup is 15-20 for master-worker and 8-15 for peer-to-peer, in our synthetic data sets. Our project has shown that parallel algorithms can be applied to the execution of spatial search operations, and it can improve the overall performance.

7. ACKNOWLEDGMENTS

We wish to thank Professor Barzan Mozafari at University of Michigan for his insightful comments and suggestions, Isaac Bowen for his assistance on arranging bigdata server, and multiple authors/contributors of the legacy code for serial implementation of R-tree.

8. REFERENCES

[1] R. Buyya. *High performance cluster computing: programming and applications*. Prentice-Hall PTR, Upper Saddle River, New Jersey, 1999.

[2] X. Ding, K. Wang, P. B. Gibbons, and X. Zhang. Bws: Balanced work stealing for time-sharing multicores. In *Proceedings of the 7th ACM european conference on Computer Systems - EuroSys '12*, pages 195–204, April 2012.

[3] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *2009 IEEE International Symposium on Parallel and Distributed Processing*, 2009.

[4] A. Guttman. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD Record SIGMOD Rec.*, 14(2):47.

[5] A. Guttman, M. Stonebraker, M. Green, P. Brook, G. Douglas, and Y. Barkan. A c++ templated version of the rtree algorithm. https://github.com/nushoin/RTree.

[6] I. Kamel and C. Faloutsos. Parallel r-trees. In *Proceedings of the 1992 ACM SIGMOD international Conference on Management of Data. M. Stonebraker, Ed.*, pages 195–204.

[7] Q. F. Stout. Hypercubes and pyramids. *Pyramidal Systems for Computer Vision*, pages 75–89, 1986.