Parallelized Insertion in R-Trees Using POSIX Threads

Liam Ralph
University of Texas at San Antonio

Abstract

The R-Tree is a data structure used for querying geospatial data introduced by Antonin Guttman in 1984. This paper shows the usefulness of POSIX threads on multicore environments in speeding up R-Tree insertion where the number of index records per node (denoted by M) is large. Much work has been done in the area of parallelizing operations on R-Trees in the hopes of speeding them up. Luo et. all (2012) achieved a 23x speedup in R-Tree construction by using GPUs. Prasad et. all (2015) pushed that to 226x speedup. This paper aims to show how speedup of individual insertions of R-Trees can be achieved through parallelization. Although parallelized insertions lead to slower insertion times for narrower trees, (0.24x speedup and 0.68x speedup for M=128 and M=256 respectively), wider trees saw significant improvements in insertion time (2.2x, 5.5x, 9.5x and 6.9x speedup for M=512, 1024, 2048, and 4096). However, the specificity of conditions required to gain this speedup may render these parallelizations impractical.

Introduction

R-Trees are data structures that consist of three main types of objects: nodes, index records, and spatial objects. Nodes are areas of memory that contain multiple index records. Index records contain a spatial object (a minimum bounding rectangle (MBR) in this case) and a pointer to a child node (if the index record is not at the leaf level) or an object in a database. Index records in an R-Tree contain MBRs that fully contain the MBRs and data objects of any descendent index records. One of the most important functions of an R-Tree is its ability to insert new index records. Because multiple index records in an R-Tree node may represent objects that are spatially overlapping, an R-Tree must perform multiple downward traversals to find a data item that is being queried. As a result, an insertion of a new index record into an R-Tree must perform two functions:

1. Minimize the expansion of parent MBRs (so as not to make future queries more inefficient by increasing the number of index-records they have to recurse down to), and
2. Expand parent MBRs of an index record so as to maintain their validity

The type of R-Trees implemented in this paper are limited to two dimensions, although R-Tree implementations with MBRs of an arbitrary number of dimensions are also possible. This paper implements R-Trees according to the instructions laid out in Antonin Guttman's 1984 paper but with a few differences:

1. The minimum bounding rectangles are limited to two dimensions
2. Leaf-level index records point to NULL rather than real database objects (this is acceptable since we are trying to demonstrate R-Trees' capabilities rather than use them for real-world querying)
3. There is no minimum number of index records in a node - a node can have any number of index records between one and the limit, M, specified by the programmer

The insertion algorithm consists of three parallelizable subroutines. Each of these subroutines has a sequential and parallel version. These routines are ChooseLeaf, PickSeeds, and LinearSplit. These routines are called by the routines InsertAtNode and Insert. Additionally, a routine called AdjustTree is used to maintain the validity of the tree's MBRs. Each of these functions, with the exception of AdjustTree, which is purely sequential, accepts a parameter specifying the number of threads to use.

ChooseLeaf and ChooseLeafParallel

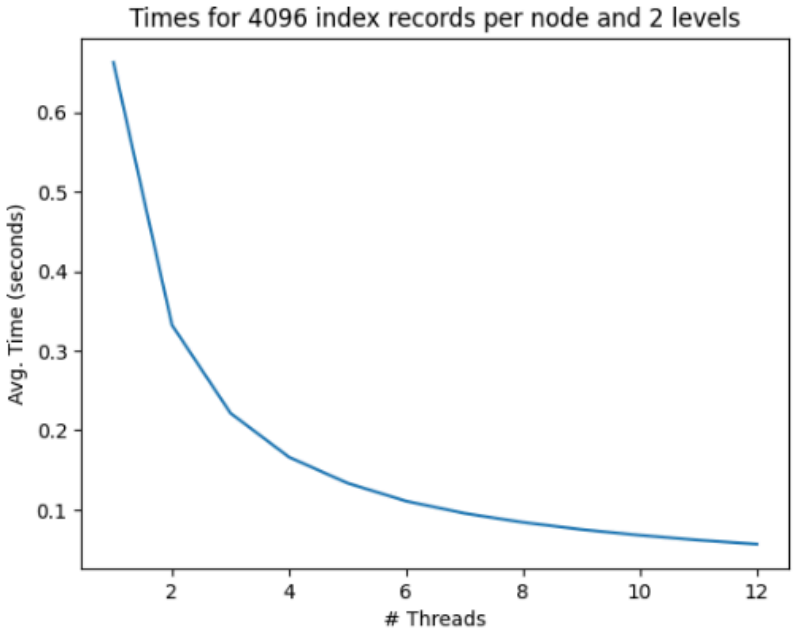
In R-Trees, insertions of new index records are usually done at the leaf level. Thus, one essential step for insertion is finding the optimal leaf node to insert the new index record into such that parent MBRs are minimally expanded. ChooseLeaf performs this function when passed the root (rt) of the R-Tree and the index record (ir) for insertion. This function was implemented according to the pseudocode below, which is based off of Antonin Guttman's original pseudocode.

1. If at the leaf level, return rt
1. Else, find the index of the index record whose MBR has to expand the least when expanding to include ir->MBR.
2. Call ChooseLeaf again, passing the child of the index record you found in the previous step and ir.

I've implemented a parallelized version of this algorithm called ChooseLeafParallel that can work on a variable number of threads. It works by taking all of the index records in a node, splitting them up into n groups of approximate length M/n each (for n threads and M index records per node), and finding the minimum area enlargement for each of those groups of index

records. Each of the threads then writes the local minimum area enlargement and the index of the index record for that enlargement to two shared memory arrays. The minimum enlargement area of the shared memory array is then found by the parent thread once the child threads have joined. In order to make the effect of the speedup more obvious, I've placed a time delay on the constant time operation get_area_increase so that the time by the parallelized algorithm isn't dominated by the creation and joining of threads.

Antonin Guttman's algorithm works in $O(Mlog_M n)$ time, where M is the number of index records per node and n is the total number of nodes. By allowing the search of the optimal node for insertion in parallel time, the O(M) component of the runtime can be sped up to O(M/P) for P processors, changing the overall runtime to $O(\frac{M}{P}log_M n)$. We can see the increase by testing out our algorithm with 1-12 threads on a randomly generated on a 2-level R-Tree with 4096 index records per node. For each number of threads, the time figure given is the average of 25 runs.



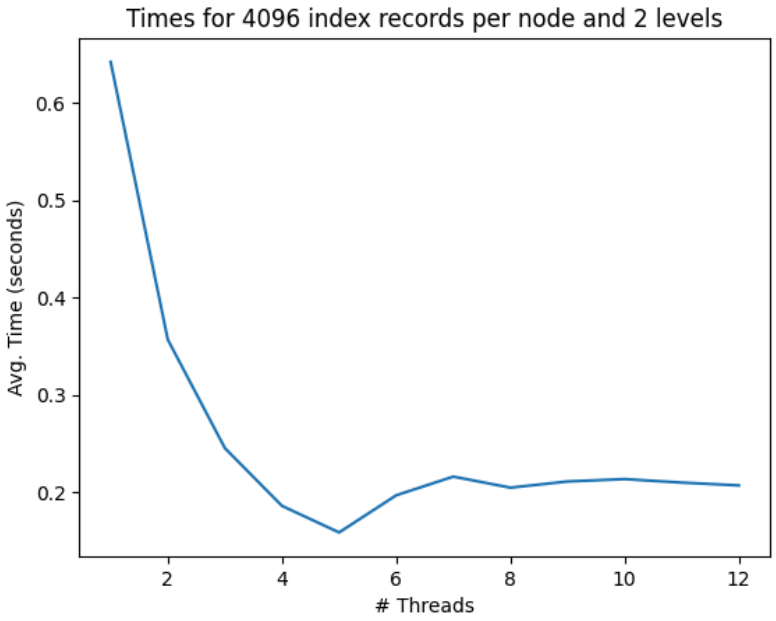
PickSeeds and PickSeedsParallel

Antonin Guttman's paper introducing R-Trees introduced an algorithm called PickSeeds. This is a subroutine used in the insertion of index records into R-Tree nodes which are already full (meaning they contain the maximum number of index records). When such an insertion occurs, the R-Tree node must be split in two in a way that minimizes the area of the two nodes' minimum-bounding rectangles. The PickSeeds routine is used to find the pair of index records in an R-Tree node (including the index record that is about to be inserted) whose merged minimum

bounding rectangle is the most “wasteful.” After these two “seeds” are found, they are each put into separate R-Tree nodes - the original one and the new one created in the process of splitting the original one.

The merging of two MBR’s is considered wasteful if the difference between the MBR containing both of the two original MBR’s and the sum of the areas of each of the two original MBR’s is high. For example, in the illustration below, the two minimum bounding rectangles A and B are considered wasteful when grouped under their overall minimum bounding rectangle C because there is a lot of space in C that does not include A or B.

I implemented a parallelized version of PickSeeds that I ran along with ChooseLeaf for a varying amount of threads. Unlike when testing ChooseLeaf on its own, no time delay was added for each elementary operation because the time complexity of this algorithm was enough to make the effect of parallelization obvious. Seen below are the timings for a few different parameters:



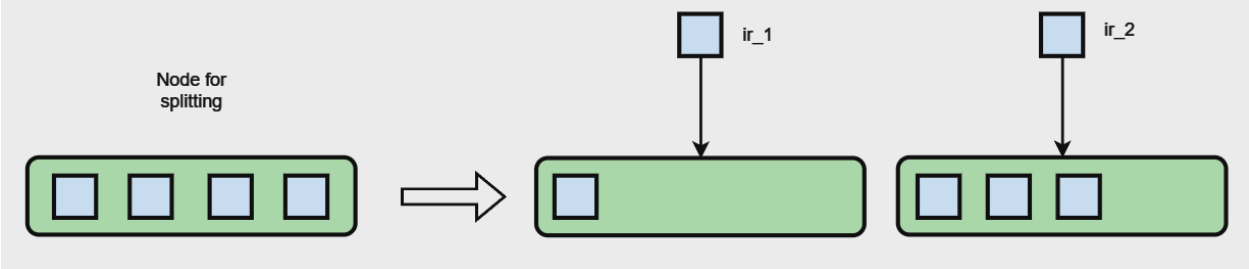
The time complexity of the PickSeeds algorithm is O(M^2) (M=max index records per node), as it has to pick the most wasteful pair out of M index records in an R-Tree node. With parallelization, I was able to bring the cost of this algorithm down to O(M^2)/p. Unlike ChooseLeaf, there is no logarithmic component of this algorithm because it does not include any recursion up or down the R-Tree.

LinearSplit and LinearSplitParallel

After finding the two seed index records for splitting, the rest of the index records in the node you are splitting need to be allocated based on the MBRs in the two seeds. The algorithm LinearSplit performs that function. Unlike the previous algorithms, this algorithm was only loosely based on descriptions by Antonin Guttman. As such, it is worth noting that the LinearSplit algorithm in this paper should not be confused with the one discussed in his paper.

This algorithm takes in an R-Tree node for splitting along with two index records (ir_1 and ir_2) as parameters. Both ir_1 and ir_2 are index records that point to children who will hold the index records previously contained in the node being split. The purpose of this algorithm is so that after it is called, ir_1 and ir_2 can be inserted in the old node's parent.

The effect of LinearSplit can be seen in the graphic below:



The MBRs of ir_1 and ir_2 are set to the first index records in their children, which are the two seeds found by PickSeeds or PickSeedsParallel. The LinearSplit routine works by going through the index records in the node to be split one-by-one and seeing whether ir_1 or ir_2 would require less expansion to accommodate the new index record. It then puts the index record in ir_1's child or ir_2's child depending on the result of that calculation. The parallel version of this algorithm, PickSeedsParallel, works by assigning multiple threads to different subsets of the index records in the node to be split and writing the results of the calculations in a shared memory array. Then, it sequentially iterates through that array and assigns each of the index records in the node to be split to ir_1->child or ir_2->child.

The runtime of this algorithm in its sequential implementation is O(M), and O(M/p) in its parallel implementation, thus its name including the word "Linear". Unlike ChooseLeaf, there is no logarithmic component of this algorithm because it does not include any recursion up or down the R-Tree.

AdjustTree

This algorithm was implemented based on the description by Antonin Guttman in his paper. Insertions of new MBRs in an R-Tree may cause the tree to be temporarily invalid - meaning that MBRs at higher levels do not fully encompass MBRs below them. In order to maintain the tree's validity, each insertion into the R-Tree calls the AdjustTree routine. This routine, which is purely sequential, adjusts the parent index record's MBR of the node in which a new index record is being inserted. After doing so, it recurses upward, performing the same expansion until it finds the root, at which point it returns. The runtime of this algorithm is $log_M(n)$.

Insert and InsertAtNode

The function Insert takes the index record to insert into the R-Tree, the root of the R-Tree, a boolean value to indicate whether it should run parallelizable subroutines (instead of their sequential versions), and the number of threads to run. It starts by getting the leaf node for insertion using ChooseLeaf (or ChooseLeafParallel if indicated). Then, it calls the InsertAtNode

routine which inserts the node at the indicated leaf-level node. After successful insertion, it calls AdjustTree to make sure that the R-Tree is valid. If the indicated leaf-level node is full, it splits it using either LinearSplit or LinearSplitParallel. If the parent of the node for insertion itself is full as well, InsertAtNode recurses upward to the parent node. If the root happens to be full, InsertAtNode creates a new root. As a result, it sets the root to the new root so that future queries or insertions start at the correct root.

An interesting feature of InsertAtNode is that if the leaf level node and its parent both happen to be full, it must choose either ir_1 or ir_2 to pass to InsertAtNode when recursing to the parent. In this case, it always chooses whichever of the two is above the leaf node that contains the newly inserted index record. This way, when InsertAtNode reaches its base case, it will call AdjustTree using an index record that is above the newly inserted record, and thus the validity of the tree will be maintained. This also works because the routine which adds an index record to a node, AddMember, expands the parent index record's MBR upon each insertion to the node.

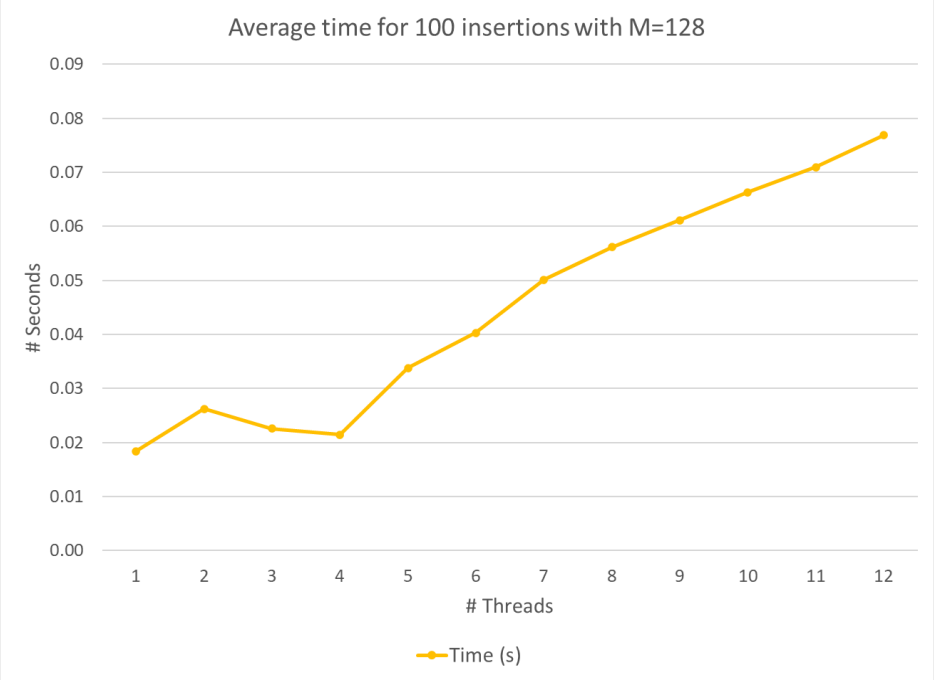
The speed of this algorithm is constrained by PickSeeds, which has an O(M^2) runtime. As such, its overall runtime is O(log_M(n) * (M^2)/p).

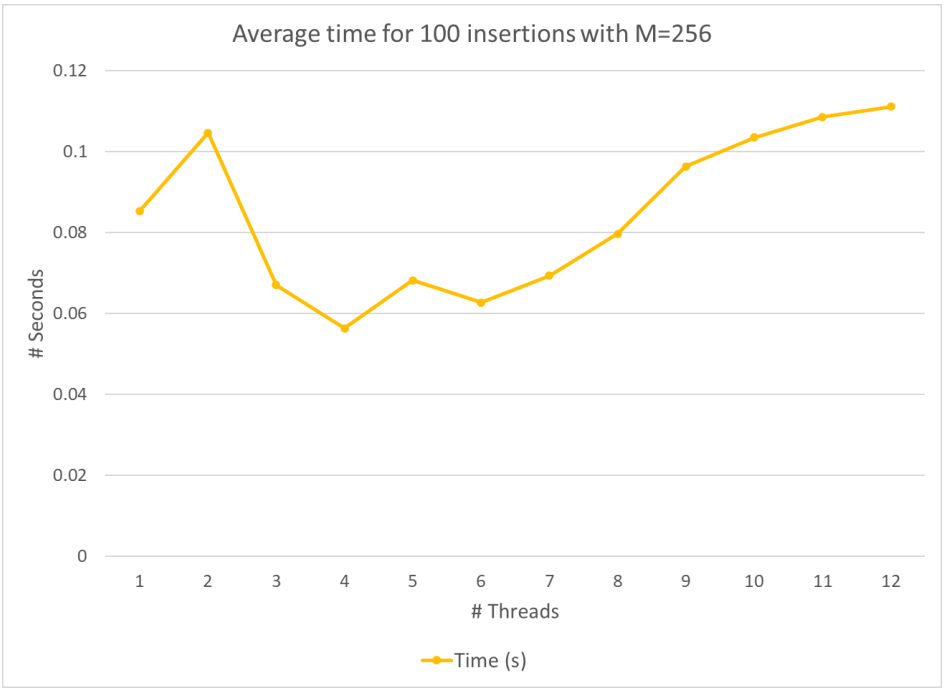
Performance of Parallel Insertions

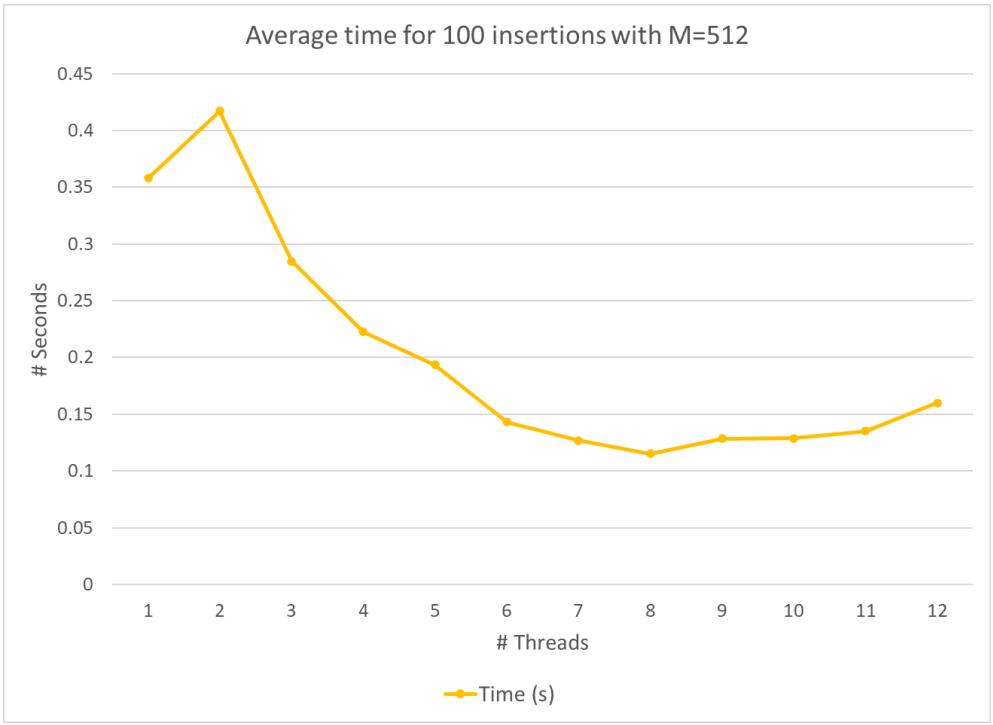
Parallelized insertions are insertions that use the parallelized versions of the ChooseLeaf, PickSeeds, and LinearSplit subroutines. These insertions have proven to be significantly faster on wide R-trees (trees with M >= 512). However, for more narrow trees, they are impractical due to the overhead and context switching that multithreading requires taking too much time.

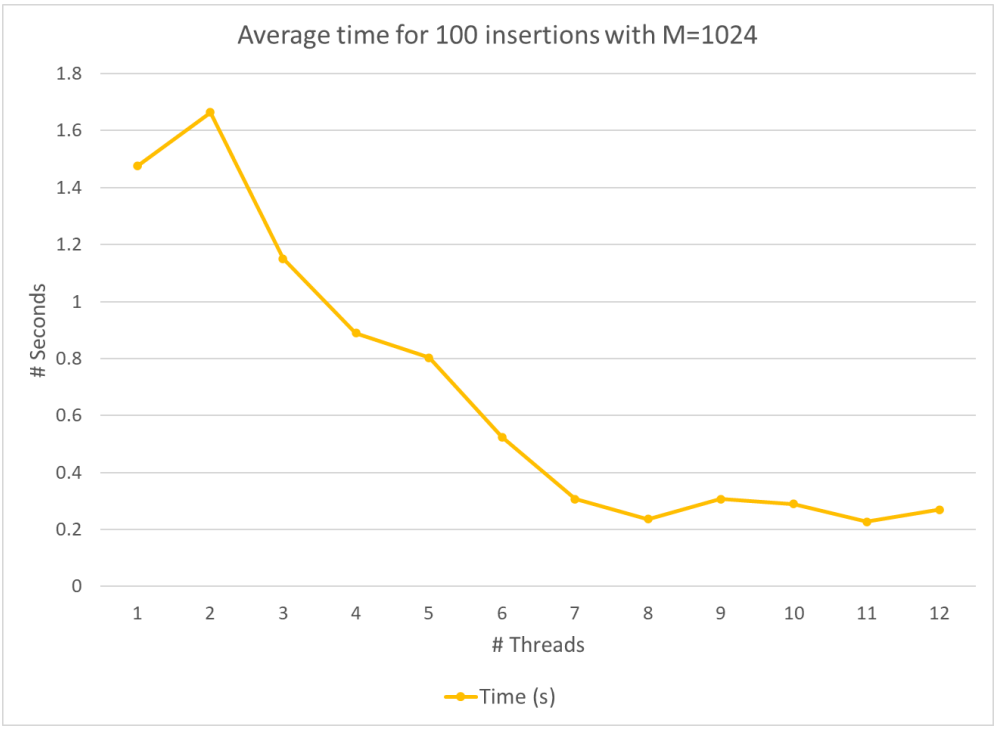
The graphs below show how long 100 insertions took with 100 index records with each index record containing a small and randomly generated MBR. For each value of M tested (128, 256, 512, 1024, 2048, and 4096), the average time taken for 25 instances of sequentially inserting 100 random index records was recorded. Each randomly generated R-Tree had an initial depth of 2 (meaning it had two levels of nodes). Additionally, the randomly generated R-Trees were generated to be full, such that the first insertion was guaranteed to cause a split and the creation of a new parent, and such that later insertions were also very likely to cause splits. The table below shows the speedups for 1 compared to 12 threads for each different tree width.

Tree Width (M)	Speedup for 100 insertions for 1 vs 12 threads
128	0.24
256	0.69
512	2.24
1024	5.47
2048	9.48
4096	6.87

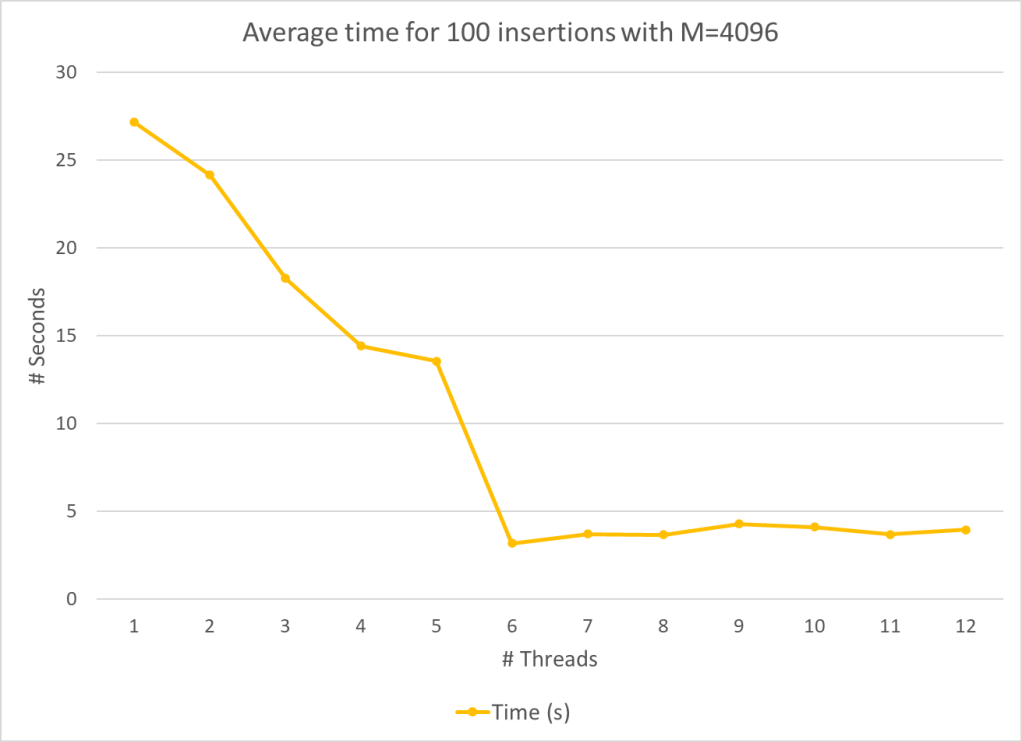



Average time for 100 insertions with M=256
0.12
0.1
0.08
0.06
0.04
0.02
0
Seconds
1
2
3
4
5
6
7
8
9
10
11
12
Threads
Time (s)


Average time for 100 insertions with M=512
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
Seconds
1
2
3
4
5
6
7
8
9
10
11
12
Threads
Time (s)


Average time for 100 insertions with M=1024
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
Seconds
1
2
3
4
5
6
7
8
9
10
11
12
Threads
Time (s)


Average time for 100 insertions with M=2048
7
6
5
4
3
2
1
0
Seconds
1
2
3
4
5
6
7
8
9
10
11
12
Threads
Time (s)



For M=128 and M=256, no significant overall speed gains are made due to the speed savings in parallelization being too small compared to the overhead of creating and managing threads. For M=512 and M=1024, the change from one thread to two threads causes a slight increase in the overall amount of time taken due to context switching. However, for all tests where M>=512, there is a monotonic decrease from 2 threads to 6 threads. After 6 threads, the time starts to level off. This may be due to the multicore environment where these tests were run limiting threads to 6 cores despite there being a total of 12 cores.

Conclusion

The ability to speed up individual insertions into an R-Tree using threads has been shown to be feasible for wider (M>=512) trees. However, some programmers may deem parallelizing individual insertions impractical for a couple of reasons. The main speedup of insertion is due to the parallelization of the routine with quadratic runtime: PickSeeds. All other subroutines of insertion have linear or better runtime. Due to its quadratic runtime, parallelizing PickSeeds will have the greatest effect when having to perform a split on a wide R-Tree. However, wide R-Trees

are also the R-Trees where splits will happen the least. The decision to parallelize individual insertions must be made considering these two factors and the use case of the R-Tree at hand.

An implementation of this type of R-Tree can be found at my GitHub:
https://github.com/liamr333/R-Tree-Parallelization/tree/main

References

Guttman, Antonin. "R-trees: A dynamic index structure for spatial searching." *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 1984.

Luo, Lijuan, Martin DF Wong, and Lance Leong. "Parallel implementation of R-trees on the GPU." *17th Asia and South Pacific Design Automation Conference*. IEEE, 2012.

Prasad, Sushil K., et al. "GPU-based Parallel R-tree Construction and Querying." *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015.