Efficient implementations of range trees

Radu Berinde

Abstract

Range-trees are a data structure for solving the bi-dimensional orthogonal range query problem: given a set of points in the plane, efficiently find the points which lie inside any given rectangle. This paper gives a complete description of the data structure, and investigates what is the most efficient way to implement it. Methods to improve the performance of the data structure when ran on a multi-core machine are also investigated. The performance of several versions of the structure is analyzed on a practical data set.

Introduction

Given a set of points, the d-dimensional orthogonal range searching problem is the problem of finding the points inside a specified d-dimensional hypercube. For $d = 2$, this equates to finding the points that lie inside a given rectangle. Of interest is the repetitive version of the problem, in which many range queries must be resolved.

We restrict our attention to the bi-dimensional, static version of the problem, in which all the points lie in the plane and are known a priori. This version of the problem is widely useful to solve in practice; applications of this problem arise in database searches, geography, statistics, design automation ([1]). In this static framework, we have the opportunity to pre-process the data to build a data structure that allows efficient resolution of range queries.

Range-trees are a data structure for this problem that is optimal in terms of the query time: a query runs in $O(\log n + k)$ time, where n is the number of points and k is the answer size (the number of points inside the query rectangle). We have seen a high-level presentation of range-trees in class; in this project, I attempt to work out the details of efficient implementations of this data structure. I also investigate improvements in the preprocessing and query running times using parallelization on a dual-core platform.

I will first present a precise description of range-trees and the refinement that leads to optimal query time. I will then describe the details of the most efficient implementations I was able to achieve - along with mentions of some of the attempts that have failed to improve the data structure. I will then present modifications that allow improved performance on a dual-core platform. Finally, I will present the comparative performance of several chosen implementation versions.

1 Range Trees

Range trees are the extension of a unidimensional data structure called the segment tree. I will first describe the segment tree, and then discuss how to extend it to the multi-dimensional case. I acquired much of the information in these descriptions from Shamos and Preparata's book ([1]).

1.1 Segment Trees

The segment tree was introduced by J.L. Bentley [2]. It is a data structure for intervals on the real line whose extremes belong to a fixed set of N abscissae. For simplicity, we will assume that this set of abscissae is formed by the N integers between 1 and N. It will become obvious how to extend it for arbitrary sets of coordinates - in theory, this is not even necessary, as we can always "normalize" coordinates of incoming queries using binary searches.

The segment tree is a binary tree, in which each node is responsible for a range of coordinates. In particular, the root of the tree is responsible for the entire $[1, N]$ range, whereas a leaf is responsible for an elementary interval of the form $[x, x]$. The tree is defined recursively: if a node is responsible for the range $[l, r]$, then its left child is responsible for the range $\left[l, \lfloor\frac{l+r}{2}\rfloor\right]$, while its right child is responsible for the range $\left[\lfloor\frac{l+r}{2}\rfloor + 1, r\right]$. The tree is balanced and has height $\lceil\log_2 N\rceil$.

The useful feature of this tree is that any given subinterval $[a, b] \subset [1, N]$ can be broken up in $O(\log N)$ intervals for which nodes in the tree are responsible. This is achieved by a simple recursive search down the tree, stopping whenever the interval of the current node is entirely inside $[a, b]$. To see why the number of nodes in which the search finishes is small, imagine we color in red all the terminal nodes of the search - the nodes which make up the interval $[a, b]$. Then, on each level of the tree, the red nodes will form a contiguous subset of the nodes on that level (imagining the tree drawn left-to-right). Then, if there are more than two red nodes, two of the red nodes will inevitably share the same parent, which is a contradiction because the search would have finished at the parent (if both children are completely contained in $[a, b]$, so is the parent). Thus, there are at most 2 terminal search nodes on each level, and thus the search takes $O(\log N)$ time.

Note that if $N = 2^k$ for some integer k, the segment tree is a complete binary tree of depth k. The tree breaks up the interval $[0, 2^k)$ into what are called *dyadic intervals*: intervals of the form $[j2^i, (j+1)2^i)$, for $0 \leq i \leq k$ and $0 \leq j < 2^{k-i}$.

This data structure is very versatile: many update/query problems can be solved by adding data in the nodes of this tree resulting in updates and query that take time $O(\log n)$ (plus the size of the query output where relevant). A few examples are:

- Maintain a vector A_i for $1 \leq N$.
  UPDATE(a, b, x): $A_i \leftarrow A_i + x$, for all $a \leq i \leq b$.
  QUERY(a, b): $\sum_{i=a}^{b} A_i$.
- Same as above, except update is: $A_i \leftarrow x$ for $a \leq i \leq b$.
- Maintain a set of intervals over $[1, N]$.
  INSERT(a, b): insert interval $[a, b]$.
  DELETE(a, b): delete interval $[a, b]$.
  QUERY(x): the intervals (or the number of intervals) spanned by coordinate x.
- Maintain a set of points over $[1, N]$.
  INSERT(p, p_x): insert point p at coordinate p_x.
  DELETE(p): delete point p
  QUERY(a, b): the points (or the number of points) inside interval $[a, b]$.

Note that the last problem is the orthogonal range problem in one dimension. Also observe that to allow arbitrary fixed sets of coordinates, we can simply pre-sort the coordinates inside an array A_i of size N; then a node in the tree will correspond to interval $[A_l, A_r]$ rather than $[l, r]$ above.

1.2 Extending segment trees to range trees

To build a data structure that allows 2D range searching, we use the segment tree to reduce the 2D problem to a number of instances of 1D problems. More precisely, we build a segment tree that ignores all the y coordinates of the points; inside each node v of the segment tree with corresponding interval $[a_v, b_v]$, we build a second-level structure $Y(v)$ which contains all the points with the x coordinate inside this horizontal interval $[a_n, b_n]$. We can think of each node in the segment tree as corresponding to a vertical "slice" of the plane. $Y(v)$ stores all the points in this slice; it ignores the x coordinates of the points and allows binary searching of y coordinates. In general, $Y(v)$ can be any such data structure; for simplicity (as well as efficiency), we simply use a static array in which the points are sorted by increasing y coordinate. The resulting tree is called a range-tree.

Assuming we have built such a structure, we can solve queries of the form $[x_1, x_2] \times [y_1, y_2]$ in the following way: perform a search for $[x_1, x_2]$ on the main level tree; for each node v of the $O(\log n)$ nodes that make up $[x_1, x_2]$, perform two binary searches for y_1 and y_2 inside $Y(v)$ to find the (contiguous) sequence of points contained inside $[y_1, y_2]$. Report or count all these points. The query involves $O(\log n)$ binary searches, as well as processing each point in the answer. The query time is thus $O(\log^2 n + k)$, where k is the number of returned points. Note that if we only want to count the number of points inside the query rectangle, the query only takes $O(\log^2 n)$ time.

We can build this structure easily by noticing that for any node v in the main tree, with left and right children v_l and v_r, the points which should be stored in $Y(v)$ are exactly the points stored in $Y(v_l)$ and $Y(v_r)$. If $Y(v_l)$ and $T(v_r)$ have already been built, we can build $Y(v)$ by merging the two arrays. The result is an algorithm which is very similar to merge-sort, with the difference that the partial sorted sub-arrays are stored inside nodes of the segment tree. This takes $O(n \log n)$ time. The space requirement is also $O(n \log n)$ because for every point, there is a copy of that point in each of the $\Theta(\log n)$ nodes on the path from the root to the leaf corresponding to the point's x coordinate.

Note that we assume that each point has a unique x coordinate and thus the points in the secondary structures are spread out evenly. In practice, even if some of the points share the same x coordinates, the unbalance is not significant unless the data set is very unnatural. Even so, we can apply infinitesimal random shifts to the points to enforce the condition.

1.3 The Willard-Lueker refinement

There is some redundancy in the work we do on a query as described above; we do $O(\log n)$ independent binary searches as if the secondary data structures were completely unrelated. Willard [3] and Lueker [4] independently discovered how to improve the query time using the relationship between the secondary data structures.

Let us restrict our attention to the bottom line of the query rectangle y_1. Let $LowerBound(y_1, Y(v))$ for some node v be the element p of $Y(v)$ with minimal ordinate p_y such that $p_y \geq y_1$. Assuming that for a node v we know where y_1 would "land" inside $Y(v)$ - more precisely, we know *Lower-Bound*$(y_1, Y(v))$ - can we quickly find where y_1 would land inside $Y(v')$ for a child v' of v? The key observation is that $Y(v') \subset Y(v)$; then if $p = \textit{LowerBound}(y_1, Y(v))$,

$$\textit{LowerBound}(y_1, Y(v')) = \textit{LowerBound}(p_y, Y(v'))$$

This is the case because either both expressions evaluate to p or if the left side evaluates to some other p', we know that $y_1 \leq p_y \leq p'_y$ from the definition of point p, which implies that the right side also evaluates to p'.

The idea is then that for each element of a node array, we only need to know where that element would land in each of the node's children arrays. We can simply pre-compute and store this information inside the arrays; for each element p in an array $Y(v)$, we also maintain two pointers $\textit{lbridge}(p)$ and $\textit{rbridge}(p)$, so that $\textit{lbridge}(p)$ points to $\textit{LowerBound}(p_y, Y(v_l))$ and $\textit{rbridge}(p)$ points to $\textit{LowerBound}(p_y, Y(v_r))$.

Given these bridge pointers, we can solve a query in the following way: first, do two binary searches on the array stored in the root node r to find (the positions of) $p_1 = \textit{LowerBound}(y_1, Y(r))$ and $p_2 = \textit{LowerBound}(y_2 + \varepsilon, Y(r))$, for some sufficiently small value of ε. Then proceed with the segment tree search procedure: whenever the search goes down a node v's (left or right) child w, access the (left or right) bridge pointers for p_1 and p_2 to find the new points/positions in the child node's array (the new values for p_1 and p_2). This takes constant time. Whenever the search reaches a terminal node, report (or count) the points in the array between the current p_1 and p_2 positions (including p_1 but excluding p_2). The query runs in $O(\log n + k)$ time (or $O(\log n)$ if we only count points).

The storage space is increased by a constant factor. Given our merge-based building procedure, the bridge pointers are easy to compute from the merging step. Suppose we merge $Y(v_l)$ and $Y(v_r)$ to obtain $Y(v)$; whenever we add a merged element to $Y(v)$, the two bridge pointers correspond to the current positions inside $Y(v_l)$ and $Y(v_r)$ (one bridge will point to the element currently being copied to $Y(v)$, and the other bridge will point to the first yet-not-merged element in the other array).

We thus obtain an $O(n \log n)$ preprocessing time and storage space data structure that solves 2D range queries in $O(\log n + k)$ time. Note that these results can be extended to arbitrary dimension d by recursively applying the same dimensionality reduction idea: for example, the nodes of a 3D range-tree stores 2D range-trees as second-level data structures. This results in $O(n \log^{d-1} n)$ storage space and preprocessing time and $O(\log^{d-1} n + k)$ query time.

2 Implementation details

I will first discuss certain aspects that are relevant in practice; then I will move on to describing the exact implementations.

2.1 Notes

The implementations have been developed on an AMD Athlon 64 X2 CPU with 2Gb of DDR2 RAM, under Windows XP x64, using Microsoft Visual C++ 2003; the boost::threads library was used for the multi-threading functionality. In hindsight, this was not the perfect choice as the MSVC compiler proved inconsistent in optimizing code and sometimes required a lot of fine tuning.

2.1.1 Output sensitivity

When implementing and testing these data structures, the $O(k)$ factor in the query time is dominant unless we restrict to small queries. For this reason, when measuring the performance of any implementation, the reporting of the results is disabled, and the queries are restricted to counting the points inside the query rectangle. But the capability to also report the points without any overhead is retained at all times; in practice, this framework can be useful in situations where we do want to report the resulting points, but only if they are not too many (in which case we would still want the total count). Or similarly, we might want the output, but only up to some fixed number of points. This is important because there exist data structures that solve the counting problem more efficiently while giving up the ability to report the points (see [10]).

2.1.2 Data

Tests used two types of data: random points uniformly distributed in some range of coordinates, as well as real geographical data, derived from the TIGER database. TIGER/Line maintains cartographic information and includes complete coverage of United States. Using this data, I have compiled test sets using points corresponding to street intersections in California. The database contains about 4.5 million such intersections; subsets of these points are obtained by restricting the points to a certain smaller longitude range. The points are given as longitude and latitude, with 6 digits of precision, and are represented as fixed-point integers: a point's x coordinate is equal to its longitude multiplied by 10^6, the y coordinate is the latitude multiplied by 10^6. Note that this results in a precision of about 4 inches. The data is useful as it is the kind of data that might be used in mapping applications; for example if we have many interest points in cities (hotels, bars, banks, etc.), we expect their distribution to resemble the distribution of street intersections - the more dense the street intersections, the more crowded an urban area is.

For random data, the query rectangles were formed by randomly choosing two corners of the rectangle in the range of coordinates spanned by the points. For real data, a more elaborate scheme was used: a set of sizes (in miles) was selected, as well as a plausible probability distribution for these sizes; a random query is built by randomly choosing the size of the query rectangles according to the distribution; then, a random point is uniformly sampled from all the points, and the rectangle is randomly positioned such that it includes this point. The idea behind this is that in a mapping application, we expect proportionally more queries in crowded areas than in uninhabited areas; thus, the more crowded an area, the more points are in that area, and the more likely a query is to hit that area.

It is important to note that in practice, range-trees proved insensitive to the distribution of input points. They are somewhat sensitive to the distribution of queries relative to the points, in that smaller query rectangles result in slightly better query times (explained by the fact that for small x intervals, the main tree search touches a smaller number of nodes); for this reason, random queries in random data, which implies very big rectangles on average, result in about 10% to 30% worse performance than the more natural mapping queries in the geographical data. Note that in geographical data, generating queries by uniformly choosing two corners actually results in better performance than the more natural queries, because many such query rectangles, despite being large, span sparsely inhabited areas. In any case (and most importantly), the choice of data and queries did not significantly influence the relative performance of any two compared range-tree implementations.

2.1.3 On space-performance trade-offs

Many times a trade-off between space and query performance has to be made. One can imagine many applications where the space usage of the data structure is very important (e.g. the data structure must fit in one computer's memory); but there are also situations in which the space usage is not critical. For example, consider a multi-server environment where we have N machines and want to use these machines to answer as many queries as possible; suppose the space requirement of the data structure is so that the dataset must be split into M (with $M \ll N$) pieces in order for each piece to fit on one machine. In the worst case, if one machine is able to answer q queries per second (on average), we can answer $q\frac{N}{M}$ queries per second - and trading off space for query time does not seem to make sense. But if we split our data wisely, for example each of the M pieces correspond to disjoint "slices" or regions of the plane, then we might be able to assume (depending on the application) that most queries will only need to go to one (or a small) number of machines. Then the average query performance of the system is proportional to qN, and we might afford to trade large amounts of space for even small increases of query time, as M is not important anymore.

With these arguments in mind, I will present a few chosen implementations with different space-to-query-performance ratios. More precisely, I will describe a middle-ground implementation which implements the Willard-Lueker refinement in its described form; I will also describe a solution with a slightly better query time, but worse space requirement, as well as a small-space version which uses the $O(\log^2 n)$ data structure to cut space usage.

2.1.4 The memory wall

While implementing these data structures, it became obvious that the bottleneck in query performance is not processing time, but memory bandwidth and latency; in this particular application, we have hit the memory wall. The memory wall refers to the growing gap between memory and processors: statistics show that from 1986 to 2000, the CPU speed improved at an annual rate of about 60%, while memory improved at only 10% ([6]). The disparity is thus huge today compared to what it was 20 years ago. While working on the project, I became gradually more aware of this problem as optimizations that should normally speed up programs were not helping.

The memory wall is the reason why the arguments and ideas presented in this paper revolve more around efficient cache usage rather than total number of operations. Note that it is probably worthwhile to move away from range-trees and to investigate the range searching problem in a different model in order to build cache-oblivious or cache-aware data structures; however, the scope of this project will remain limited to range-trees.

2.2 The Willard-Lueker implementation

The nodes of the segment tree contain:

1. the information necessary to deduce the intervals of the child nodes (two “middle” x coordinates).
2. the number of points in the secondary data structure
3. a pointer to an array storing the points
4. a pointer to an array storing Willard-Lueker bridge pointers (indices)

The tree does not need child pointers, as nodes are stored inside an array in BFS-order (like heaps), although as we shall see later, this hardly makes a difference in query performance (or space usage). The tree building is performed in the following way: first, all points are sorted by increasing x coordinate (as per the observation about segment trees and arbitrary abscissae sets in section 1.1). Let P_i be this sorted array, so that P_i is the i^{th} point in the plane from left to right. A recursive BuildNode function is used to create the actual tree. The function takes two indices l, r in the sorted points array and creates a node responsible for all points P_l to P_r. If all these points share the same coordinates (or $l = r$), a leaf is created - the secondary arrays are built and re-sorted by the y coordinate. Otherwise, the points are split at a midpoint $m = \lfloor \frac{l+r}{2} \rfloor$, and the two children are built recursively for points in the ranges $[l, m]$ and $[m+1, r]$, respectively. The two child arrays are merged into the node's secondary arrays, process which also computes the bridge pointers.

Note that we can very cheaply obtain the points in sorted order of increasing y coordinate as a side-effect from the build procedure. We do this to create a top-level array which only stores the y coordinates of the points in increasing order. A query $[x1, x2] \times [y1, y2]$ proceeds as following: first, two binary searches are used to obtain the positioning of $y1$ and $y2$ in the top-level array of sorted y coordinates - i.e. we find the indices of $LowerBound(y_1, Y(root))$ and $LowerBound(y_2, Y(root))$. Using these left and right indices we can start the range tree search; inside a node, we perform two lookups in the bridge array using these indices, obtaining the corresponding indices for the child nodes. In terminal nodes (whose x interval is completely inside $[x1, x2]$), we simply report/count the nodes between the two indices.

2.2.1 Improvements

The figures I will mention are obtained from datasets of size on the order of 1 to 4 millions of points. The figures are similar for both the random and the geographical data.

The first important fact is that less than 15% of the query time is spent doing the preliminary binary searches [1]. A simple improvement tried was to perform both binary searches at the same

[1]An interesting fact is the way we measure this: if we simply disable the range tree search part of the program, the program (for large enough dataset and number of queries), runs in less than 5% of the time it usually does. However,

time, in hope that both binary search paths share a common prefix, but this resulted in no performance gain (probably because the cache usage pattern is not improved). We could improve the binary searches by replacing the arrays with a more cache-friendly data structure, like the CSS-trees in [7]. I did not implement this, because even if we manage to achieve a 100% speedup in the binary searches (which is about what [7] claims for datasets of a few million items), we would still only get 7% speedup in the total query time.

Thus most of the time is spent in the range-tree search. I have tried many different ways to change the structure of the main tree to speed up this search, most of which have failed. I enumerate a successful attempt along with some of the more notable failed attempts below; I then explain why most methods for optimizing the range tree do not work.

Fat leaves

A simple worthwhile optimization is to limit the height of the tree: the lower levels are used to separate a small number of points; it is cheaper to pack more nodes into leaves in order to decrease the depth of the tree. During searches, if we reach a leaf which is not completely contained in the query interval, we know the range of points in the arrays which are between y_1 and y_2, but we do not know which of them are between $x1$ and $x2$, so we have to check them. We can place a packed array in leaves containing the x coordinates of the points in the leaf arrays, so that performing the x-coordinate checks is very cheap (cache-wise). Surprisingly, in practice this works well for packing as much as 8 extra levels (meaning that we pack 256 points in every leaf!); the reason is probably that in practice we don't necessarily reach many leaves, and when we do the subinterval is considerably less than 256. We also get space benefits, since we are omitting many copies of the points (together with their bridge arrays).

This improvement results in a small (5-10%) improvement in the average query time, but also a considerable improvement in space usage (30-40%) and in pre-processing time (50-60%). These figures are for datasets of 2 to 4 million points. Note though that as the dataset size increases, these improvement factors diminish - we save $O(1)$ node operations and $O(n)$ space, out of totals of $O(\log n)$ and $O(n \log n)$, respectively.

Packed nodes

Most of the data in the range-tree nodes is not used when processing non-terminal nodes of the search (e.g. the point array, the number of points). A failed attempt to improve the query time was to ship off all this data away from the nodes; the reasoning was that more compact nodes would allow better cache-coherency (as more data would fit in the same number of cache lines). This attempt actually increased the query time because of the overhead (indicating that the cache misses of node accesses are not really the issue).

van Emde Boas layout

Another attempt was to restructure the range-tree using a van Emde Boas layout (see [8]); a tree of height h is split at level $h/2$, resulting in a tree for the top part and $2^{h/2}$ trees in the bottom part. These trees are laid out recursively, first the top tree, then the bottom trees in left-to-right

when measured with a code profiler, we get a number closer to 15%. This shows how important the L2 cache usage pattern is: we get 10% slowdown in the binary searches when the other accesses cause data relating to these arrays to be evicted from the cache.

order. I have used the idea in [8] to create a function `normToBoas` which converts indices in the BFS-ordered layout into indices in the van Emde Boas layout. More precisely, the function takes an index i; in the BFS-ordered (heap) linearization of a binary tree, i corresponds to a certain node in the binary tree (e.g. $i = 1$ is the root of the tree, $i = 3$ is the right child of the root, etc.). The function returns an index i' which is the position of where this node should go if the tree is linearized using the van Emde Boas layout (i.e. how many nodes are laid out before this node in the v.E.B. layout). The function is a very good idea as it applies the layout transparently: the algorithm works as if the tree is stored in BFS order, except that whenever we read a node, we use the `normToBoas` function to "redirect" the access. The function performs O(log log n) simple arithmetic operations. The hope is that the number of cache lines touched in a tree search is significantly decreased, and the benefit outweighs the extra computing cost of the function.

Unfortunately, when using this function we actually get a significant slowdown in the query times, again suggesting that the cache-coherency of the range tree is not the problem. This is surprising, as surely the memory must be the bottleneck in a data structure in which everything seems "precomputed" in some sense. The explanation of this is that the L2 cache does a good job with the range-tree nodes anyway: the L2 cache is very fast (on modern processors it usually runs at full CPU speed), and is pretty large (1-4Mb). Since the nodes in the top levels of the range tree are frequently visited, it is reasonable to expect that they are always in the L2 cache. The L2 cache is big enough to hold quite a few levels of the range-tree; also, in the range-tree searches we can expect part of the terminal nodes to be on the higher levels of the tree, and thus only a fraction of the $O(\log n)$ visited nodes result in cache misses when accessing the range-tree node. Thus, improving the cache-efficiency of the range-tree does not really help the query times, as most accessed node data is already readily accessible. The same argument explains why the preliminary binary searches are so fast: the middle points chosen by the binary searches also form an implicit tree, and we can expect the top part of this tree to be readily available in the L2 cache.

This hypothesis was confirmed using a code profiling program, AMD CodeAnalyst, which is capable of detecting L2 cache miss rates. Profiling showed that most of the query time is spent *when reading the bridge pointers*, due to L2 cache misses. This makes a lot of sense, since the bridge pointers form the bulk of the space used; the accesses are non-local, because we access entries for both the query rectangle bottom y_1 and top y_2 which are far away in the arrays, and then we move to a new node for which we do a similar access in an entirely different array.

2.3 A slightly faster version

It is clear now that our efforts should go towards improving the cache-coherency of the bridge pointer accesses. Because y_1 and y_2 change with each query, it is hard to see how we could arrange our structure so that at each level, the bottom (y_1) and top (y_2) bridge pointers are somehow close to each other. So let us restrict our attention to only one of the two paths of jumps through bridge pointers. The bridges define an acyclic graph between the elements of the arrays; we observe that once we have accessed an element e, the next accessed element (if the search continues down that path) must be an element e' towards which e has a bridge. Unfortunately, the bridges do not form a tree structure, as we have at least two bridges pointing to any element; it is very hard to design a cache-friendly layout for this graph.

However, we can implement an improvement based on a particular case of the observation above. Let the bridge array (for a node v) be `v.bridges`, so that the left bridge for p is `v.bridges(p).left` and the right bridge is `v.bridges(p).right`. Let v_l and v_r be the left and right child nodes of node v. We know that after accessing some `v.bridges(p)`, the next access is (if the search continues down the tree) either $\mathtt{v}_l$`.bridges(v.bridges(p).left)` or $\mathtt{v}_r$`.bridges(v.bridges(p).right)`. The idea is then to trade off space and place copies of these two entries inside `v.bridges(p)`, so that two levels of the bridge pointers are accessible directly. When we are searching down the tree, we then only need to access the entry when we are at an odd level; we pass the next level bridge indices to the recursive search function so that at even levels they are already available.

Observe that we can then drop all arrays at even levels of the tree. We can build this data structure in the same way as before; at odd levels we simply do a traversal of the two child arrays and copy the entries to the current array, and then we deallocate the child arrays to free up the memory. The array entries are three times bigger (they store three entries instead of one), but we only store them at odd levels of the tree. The space usage should then increase by 50%. In practice, the space usage increase is only about 35%, because of the fat leaves optimization.

One would expect that with this optimization, the number of cache misses would be roughly halved; in practice, this does not happen. The speedup in query time is about 18%. It is probable that the extra data stored results in a less efficient use of the L2 caches; there is also some extra overhead in the search function, as many indices must now be passed around recursively.

2.4 A compact version

As a different space-query time trade-off, I have also implemented the slower $O(\log^2 n)$ algorithm in order to reduce space usage. In practice, this data structure is only a small number of times slower than the $O(\log n)$ version. The details are simple: every node has an array of pointers to points; the search procedure performs two binary searches in each terminal node array. Note the extra level of indirection which slows down the query, but saves significant amounts of space. Fat leaves are used in this version as well. The results of this implementation can be seen in the performance figures in section 4.

3 Improvements for a dual-core platform

The platform available to me for this project was based on an Athlon 64 X2 5200+, 2.6Ghz, dual-core CPU; unfortunately, I had no access to a machine with more cores. I attempted to use multi-threading to take advantage of the two CPUs and improve build and query times of the structures. The attempt was successful with respect to the preprocessing time. Query throughput can also be increased by processing multiple queries in parallel. However, it turned out to be practically impossible to parallelize the work done by a single query in order to reduce the latency of the query. I will show some artificial tests designed to give a better understanding of the platform, which support the conclusion that the query latencies cannot be improved. I will then shortly describe how parallelizing of the pre-processing step is implemented. Experimental results of the described implementations can be seen in section 4.

3.1 An investigation of the dual-core platform

The experiments in this section are designed to show how the dual-core platform behaves when both cores are running memory-intensive applications. The setup is the following: we create two huge arrays A and B of size N containing 32-bit integers; I chose $N = 2^{26}$, so that the size of one such array 256Mb. We attempt to compute the sum of these integers.

We choose two methods of processing an array: one is by cache-friendly linear access, in which the elements are accessed in order; the other is random access, in which elements are accessed in a "random" order: the order is given by the function $f(x) = (x + k) \bmod N$ with $k = 1137341$ a large odd number, so that the order is $x, f(x), f(f(x)), \ldots$ for some random starting x; note that each element is touched exactly once.

Each method is implemented in serial mode, where first A is processed, and then B is processed, as well as in parallel mode, in which both arrays are processed at the same time, in different threads. The results of this experiment are shown in the table below, as running times in seconds:

Mode \ Method	Linear	Random
Serial	0.22s	9.14s
Parallel	0.12s	5.63s

The experiments are promising: even though the two cores share the same memory, the program scales well: for linear access we get a 45% reduction in the running time, and for random access we get a 40% reduction.

This data does not explain the[2] inability of reducing the query times using parallelization; however, we have not taken into account the data exchange between the two cores when we are trying to move some computation to a different core (which is what must be done during a query, which initially starts running on a single core). We implement the parallel method in a more complicated way: we split the N accesses in slices of size K. For each piece, the main thread passes the information about the current slice to a worker thread. The worker thread busy-waits on a variable until the main thread changes it; the thread then starts doing the K computations on array B. After setting this up, the main thread does the K computations for the current slice on array A. It then busy-waits on another variable (if necessary) until the worker thread changes it to signal its work is done.

This implementation is meant to simulate what happens when we parallelize a query: a main thread decides what computation needs to be done by another thread, and must somehow communicate with this thread to set up this computation. We have used busy-wait because it is the method with minimal overhead, faster than using a mutex or barrier. Note that waiting and yielding the current thread timeslice inside the loop was also tried with identical results. This experiment was created to show the overhead of passing data between CPU cores; by varying K, we can get an idea of how small the computation can be before it is cheaper to just run it on a single core. We show how the running time varies with K for the linear and the random access methods. We first show some running times for the linear access:

[2] my

K	128	256	512	1024
time	0.30s	0.22s	0.19s	0.17s

Notice that for $K \leq 256$, the overhead is so big that this version runs slower than the non-parallelized version! It takes K at least on the order of thousands for this version to get close to the simple parallel version. The overhead is thus significant - comparable to 256 cache-friendly memory accesses. For the random access implementation, we get the following results:

K	8	16	32	64
time	9.11s	7.95s	6.92s	6.16s

From these results, we note that for $K = 8$, we get the running time of the non-parallelized version, and thus the overhead is comparable to 8 random memory accesses - which are, with very high probability, L2 cache misses.

It is clear now that the overhead of setting up processing on another core cannot be ignored when the amount of processing is small. Given that our range queries already run very fast (on the order of 5 to 10 microseconds), even if there was a way to perfectly balance the workload among two cores, the scheduling overhead would still consume a major part of the running time and the improvement in the latency would be very small. In this case query throughput would be close to the single-thread throughput, which is considerably worse than the throughput obtained by running queries in parallel in two query threads.

3.2 Improving pre-processing time

As we have mentioned, the build algorithm is very similar to mergesort; it is not surprising that it can be easily parallelized. First, we need to sort the points. We can use a specialized algorithm, or for a small number of cores we can simply break the data into pieces, sort the pieces (on different cores) and merge the results. For two cores, we break up into two pieces; in practice, this results in a 42% reduction in running time. The rest (and most) of the pre-processing is spent inside the recursive build function; a very practical and efficient way to parallelize this function is to run the two recursive calls in different threads. We only need to do this at the top few levels of the recursion (for dual-core, the first level is sufficient). This leads to an improvement that varies with how complex the merge operation is. Relevant results are shown in the next section.

4 Experimental results

The performance of the three described implementations is shown below; the space-efficient simple range-tree version is reffered to as RT, the regular implementation of the Willard-Lueker improvement is reffered to as WL, and the slightly faster but less space efficient version of Willard-Lueker is reffered to as WL2. Measurements are taken on the geographic data set restricted (as a vertical slice) to 1, 2, and 4 million points. Queries are generated as described in 2.1.2.

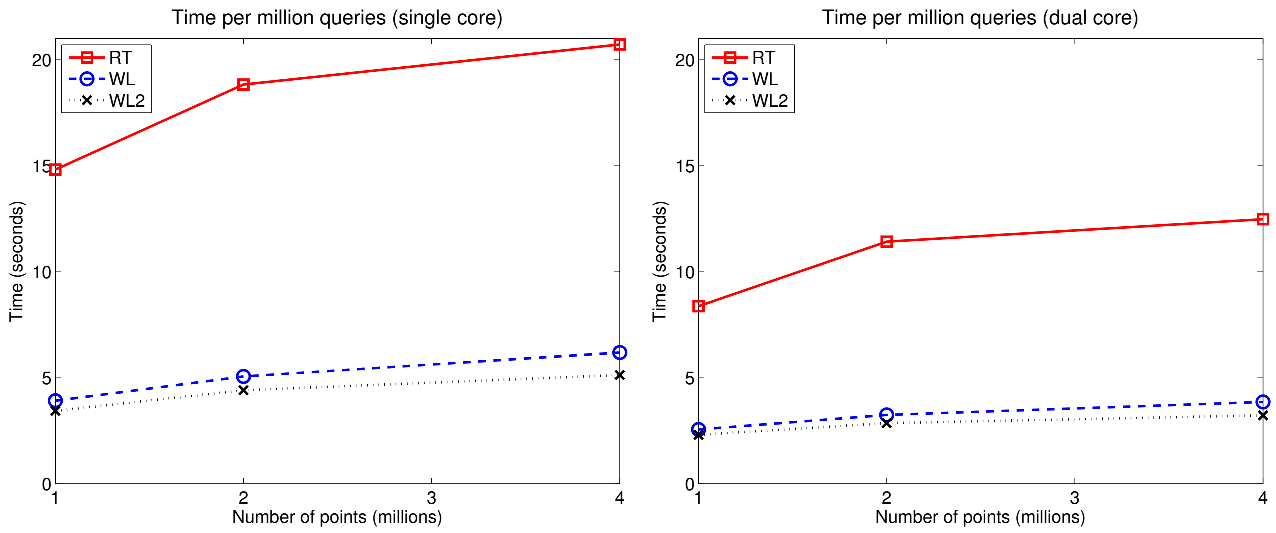

Figure 1: The query performance - shown as the time necessary to resolve 10^6 queries.

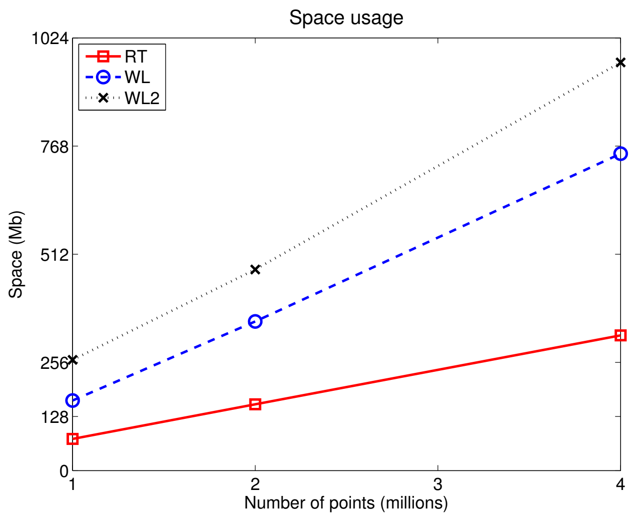

Figure 2: The space occupied by the data structures.

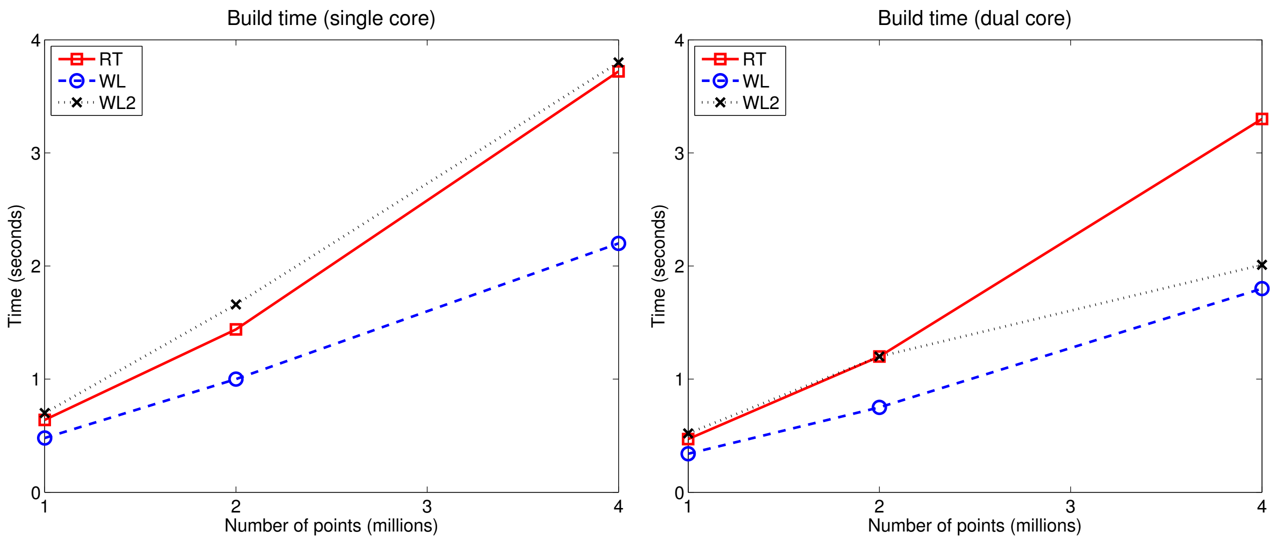

Figure 3: The time needed to build the data structures.

Conclusions

The experimental results show that the range-tree is capable of solving 10^5 to 10^6 queries per second on mid-range hardware. It is very likely that in most applications, the range-tree searching routine will not be the bottleneck; it is more likely that a lot of time is spent in transferring the actual output and/or in generating/gathering queries.

If range-searching is truly the bottleneck of a system, then the next step would be to use a truly cache-oblivious or cache-aware data structure. Such a data structure is presented in [9]; it answers queries in $O(\log_B n)$ memory transfers (where B is the block size of the cache), at the expense of $O(n \log^2 n)$ storage space. Note that the increased space requirement might make the data structure impractical for large datasets - we have seen how even range-trees with $O(n \log n)$ space can use more than 512Mb of memory for a reasonable dataset. See also [10] for a similar result with a slightly better storage bound $O(n \log^2 n / \log \log n)$.

The results of this project are a nice example of how computer architectures can change over time and aspects that were insignificant become important in implementing and choosing the right data structures.

Acknowledgments

I would like to thank Prof. David Karger for help and suggestions on the project; also, the U.S. Census Bureau for the TIGER database; the 6.170 staff for code to parse the TIGER database; and AMD for the great profiling program CodeAnalyst.

References

[1] M.I. Shamos, F.P. Preparata. Computational Geometry - An Introduction. Corrected and expanded second printing. Springer-Verlag, New York - Berlin - Heidelberg - Tokyo 1988.

[2] J.L. Bentley, Algorithms for Klee's rectangle problems. Technical Report, Carnegie-Mellon University, Pittsburgh, Penn., Department of Computer Science, 1977.

[3] D.E. Willard, Predicate-oriented database search algorithms, Harvard University, Cambridge, MA, Aiken Computation Laboratory, Ph.D. Thesis, Report TR-20-78, 1978.

[4] G.S. Lueker, A data structure for orthogonal range queries, *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*, pp. 28-34 (1978).

[5] TIGER: Topolically Integrated Geographic Encoding and Referencing system, U.S. Census Bureau, *http://www.census.gov/geo/www/tiger/*

[6] T.M. Chilimbi, J.R. Larus, M.D. Hill. Improving pointer-based codes through cache-conscious data placement. Technical report 98, University of Wisconsin-Madison, Computer Science Department, Madison, Wisconsin, 1998.

[7] J. Rao, K. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. Proceedings of the 25th International Conference on Very Large Data Bases, pages 78-89, 1999.

[8] Z. Kasheff. Cache-Oblivious Dynamic Search Trees. Master Thesis, MIT, 2004.

[9] P.K. Agarwal, L. Arge, A. Danner, B. Holland-Minkley. Cache-Oblivious Data Structures For Orthogonal Range Searching. Proceedings of the 19th Annual Symposium on Computational Geometry, 2003.

[10] L. Arge, G.S. Brodal, R. Fagerberg. Cache-Oblivious Planar Orthogonal Range Searching and Counting. Proceedings of the 21st Annual Symposium on Computational Geometry, 2005.