Master-Client R-trees: A New Parallel R-tree Architecture *

Bernd Schnitzer
University of Erlangen-Nuernberg
Department of Computer Science IV
Martensstr. 1, 91058 Erlangen, Germany
email: schnitz@informatik.uni-erlangen.de

Scott T. Leutenegger
Math and Computer Science Department
University of Denver, Denver, Colorado 80208-0189
email: leut@cs.du.edu

Abstract

Scientific databases must be able to efficiently run subset retrievals of multi-dimensional data sets. If the data sets are very large significant retrieval speedups can be obtained via parallelism. In this paper we present a new parallel distributed shared nothing Rtree architecture. To the best of our knowledge this is the first significant experimental study demonstrating practical application of parallel Rtrees in a shared nothing environment. We provide experimental results demonstrating actual speedups for several synthetic and real data sets. In addition, we conduct experimental studies to investigate the effect of several declustering strategies and communication parameters.

1 Introduction

Scientific databases can be used to store and retrieve output from numerical simulations. The outputs from numeric simulations are often very large. This size problem is further exasperated by the use of parallel supercomputers resulting in grids and grid solution sets in the hundreds or thousands of megabytes. In addition to being large, numerical simulation output is usually 2 or 3 dimensional and sometimes of higher dimensions.

After solving a numerical simulation, scientist usually conduct some form of post-mortem analysis and/or data visualization. Often this exploration phase is a significant effort and can be sped up by tools to aid the scientist. One example is to integrate efficient database subset retrieval with visualization systems to enable a scientist to explore data at a coarse granularity and then zoom in on regions of interest [17]. A system that would be tolerated by end users would require interactive subset retrieval times (on the order of a second or two). In addition to post-mortem analysis, subset retrieval could be integrated into a steering program to direct currently executing simulations.

The database community has developed numerous indexing schemes to support subset retrieval of multi-dimensional data with Rtrees [10] and their variants being the most popular choice. By storing the bounding boxes of arbitrary geometric objects, such as points, polygons, or more complex objects, Rtrees can be used to determine which objects intersect a given query region. A significant body of literature dealing with improving dynamic and static Rtrees exits [1, 2, 4, 8, 7, 9, 12, 13, 16, 15, 20, 22, 25, 26].

For large data sets a single Rtree may prove insufficient at providing interactive response times. Parallelism provides the potential to significantly speedup of retrieval. Parallel shared nothing architectures [5] are a particularly attractive approach for cost efficiency. In this work we provide an experimental demonstration of the efficacy of the parallel shared nothing approach for speeding up queries on actual scientific data sets.

Despite significant attention to sequential Rtrees, little work has been done on parallel Rtrees. Kamel and Faloutsos first argue that since disk I/O is the bottleneck it makes sense to build a "parallel" Rtree on a single machine with multiple disks [12]. The paper proposes three architectures: super-nodes (striping), independent trees on each disk, and multiplexed Rtrees (a single tree whose nodes are spread across all the disks in the system). Furthermore, they consider four algorithms for declustering nodes across the disks. Specifically they consider Round Robin, Minimum Area, Minimum Intersection, and their proposed Proximity Index method. They show through simulation that multiplexed Rtrees are superior to super node Rtrees, and that using Proximity Indexing declustering results in up to 55% fewer disk accesses relative to Round Robin.

*The work of Schnitzer was done while a visiting student at the University of Denver. This work has been partially supported by the National Science Foundation under grant IRI-9610240 and by the Colorado Advanced Software Institute under grant TT-97-06.

We will show that Proximity Indexing is not so effective in a distributed shared nothing environment.

We are interested in a distributed shared nothing environment [5] for the following two reasons. First, often an existing network of machines with one disk per machine already exists. A parallel shared nothing architecture allows for query speedup without additional hardware expense. Second, often a network of machines is used for computation as well as data access. In this case CPU utilization does become an issue and hence one disk per machine will allow the CPU to be utilized by other applications. Thus, we consider how to best parallelize Rtrees in a parallel shared-nothing environment.

Multi-machine parallelization of Rtrees has been considered in a few previous works. In [3] the authors propose a new parallel nearest neighbor method and compare with parallel $R^{\star}$ trees. The work focuses on nearest neighbor queries (not range queries) and no details on their implementation of parallel $R^{\star}$ is given.

In [14] the authors suggest a "Master Rtree" architecture. One site contains all the internal nodes of the parallel Rtree and the leaf level nodes are declustered across several sites. The major focus of the work is on finding the optimal declustering "chunk size". The authors develop an analytical model which they then use to investigate chunk size trade-offs. An experimental implementation is done but only to validate the model. No results investigating potential for query response time reduction are discussed or presented. In Section 3 we describe the Master Rtree (M-Rtree) and Master-Client Rtree (MC-Rtree) architectures.

To the best of our knowledge, significant experimental results demonstrating the utility of speeding up multi-dimensional subset queries via parallel Rtrees have not previously been presented.

The main contributions of this work are:

1. We provide real experimental (not simulation) results demonstrating the potential (and limits) of query response time reduction via parallelism. We consider both real and synthetic data sets.
2. We experimentally study various declustering algorithms to determine the importance of the declustering method in a shared-nothing platform.
3. A new parallel Rtree architecture (Master-Client Rtree).
4. We experimentally demonstrate the effect of tuning communication parameters.

The rest of this paper is organized as follows. In Section 2 we present background material on Rtree packing. In Section 3 we present both the M-Rtree and MC-Rtree algorithms. In Section 4 we present our experimental methodology. In Section 5 we present experimental results and in Section 6 we conclude.

2 Loading Algorithms

In this section we provide a brief review of Rtree packing. We assume the reader is familiar with Rtrees. Our MC-Rtree assumes the trees are packed at the master and at each client. We have conducted experimental studies with both Hilbert and STR packing. The qualitative results and trends are the same for the two methods. For space limitations we choose to present only the STR results. Complete results including the Hilbert results can be found in [23].

For packed Rtrees, all data must be available before the Rtree is built. This allows preprocessing of the data, thus improving the Rtree's structure by looking at the complete data set in order to decide which object should be stored in which node, and creating Rtrees with nearly 100% space utilization.

The packing algorithms we consider follow a general process similar to building a B-tree from a set of keys by creating one level at a time from the bottom (leaves) up [21, 22] as follows:

Let M be the maximum number of entries per node, and r the number of rectangles in the data file.

1. Preprocess the n (initially r) input rectangles for the current level (initially the bottom) so that rectangles are ordered in $\lceil n/M \rceil$ consecutive groups of M rectangles (the last group with possibly fewer). A group defines a node in the current level.
2. Load the $\lceil n/M \rceil$ groups of rectangles into disk pages and output each (MBR, page-number) into a temporary file. The page-numbers are used as the child pointers in next higher level.
3. Recursively pack these MBRs into nodes at the next level up, until the root node is done.

The original Rtree packing algorithm of Roussopoulos and Leifker sorts the data on the first coordinate. We consider two different preprocessing methods:

Hilbert packing [12]: The preprocessing is a sorting by the distance along the Hilbert curve from the origin of the curve to the center of each rectangle. Points that are close in Hilbert distance are also close in Euclidean space, thus the method packs nodes appropriately.

Sort-Tile-Recursive (STR) [16] : For a k-dimensional set of n (hyper-) rectangles, the STR preprocessing is based on the "tiling" of the plane for the 2D case. Determine the number of current level pages $P = \lceil n/M \rceil$. Then, sort the (hyper-) rectangles according to the first coordinate of their center. Divide the sorted set into $S = \lceil P^{\frac{1}{k}} \rceil$ slabs, where a slab consists of a run of $M \cdot S^{k-1}$ consecutive (hyper-) rectangles [1]. Recursively process each slab by its remaining $k - 1$ coordinates.

3 Distributed Rtrees

In this section we describe the tree structure, query, and loading algorithms for the previously proposed Master Rtree (M-Rtree) and our new Master-Client Rtree (MC-Rtree). Last we describe the declustering algorithms considered.

3.1 Master Rtree (M-Rtree)

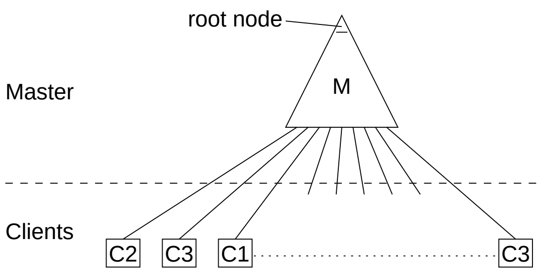

Figure 1: The M-Rtree architecture for one master and three client machines. The triangle denotes an Rtree, squares represent leaf nodes (or chunks).

The M-Rtree was proposed in [14]. A drawing of the overall architecture is shown in Figure 1. One dedicated machine becomes the master server and contains all internal nodes of the parallel Rtree. The leaf level at the master does not hold the leaf level of the global tree, but rather (MBR,site-id,page-id) tuples for each global leaf level node. The leaf nodes of the global tree are declustered across the other sites. The (site-id,page-id) is used to designate at which page and which site the leaf level page is located.

A query is sent to the master. The master searches the local portion of the M-Rtree and builds a list of all (site-id,page-id) pairs needed. Once the entire list is constructed, each site containing required data is sent the query MBR and the page list pertinent to that site. Each site then retrieves the pages from disk and sends qualifying rectangles back to the master.

We now describe the M-Rtree loading process. The process proceeds in four steps:

1. Preprocessing: At the master the data is partitioned into G groups of size M, where M is the maximum number of data rectangles that fit per disk page. In our studies this partitioning is done using the Hilbert [12] or STR [16] packing preprocessing sort.

2. Declustering: Assign each one of this G groups to one of the sites according to a declustering algorithm described in section 3.3.

3. Write client pages to disk: Client sites simply write the pages out to disk. Note this can be done as the data is declustered. Return to the master a list of (MBR,page-ids).

4. Build the Master: After declustering the data set, the master Rtree is built by Hilbert or STR packing the (MBR,site-id,page-id) information returned from the clients.

3.2 Master Client Rtree (MC-Rtree)

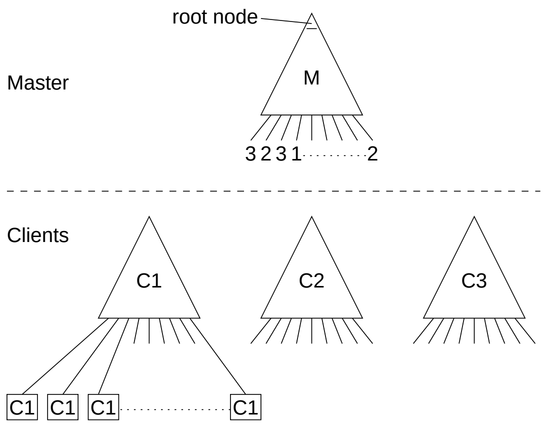

Figure 2: The MC-Rtree: Both, master and client machines, contain an Rtree as index structure (triangles). The master (M) stores only the client ids of the data nodes, whereas the data nodes (rectangles) themselves are kept, embedded into Rtree structures, at the client machines (C1 – C3).

[1]The expression $M \cdot S^{k-1}$ was misprinted in [16].

A drawing of the overall MC-Rtree architecture is shown in Figure 2. The data at the master machine is organized the same way as suggested by the M-Rtree, i.e. only non-leaf nodes are stored at the master site. Unlike the M-Rtree, only the (MBR,site-id) pairs are needed at the leaf level of the master site, not the page-ids. Each client builds a complete Rtree for the portion of the data assigned to it. Since there are Rtrees at the master and the clients, we refer to this new model as Master-Client Rtree. Note that there is redundant information stored in the MC-Rtree compared to the M-Rtree. Having this redundant information allows for a reduction in overhead at the master and global communication costs. Since tree fan-out is 100, this redundant storage of the internal nodes adds only an additional 1% space overhead.

A query is sent to the master and is processed locally as in the sequential case. When a leaf node is reached, the query MBR is sent to the client site designated in the leaf node. As soon as a client receives a request, it starts processing the query autonomously. No further information about which nodes need to be fetched is required. All qualifying data rectangles are retrieved and shipped back to the master.

After sending a request to a client, the master adds the client site-id to a list that keeps track of which clients are working on the query. Since the clients work autonomously, a maximum of one request is sent to each client. The master continues searching until either all clients are notified of the query, or no more MBRs intersecting the query are found. The master then waits for answers and collects the qualifying data items sent back by the clients.

We now describe the MC-Rtree loading process. Again the process is composed of four steps:

1. Preprocessing: Same as the preprocessing step for M-Rtrees.
2. Declustering: Same as the declustering step for M-Rtrees.
3. Build the Master: After declustering the data set, the master Rtree is built by Hilbert or STR packing the (MBR,site-id) pairs of the declustered chunks.
4. Build the Clients: At each client site build a tree by Hilbert or STR packing data assigned to that client.

Note, unlike the M-Rtree, building all trees including the master tree can be done in parallel, reducing the load time of the tree.

3.3 Declustering Algorithms

We consider several different declustering algorithms. The declustering algorithm offers the potential to effect performance. In fact, in [11] where a single machine multiple disk architecture is simulated the authors show the PI declustering method to result in up to 55% fewer disk accesses than the round robin method. As we will show in Section 5 this is not the case for distributed Rtrees.

The declustering algorithms divide the set of data rectangles into C chunks of M rectangles each and then assign each chunk to one of the S sites. Again, in our studies we assume a packing algorithm (Hilbert or STR) is used at the master to pack the data rectangles into C chunks, where each chunk corresponds to a leaf level page in a packed sequential Rtree. Note, it may be possible that a lower utilization packing as suggested by DeWitt et. al [6] may further reduce actual response times. We intend to investigate this in the future. We then decluster the chunks using one of the following declustering methods:

Existing Declustering Methods

- Random (RAND). Each chunk is assigned to a site with equal probability thus ensuring good load balance. No information about earlier decisions is needed so the space requirement is $O(1)$. The time to determine the destination site for one chunk is constant. Thus, time complexity grows linearly with the number of chunks and is $T(C)$, where C is the number of chunks.
- Round Robin (RR). Ensures perfect load balance. Once again the space requirement is $O(1)$ and the time requirement is $T(C)$.
- Proximity Index (PI). This metric was proposed in [11] and was shown to outperform all other declustering approaches. First each site is assigned one chunk. For the remaining chunks, at each site find the chunk site most similar (as defined by the PI metric) and record the PI index for this chunk versus the chunk to be assigned. Send the chunk to the site whose metric is most dissimilar. The objective function is the probability that two chunks will be accessed by a query. Intuitively, if two

chunks are likely to be accessed by the same query they should be placed on separate sites. Exact details of the metric can be found in [11].

For the PI declustering algorithm to work, knowledge about the locations of all previously assigned chunks are required. Therefore, the space needed by this method grows linearly with the number of chunks and is $O(C)$. For each assigned chunk all previously assigned chunks have to be examined. Thus, the time complexity is $T(C^2)$.

New Declustering Methods

- Proximity Index STR (PISTR). This is a modification of PI to reduce both the time and space complexity. Since the chunks of STR are processed slice by slice, only the contents of the most recently touched slice are relevant for the assignment of the current chunk. The chunks from before the most recently touched slice are farther away and less similar to the current chunk. Note, it may be possible in some pathological case that chunks from a previous slice are closer, but this is highly unlikely. Thus, it is sufficient to keep only information about the recently processed slice which contains $S = \sqrt{C}$ chunks. This leads to $O(\sqrt{C})$ space and $T(C\sqrt{C})$ time complexity. This declustering method is referred to as PISTR, because it was developed with regard to the STR packing algorithm. Note that it can also be applied to data sets packed with other algorithms.

- Volume Bounding Rectangle (VBR). The algorithm is similar to PI except the metric of "similar" is defined as the volume of the the MBR enclosing two chunks. The smaller the volume is, the more similar the two compared chunks are. Since the basic algorithm is the same as for PI, the time and space complexity are the same: $T(C^2)$ and $O(C)$. Also, like in the case of PI, a special algorithm for STR packing can be developed, reducing the complexity to $O(\sqrt{C})$ and $T(C\sqrt{C})$.

4 Experimental Methodology

In this section we describe our experimental methodology, hardware, and data sets used. The hardware platform which all tests were run on was a workstation cluster (70Mhz Sun Sparc 5s) running Solaris 2.5. The sites were connected with a 100 MBit full duplexed Ethernet. One disk per site was used for storing the Rtree (or parts of it). The disks were accessed as a "raw device" thus eliminating interference of UNIX buffering mechanisms. This allowed us to use relatively small data sets and still have disk resident (rather than main memory resident) data. The device was buffered by an LRU buffer containing 100 4K pages, thus the buffer can hold roughly 5% of the sequential Rtree for the smallest data set, and 2% for the largest. As the number of clients increases, the total amount of buffer space increases, e.g. for five clients, the sum of the buffers totals to 500 pages (10% – 25% of the data sets). In order to obtain valid results, the buffer was "initialized" before running time measurements. This was done by performing about 1000 small queries, which cause the buffer to be filled with nodes by the time the measurements started.

The presented results were obtained by measuring the real query response times. Reported response times are the measured difference in the time the last byte is received from all clients involved minus the time the query started at the master. To obtain accurate results, all tests were run during times of low or no user interference. However, it cannot be guaranteed that there was no interference at all. And like on all UNIX systems, there were always background processes (demons) with unpredictable behavior.

The queries themselves were distributed uniformly in the unit square. First, small queries, all of the same size, were generated. Then the size of the query region was increased and another set of queries was run[2]. This was repeated until the query size exceeded a certain limit (usually 75% of the unit square). Note that the area of the query region was increased exponentially. For each set of equally sized queries, the mean value of the response times is calculated and will be referred to as response time (instead of average response time) from now on.

4.1 Data Sets

All tests were run using the following four data sets. The data sets vary in size, skewedness, and type of data (points or rectangles). The UNI and EXP data sets are synthetic data whereas the CFD and VLSI are real data sets. The UNI and VLSI data sets are rectangle data, whereas the EXP and CFD are point data sets.

[2]Usually about 1000 queries for each query size were performed. But for large queries, e.g. retrieving more than 10,000 data items, the number of queries was reduced in order to keep the time for test runs in an acceptable limit.

Note, all data sets have been normalized to the unit square. Pictures of these data sets can be found in the full technical report [24].

- Uniform (UNI): 500,000 rectangles are distributed uniformly across the unit square. The density of this data set is one, i.e. the sum of the areas of the rectangles equals 1.0. This data set is the most regular one: all data rectangles have approximately the same size and there are no areas of high or low density in the unit square.
- Exponential (EXP): 500,000 points were synthetically generated. The x and y coordinates of data points are distributed according to the exponential distribution with mean 1.0 and then the data set is normalized to the unit square.
- Computational Fluid Dynamics (CFD): A 2-dimensional system of equations is used to model the air flows over and around aero-space vehicles [19]. The data sets are for a cross section of a Boeing 737 wing with flaps out in landing configuration at MACH 0.2. The data space consists of a collection of 208,688 points of varying density. Nodes are dense in areas of great change in the solution of the equations and sparse in areas of little change. See `www.cs.du.edu/~leut/MultiDimData.html` for more details and to obtain the data sets.
- (VLSI): A set of 453,994 highly skewed CIF data objects from the design of a chip provided by Bell Labs [18]. Not only do regions of very high and regions of low data density exist, but also the size of the rectangles differs strongly: the smallest and the biggest rectangle differ by a factor of 40,000 in size.

5 Experimental Results

In this section we present the results from our experimental studies. All results are for MC-Rtree using STR packing. Note again that the results for MC-Rtrees built using the Hilbert packing are analogous and are omitted for brevity. Hilbert results can be found in [23].

5.1 Speedup Results

Figures 3 - 6 contain the speedups of the STR MC-Rtree versus query size for the four data sets. The speedup is the response time of a sequential STR tree over the response time of the STR MC-Rtree. Round robin declustering is used for all the results. We justify this choice in the next section. Note the x-axis is logarithmic.

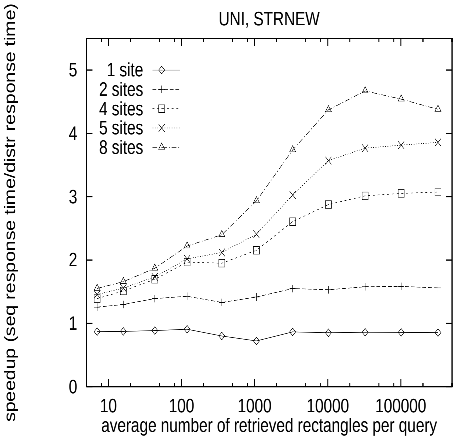

Figure 3: Speedups: UNI

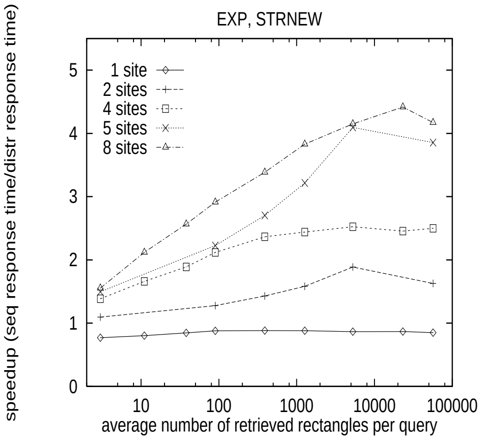

Figure 4: Speedups: EXP

There is a significant slowdown when only one site is used. This slowdown occurs since there is no parallelism and the algorithm must ship the query and results across the network. For more sites, a general trend is that small queries are not able to benefit from the parallelism whereas large queries do benefit from the additional parallelism. The exception is the VLSI data set where even small queries benefit from parallelism. This is because the VLSI data set results in a large amount of leaf level MBR overlap and even small queries must retrieve many pages. Hence, there is ample work to allow for speedup via parallelism. Note, if concurrent queries are issued even small queries will

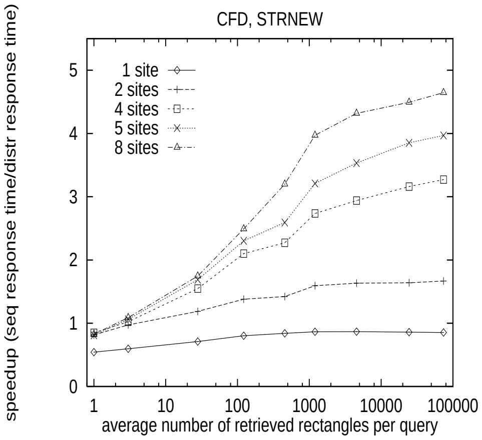

Figure 5: Speedups: CFD

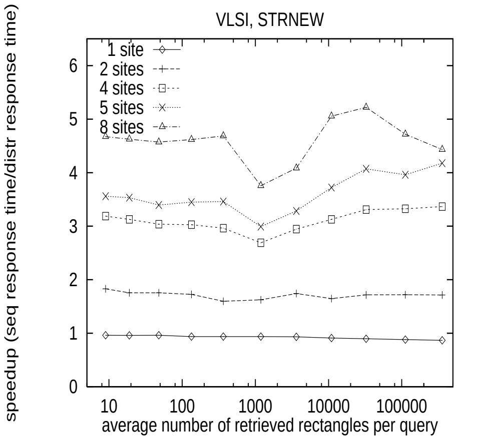

Figure 6: Speedups: VLSI

benefit from the increased processing power.

For all data sets, as soon as the number of rectangles returned is greater than 1000 (less than 1.0% of the data set for all data sets considered) significant speedups occur. Thus, distributed shared nothing parallelism of MC-Rtrees offers significant potential for decreasing query response time. The speedups level off at about 50-75% of the maximum achievable theoretical speedup. We believe this limitation is due to network saturation and are currently developing a parallel Rtree simulator to explore the effect of faster networks.

5.2 Declustering Results

The five tested declustering methods are PI, PISTR, RAND, RR and VBR. (see Section 3.3 for a description). In Figures 7 - 10 PI is compared to the other four

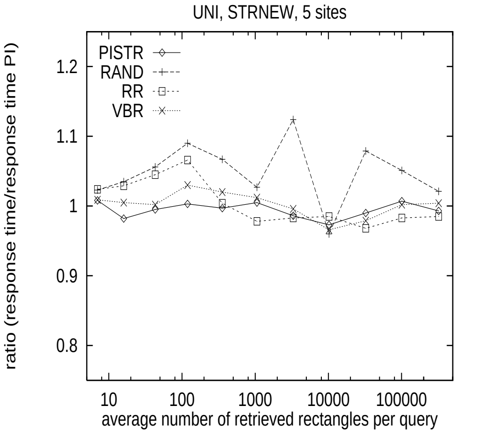

Figure 7: Declustering, Response Time Relative to PI, UNI

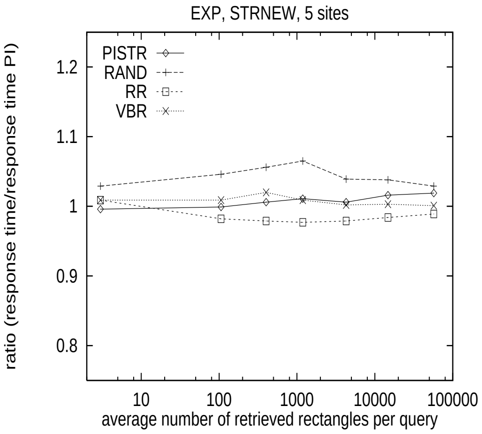

Figure 8: Declustering, Response Time Relative to PI, EXP

algorithms, i.e. for each of those algorithms, the ratio $\frac{ResponseTimeOther}{ResponseTimePI}$ is calculated. Therefore, if the line of a plot runs below 1, it means that the corresponding declustering method performs better than PI, above 1 it is vice versa. To compare two declustering methods the lower line indicates which algorithm has the shorter response time. Note, Figures 7 - 10 assume the MC-Rtree is distributed across 5 sites, the same set of experiments on 8 sites led to similar conclusions and graphs can be found in the full technical report [24].

Instead of dealing with each single data set, we make some general remarks. First, the effect of query size:

- Small queries:

  The response time should be roughly the same for all declustering algorithms because only one leaf

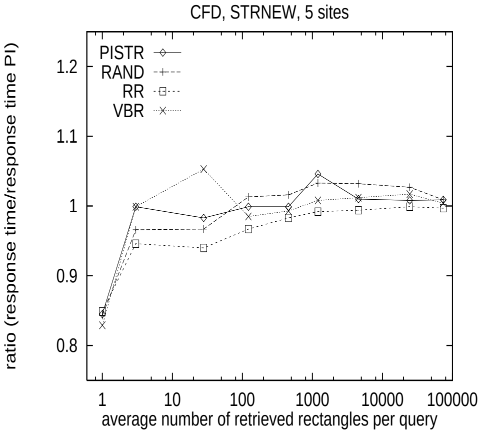

Figure 9: Declustering, Response Time Relative to PI, CFD

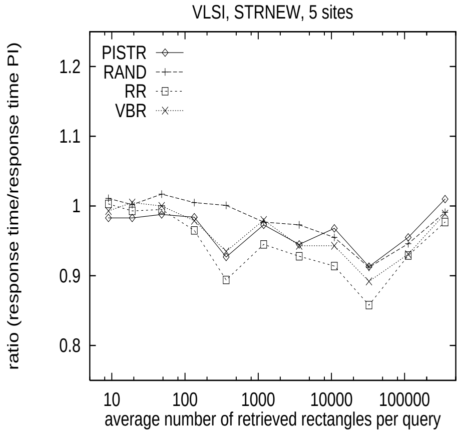

Figure 10: Declustering, Response Time Relative to PI, VLSI

level node at one site has to be fetched to satisfy the query. Thus, for small queries, no work can be done in parallel, and the declustering method does not have any influence on the response time.

- Medium sized queries:

  We define a medium sized query to be a query retrieving more than one leaf page but less than a few times the number of client machines. Now the declustering method determines how much work can be done in parallel. For example, if four nodes have to be retrieved from disk, but three of them are located at the same machine, the maximal possible degree of parallel work will not be achieved. Even so, the effects of poorly declustered data may still be blurred if the site with extra pages has them located on the same cylinder. If this is the case, the imbalance slowdown is not as significant, hence parallelism may not be as important as physical location of the nodes on disk. On our system reading blocks on the same cylinder is about one order of magnitude faster. For example reading one block takes 13ms whereas reading two consecutive blocks only takes 14ms.

- Large queries:

  In this case, all machines will work in parallel. Therefore, only extremely poor data declustering will cause a noticeable effect.

Now consider the relative performance of the different algorithms. With a few exceptions, the response times for the declustering algorithms lie within ± 10% of the mean value. In other words, no algorithm performs significantly better or worse than the others. In addition, the relative ordering is not the same in all the experiments. Thus, no conclusion about which declustering method works best can be made. This conclusion differs significantly from the claim in [11] that PI is superior to other methods. We conjecture the reason our conclusions differ is that the parallel algorithm overheads are more significant than the effect of imbalance and hence the choice of declustering algorithm is not as important for distributed shared nothing parallel Rtrees.

Also note that PISTR is very similar in performance to PI and is significantly less expensive (in terms of time and space) to implement. Furthermore, our new VBR algorithm turned out to be no better than the other algorithms. Finally, RAND is less predictable than the other algorithms.

Given that all the declustering algorithms are similar, and that Round Robin is so inexpensive, we have decided to use Round Robin in our experiments.

Note, the two exceptions where the declustering policy does change performance by more than 10% occurs for the CFD and VLSI data sets. Here, PI is up to 17% worse than the other algorithms. We offer one possible argument why PI may perform worse than other algorithms in some cases.

If the data is highly skewed, and especially when the variance in size of the data or bounding rectangles is large, PI may decluster the nodes poorly. As an example consider when a large rectangle completely contains a smaller rectangle. Assume a second small rectangle is very close but not intersecting the first small rectangle.

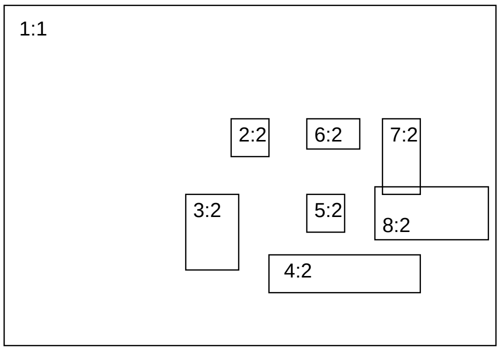

Figure 11: Example for bad declustering when using PI; the rectangles are labeled with <rectangle number>:<assigned site id>

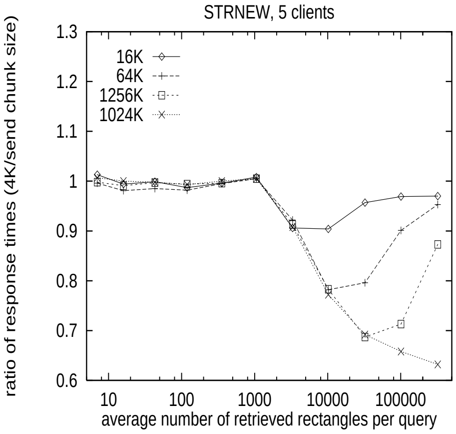

Figure 12: Send Size, 5 Sites

In this case, the large rectangle and first small rectangle will be assigned a higher proximity index than the two smaller rectangles. This can cause a problem as follows. Consider Figure 11. Assume rectangles 1 through 7 have to be declustered in that order, using the proximity index measure. Since the large rectangle (number 1) "intersects" with all other rectangles, its proximity index with any of the smaller rectangles will be greater than any pairing of the smaller rectangles. If the data has to be declustered between two clients, rectangle 1 is initially placed at site 1 and rectangle 2 at site 2, then all other rectangles will be assigned to site 2 resulting in poor balance. This anomaly may partially explain the poor performance of PI for the CFD and VLSI data sets.

5.3 Communication Send Buffer Size

The implementation of the MC-Rtree allows the variation of the communication send buffer size. Figure 12 shows how the send buffer size affects the query response time assuming 5 clients. The ratio of response time for a 4K send size over the response time for other send sizes versus query size is plotted. Thus, when the value is less than 1 the 4K send size is preferable. If only a few data rectangles are retrieved the response time is independent of the different send buffer sizes. This is because the communication buffers at the clients are large enough to store all data objects retrieved by a small query. First the query is processed completely, then the qualifying data is shipped back to the master.

As soon as the queries get bigger and cause the 4K buffer to get full, the clients send its contents immediately, before query processing is continued. While the hardware transmits the data and the master machine is busy with receiving, the client continues working on the query. On the other hand, when using bigger buffer sizes for sending, the clients still gather data and wait until the query is completed. Then, a large amount of data is sent from each client to the master which takes considerably longer than sending what is left in the last packet if 4K buffer sizes are used.

For very large queries, this effect lessens again, because the time needed for sending the last packet becomes negligible relative to the total time spent for transferring data.

In general, a smaller send buffer size (about 4K) seems to perform better than a larger size. However, in some cases – i.e. if the network becomes the bottleneck (slow network, high number of machines working in parallel and only big queries are expected) – choosing a larger send buffer size may result in shorter response times.

6 Conclusions

In this paper we provide the first real experimental speedup results for region queries using parallel Rtrees on a shared nothing architecture. We have empirically shown that shared nothing parallel Rtrees offer significant potential for reducing query response time on scientific data sets. Furthermore, surprisingly, we have shown that the choice of declustering algorithm is not a significant determining factor in overall response time. Finally, we have proposed a modified parallel Rtree architecture, the Master Client Rtree (MC-Rtree).

References

[1] B. Becker, P. Franciosa, S. Gschwind, T. Ohler, G. Thiemt, and P. Widmayer. Enclosing many boxes by an optimal pair of boxes. In *Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science (STACS 92)*, pages 475–486. Springer LNCS #577, 1992.

[2] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger. The $r^\star$-tree: an efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD*, pages 323–331, may 1990.

[3] S. Berchtold, C. Bohm, B. Braunmuller, D. Keim, and H-P. Kriegel. Fast parallel similarity search in multimedia databases. In *Proc. ACM SIGMOD*, pages 323–331, 1997.

[4] S. Berchtold, D. Keim, and H-P. Kriegel. The x-tree: An index structure for high-dimensional data. In *Proc. 22nd Conf on Very Large Databases (VLDB 96)*, pages 28–39, 1996.

[5] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM (CACM)*, 35(6):85–98, June 1992.

[6] D. DeWitt, N. Kabra, J. Luo, J. Patel, and J-B. Yu. The client/server paradise. In *Proc. of Int. Conf. on Very Large DataBases (VLDB94)*, pages 558–569, Santiago, Chile, March 1994.

[7] Y. Garcia, M.A. Lopez, and S.T. Leutenegger. A greedy algorithm for bulk loading r-trees. In *Proc. of ACM Symposium on Geographic Information Systems (GIS'98)*, pages 163–164, Washington D.C., November 1998.

[8] Y. Garcia, M.A. Lopez, and S.T. Leutenegger. On optimal node splitting for r-trees. In *Proc. of VLDB 98*, pages 334–344, New York, NY, August 1998.

[9] D. Gavrila. R-tree index optimization. Technical Report CS-TR-3292, University of Maryland, College Park, Jun 1994.

[10] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, pages 47–57, Boston, Mass, June 1984.

[11] C. Faloutsos I. Kamel. Parallel r-trees. In *Proc. ACM SIGMOD*, pages 195–204, 1992.

[12] I. Kamel and C. Faloutsos. On packing r-trees. In *Proc. of Second Int. Conf. on Information and Knowledge Management (CIKM)*, Washington, D.C., November 1993.

[13] I. Kamel and C. Faloutsos. Hilbert r-tree: an improved r-tree using fractals. In *Proc. of VLDB Conference,*, pages 500–509, Santiago, Chile, September 1994.

[14] N. Koudas, C. Faloutsos, and I. Kamel. Declustering spatial databases on a multi-computer archetecture. In *Proc. of Extended Database Technologies (EDBT)*, 1996.

[15] S.T. Leutenegger and M.A. Lopez. The effect of buffering on the performance of r-trees. In *Proc. of the 15th International Conference on Data Engineering (ICDE 98)*, pages 164–171, 1998.

[16] S.T. Leutenegger, M.A. Lopez, and J.M. Edgington. Str: A simple and efficient algorithm for r-tree packing. In *Proc. of the 14th International Conference on Data Engineering (ICDE 97)*, pages 497–506, 1997.

[17] S.T. Leutenegger and K-L. Ma. Fast retrieval of disk-resident unstructured volume data for visualization. In *DIMACS Workshop on External Memory Algorithms and Visualization, May 1998, to appear in an AMS_DIMACS volume.*

[18] M. Lopez, R. Janardan, and S. Sahni. Efficient net extraction for restricted orientation designs. *IEEE Trans on Computer-Aided Design*, 15(9):1151–1159, 1992.

[19] D.J. Mavriplis. An advancing front delaunay triangulation algorithm designed for robustness. *Journal of Computational Physics*, 117:90–101, 1995.

[20] B-U. Pagel, H-W. Six, H. Toben, and P. Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proc. ACM PODS*, pages 214–221, May 1993.

[21] A.L. Rosnberg and L. Snyder. Time and space optimality in b-trees. *ACM Transactions on Database Systems*, 6(1), 1981.

[22] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. In *Proc. ACM SIGMOD*, pages 17–31, Austin, Texas, May 1985.

[23] B. Schnitzer. Development of a distributed r-tree and experimental comparison of declustering techniques. Master's thesis, University of Erlangen-Nurnnerg, Germany and Univeristy of Denver.

[24] B. Schnitzer and S.T. Leutenegger. Master-client r-trees: A new parallel r-tree architecture. Technical Report COMP-98-01, University of Denver, 1998. URL: http://www.cs.du.edu/~leut/ssdbm99-TR.ps.

[25] T. Sellis, N. Roussopoulos, and C. Faloutsos. The r+ tree: A dynamic index for multidimensional objects. In *Proc. of 13th International Conference on Very Large Databases (VLDB-87)*, pages 507–518, sep 1987.

[26] Y. Theodoridis and T. Sellis. A model for the prediction of r-tree performance. In *Proc. of 8th Symposium on Principles of Database Systems (PODS)*, May 1996.