GRT: Global R-Trees

A Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Alec Wiseman

Graduate Program in Computer Science and Engineering

The Ohio State University

2013

Master’s Examination Committee:

Prof. P. Sadayappan, Adviser

Prof. Srinivasan Parthasarathy
ABSTRACT

As hardware technology advances, distributed architectures are becoming more and more complex. Writing software that fully and efficiently utilizes massively parallel systems can be an extremely difficult task. We seek to provide a shared memory view of distributed address space to ease the difficulty of implementing parallel algorithms. This thesis describes our parallel implementation of the R-Tree structure with this design goal in mind. Our approach differs from other parallel R-Tree implementations in that we map the linked structure of an R-Tree onto an array structure. We then leverage the Global Arrays framework to distribute the array structure across our parallel system. Global Arrays is a framework produced by Pacific Northwest National Laboratories that provides a shared memory view of multi-dimensional, distributed arrays. In order to improve the efficiency and performance of the Global Arrays framework, we have also implemented a caching system as a layer between our application and framework. Our cache provides automatic data replication across a distributed system as well as rudimentary support for write operations without affecting read performance.
ACKNOWLEDGMENTS

I would like to thank my advisor, Professor P. Sadayappan, for this support and guidance. I would also like to thank Professor Srinivasan Parthasarathy for his leadership and contributions to this project. Finally, I would like to thank my fellow graduate students that have collaborated with me on this project: Qingpeng Niu, S M Faisal, Tyler Clemons, and Ye Wang.
VITA

2008 ........................................B.S., University of Rio Grande, Computer Science, Rio Grande, OH

2010 ........................................M.S., College of William and Mary, Computer Science, Williamsburg, VA

2010-2013 .................................Ph.D Student, Department of Computer Science and Engineering, The Ohio State University, Columbus, OH

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

High Performance Computing  Professor P. Sadayappan
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td></td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td></td>
<td>iii</td>
</tr>
<tr>
<td>Vita</td>
<td></td>
<td>iv</td>
</tr>
<tr>
<td>List of Tables</td>
<td></td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td></td>
<td>viii</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>Global R-Trees: Implementation</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>C API Specification</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Parallel Access</td>
<td>8</td>
</tr>
<tr>
<td>3.</td>
<td>Generalized Global Array Cache</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Design Detail</td>
<td>10</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Brick Identification</td>
<td>11</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Data Copy</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Key Challenges</td>
<td>13</td>
</tr>
<tr>
<td>3.2.1</td>
<td>GA API Constraints</td>
<td>13</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Arbitrary Dimensionality</td>
<td>14</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Modular Replacement Policy</td>
<td>14</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Custom Heap Management</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>Write Support</td>
<td>16</td>
</tr>
</tbody>
</table>
3.3.1  Coherence/Consistency Model ........................................ 16
3.3.2  Bloom Filters ............................................................... 17
3.4  Experiment Results ............................................................ 19
  3.4.1  Qwalk Case Study ........................................................ 19
  3.4.2  Simulation Study .......................................................... 20

4.  Performance Results ............................................................ 24
  4.1  Design ................................................................. 24
  4.2  Results ................................................................. 25

5.  Future Work ................................................................. 30

6.  Conclusions ................................................................. 32

Bibliography ................................................................. 33
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Results from the hit rate experiment</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Results from the remote access rate experiment</td>
<td>22</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Global R-Tree nodes are converted into byte arrays and then stored in a 2-D Global Array. The first &quot;node&quot; in the Global Array is reserved for important global metadata.</td>
</tr>
<tr>
<td>2.2</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>A GRT Node contains 12 bytes of metadata followed by a series of index records. The node pre-allocates room for up to M index records, where M is the max_records parameter from the create function.</td>
</tr>
<tr>
<td>3.1</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>The GA address space is divided into uniform bricks of the same dimensionality. Region requests on the GA may span multiple and partial bricks.</td>
</tr>
<tr>
<td>3.2</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Bricks are uniquely indexed in row-major order, calculated using dimension stride.</td>
</tr>
<tr>
<td>3.3</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>A 2x2x2 3-D brick is distributed across linear address space into a 3x3x4 region buffer from the user. Each brick element must be copied to the correct position in the region buffer.</td>
</tr>
<tr>
<td>3.4</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Results of the Qwalk experiment</td>
</tr>
<tr>
<td>3.5</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Results from the hit rate experiment</td>
</tr>
<tr>
<td>3.6</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Results from the remote access rate experiment</td>
</tr>
<tr>
<td>4.1</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>NCAR GIS test throughput as the number of processors increase with cache disabled (left) and the cache enabled (right)</td>
</tr>
<tr>
<td>4.2</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>NCAR GIS test results from 1 processor run with the cache disabled (left) and the cache enabled (right)</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>4.3</td>
<td>NCAR GIS test results from 2 processor run with the cache disabled (left) and the cache enabled (right)</td>
</tr>
<tr>
<td>4.4</td>
<td>NCAR GIS test results from 4 processor run with the cache disabled (left) and the cache enabled (right)</td>
</tr>
<tr>
<td>4.5</td>
<td>NCAR GIS test results from 8 processor run with the cache disabled (left) and the cache enabled (right)</td>
</tr>
<tr>
<td>4.6</td>
<td>NCAR GIS test results from 16 processor run with the cache disabled (left) and the cache enabled (right)</td>
</tr>
<tr>
<td>4.7</td>
<td>NCAR GIS test results from 32 processor run with the cache disabled (left) and the cache enabled (right)</td>
</tr>
<tr>
<td>4.8</td>
<td>NCAR GIS test results from 64 processor run with the cache disabled (left) and the cache enabled (right)</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Performance optimization is a challenging problem facing software engineers that only becomes more difficult as hardware technology advances forward. Distributed architectures, in particular, can be extremely difficult to fully and efficiently utilize. Algorithms for shared memory systems tend to be much simpler and easier to implement than those for distributed memory systems due to the decreased need for data synchronization.

Distributed Shared Memory (DSM) is an area of study in which researchers target the difficulty of engineering software for distributed systems. We attempt to produce efficient, optimized libraries that provide software engineers with a shared memory view of distributed address space. When using these libraries, the engineer can access data that is stored across multiple nodes without having to explicitly move data between nodes or manually manage data coherence and consistency.

One of the major challenges of this effort is to provide efficient, fast implementations of these libraries. Each use-case presents slightly different data access patterns and thus requires different optimizations to achieve good performance. To meet this challenge, we are producing libraries targeted at common data structures, each tailored to the needs and use-cases of their respective target. Global Arrays (GA) is
one such library produced by Pacific Northwest National Laboratories (PNNL) [19] that provides a shared memory view of multi-dimensional, distributed arrays. Global Trees [10] extends the concept to tree and linked list data structures.

Following the work of Larkins et al. [10], we continue to examine linked data structures with the Global R-Trees project. R-Tree structures [7] differ from generic trees and linked lists in that they follow the B-Tree approach to their construction, guaranteeing that the tree will be balanced. Also, R-Trees are designed specifically for use with spatial data as an indexing structure. These differences present unique optimization challenges, particularly in common use-case access patterns.

Several others have published parallel R-Tree implementations before ours. One such implementation, SD-Rtree [5], leverages the client-server model to construct the distributed linked structure. Another has implemented a parallel R-Tree that utilizes the GPU to speed up queries [13]. Our implementation differs from these in two major design approaches: 1) Instead of trying to construct a distributed linked structure, we map the linked structure to an array structure and then distribute the array. 2) We do not parallelize an individual query’s execution, but rather we achieve greater throughput by making the R-Tree available to a parallel application that can then run multiple queries simultaneously.

Tree structures in general can be tricky to implement efficiently on parallel systems due to load balancing challenges inherent to their access pattern. A balanced load across a parallel system should see data residing on a given node being accessed at roughly the same frequency as data residing on any other node. Nodes at a higher level of a tree are likely to be accessed more frequently than nodes on lower levels. However, depending on the application, access frequencies on any given node could
potentially vary in unpredictable ways. Thus, our system must be able to dynamically adapt to the access pattern of the application on the fly.

This is one of the design challenges we seek to assuage with our Generalized Global Array Cache (GRC). By using a cache sub-system, we can facilitate automatic data replication across the distributed system, effectively allowing node data to be efficiently stored on multiple nodes as needed. Furthermore, because we have mapped our R-Tree to an array structure of contiguous memory, we are also able to leverage the cache to reduce network traffic in a distributed system by taking advantage of the temporal and spatial locality present in the R-Tree’s query access pattern.
Our primary goals for our parallel implementation of the R-Tree structure were to: 1) Emulate sequential R-Tree semantics and to 2) Provide efficient parallel access to all processes in the user application. The current version of GRT is designed to index spatial data stored in a 2-D Global Array database. Index records stored in leaf nodes of the GRT contain reference information for accessing their respective elements in the GA.

2.1 C API Specification

In order to properly emulate sequential R-Tree semantics, we designed our API to provide the same base-line functions given in the original paper [7]. The current version supports creating GRTs, inserting records, and executing range queries. Future versions may expand this API and include record delete support and more complex functions.
void GRT_Initialize(int *argc, char **argv)

Parameters
argc  count of the parameters passed to the main function
argv  pointer to parameters passed to the main function

The initializer function for GRT and its subsystems. Must be called once before creating any GRTs.

void GRT_Terminate()

Finalizer function that deallocates GRT’s internal metadata and its subsystems.

int32_t GRT_Create(uint32_t ndims, uint32_t max_entries, uint32_t min_entries, char *grt_name, uint32_t grt_size)

Parameters
ndims  number of dimensions of the spatial data to be index
max_entries  maximum number of entries a node may contain
min_entries  minimum number of entries a node may contain
grt_name  name for the GRT
grt_size  maximum size of the GRT (in bytes)

Return Value
int32_t  handle for the newly created GRT

GRT_Create is used to create a new Global R-Tree structure. Its return value is an integer handle that can be used to reference the structure in future calls to the API. Currently, GRT supports spatial data in 1, 2, or 3 dimensions. max_entries may be any value greater than one and min_entries must be some value less than half of max_entries.
**void** GRT_Destroy(int32_t handle)

**Parameters**

handle  handle to an allocated GRT

As one might expect, GRT_Destroy is used to de-allocate a GRT and all of its data.

```c
struct grt_result{
  // number of results returned from query
  int32_t num_results;

  // ga handle of spatial database
  int32_t ga_h;

  // 2-D array of ga lo indices for results
  uint32_t (*lo)[2];
};
```

**void** GRT_Search(int32_t grt_h, double (*query)[2], grt_result_t *result)

**Parameters**

grt_h  handle of a GRT
query  range query to execute.
result  results returned from range query

GRT_Search is the primary function used to execute range queries on the R-Tree. Queries are accepted in the form of an array of lo and hi indices; one pair for each dimension of the spatial data. A query’s results are returned as a structure (shown below) that should be pre-allocated by the user application.
struct grt_record{
    // GA database storing the record
    int32_t ga_h;

    // GA lo indices of the record
    uint32_t *lo;

    // number of points
    uint32_t npoints;

    // [X,Y,Z] indices of points in the record
    double *point;
};

void GRT_Insert(int32_t grt_h, grt_record_t *record)

Parameters
    grt_h    handle of a GRT
    record   record to be inserted

GRT_Insert is the primary function used to insert records into and build a GRT. It may only be called during a GRT write phase and is called once for each record in the database.

void GRT_Begin_Write(int32_t grt_h)

Parameters
    grt_h    handle of a GRT

GRT_Begin_Write signals the beginning of a write phase. It is a collective call that will block until called on each node in the system.

void GRT_End_Write(int32_t grt_h)

Parameters
    grt_h    handle of a GRT
Figure 2.1: Global R-Tree nodes are converted into byte arrays and then stored in a 2-D Global Array. The first "node" in the Global Array is reserved for important global metadata.

Figure 2.2: A GRT Node contains 12 bytes of metadata followed by a series of index records. The node pre-allocates room for up to M index records, where M is the max_records parameter from the create function.

GRT_End_Write is the companion function to GRT_Begin_Write that signals the end of a write phase. Like its partner, it is also a collective call, requiring all nodes to complete their write operations before exiting the write phase. Note that while this phase write design was intended to support parallel inserts, the current API version only supports inserting from one process.

2.2 Parallel Access

In order to facilitate parallel availability of our R-Tree, we leverage the Global Arrays framework to create and maintain our R-Tree in array-form. R-Tree nodes are translated into byte arrays and then mapped to pre-allocated locations on the GA. Figure 2.1 illustrates this mapping. In the GA, rows are aligned to be evenly divisible by the size of a node. When a row is exhausted, newly created nodes are placed in the next row down until the end of the GA is reached. Currently, exhausting
the capacity of the GRT requires deletion of the R-Tree and complete re-construction of a new, larger GRT.

The first node-span of the GA is reserved for metadata that must be accessible by all nodes in the parallel system. Specifically, this data includes the location of the current root node and the location of the next location in the GA available to allocate for a new node.

Each node contains its own global metadata in addition to a series of index records, shown in Figure 2.2. A node’s metadata spans 12 bytes. The first 2 bytes are reserved for flags specific to the node, which are accessed using bit masks. Currently, the only flag in use is an indicator of whether the node is a parent or a leaf node. The next 2 bytes contain a counter of how many index records are being stored in the node. This counter cannot exceed the maximum entry parameter given at the GRT’s creation.

The last portion of the metadata contains a reference to the node’s parent. This ability to travel from child to parent is necessary to facilitate the efficient execution of queries. The alternative had the potential to require the entire tree be stored in the memory of a single node, which is counter-intuitive to the purpose of storing the R-Tree across distributed memory.

Finally, an index record follows the structure specified by Guttman [7] with the exception that Guttman’s tuple-indentifier is replaced by a GA reference. This allows parent nodes to reference the location of their children in our GA as well as enabling leaf node index records to reference the location of the spatial data records in the application’s GA database.
CHAPTER 3

GENERALIZED GLOBAL ARRAY CACHE

GRC is a caching system designed for the GA framework. It began as a specialized cache designed to improve the performance of a GA-based implementation of the einspline library used in the Qwalk application [22]. However, this implementation was optimized to take advantage of the distinctive access pattern and array type of Qwalk and not usable in the general case. In order to be useful to other applications, we generalized the einspline-specific implementation to support arbitrary array dimensionality, type, and access pattern.

3.1 Design Detail

GRC is designed much like the hardware cache on a modern CPU. Main memory’s address space is divided such that bytes are grouped into contiguous blocks of uniform size. When the CPU attempts to access a memory address, the cache loads a full block from main memory rather than just the requested address.

In GRC, we use a similar construct to decompose the GA into manageable chunks. Elements in the GA are divided along each index vector into $N$-dimensional ”bricks”. The key conceptual difference between a cache block and a GRC ”brick” is that a brick does not necessarily represent contiguous elements in the GA. The dimensions
Figure 3.1: The GA address space is divided into uniform bricks of the same dimensionality. Region requests on the GA may span multiple and partial bricks.

Figure 3.2: Bricks are uniquely indexed in row-major order, calculated using dimension stride.

of the bricks are specified by the user at the creation of the cache, allowing users to optimize the cache for the spatial locality of their application.

When an element is requested from the GA, GRC will load the entire brick in which the element is contained. Figure 3.1 shows a 2-D global array of size 8x8 divided into uniform bricks of size 2x2. While each brick is typically identical in size, it is possible that the brick dimension along an index vector does not divide evenly into the GA’s dimension along that vector. In this case, the bricks will be of the specified dimension up until the last brick on that vector, which will contain any remaining elements.

3.1.1 Brick Identification

Bricks are identified in the cache using a unique index based on their calculated brick indices. Figure 3.2 shows the corresponding brick IDs of a GA.
Algorithm 1: Dereference a brick ID into it’s corresponding brick index vector.

| Data: int remain |
| Data: int V[|N|] |
| remain ← ID |
| for i ← 1 to N do |
| v_i = [remain ÷ s_i] |
| remain = remain − v_i * s_i |

First, we calculate the brick index stride along each dimension vector. In the following equations, we denote $N$ to be the number of dimensions in our GA with the $N$th dimension representing contiguous elements. We also denote several sets: $S = \{s_1, s_2, ..., s_N\}$ where $s_i$ is the brick index stride along index vector $i$, $G = \{g_1, g_2, ..., g_N\}$ where $g_i$ is the GA dimension along index vector $i$, $B = \{b_1, b_2, ..., b_N\}$ where $b_i$ is the brick dimension along index vector $i$, and $R = \{r_1, r_2, ..., r_N\}$ where $r_i$ is the brick index range along index vector $i$.

$$\forall 1 \leq i \leq N r_i = \left\lceil \frac{g_i}{b_i} \right\rceil$$

$$s_N = 1$$

$$\forall 1 \leq i < N s_i = r_{i+1} * s_{i+1}$$

Next, when calculating the index of a specific brick, we do so based on the upper-left most GA element contained within the brick. If $X = \{x_1, x_2, ..., x_N\}$ such that $x_i$ is the index along index vector $i$ of the upper-left most GA element contained within a brick $m$, then:

$$ID_m = \sum_{i=1}^{N} \left\lfloor \frac{x_i}{b_i} \right\rfloor * s_i$$

Finally, when dereferencing a brick ID into it’s index vector, we simply start at the outer most dimension and divide the ID by the corresponding stride, $s_i$. Then, we subtract $s_i * Quotient$ to determine the remaining portion of the ID and repeat for the next dimension (see Algorithm 1).
3.1.2 Data Copy

The final major component of the cache is to rebuild the requested region from
the individual bricks. Once a brick has either been found in the cache or fetched
from the GA, the portion of the requested data that it contains is calculated and
then copied into the buffer given by the user. Figure 3.3 shows the complexity of
this operation. The cube shows a 2x2x2 brick of data indexed in 3 dimensions. The
linear blocks show a 3x3x4 region buffer given by the user. Each element of the brick
must be placed in the correct position, for the GA data the brick contains, within the
region buffer, requiring address translation from the brick’s addressing to the region
buffer’s.

3.2 Key Challenges

3.2.1 GA API Constraints

When GRC was first designed and tailored to the Qwalk application, having a
targeted application provided a great deal of flexibility in the cache’s API. However,
when designing the API for the generalized version, we decided to exactly mimic the
GA API. This enables quick and easy adoption of the cache in any application that
was built using GA. Calls to the GA API are simply replaced with calls to the GRC
API. Input parameters and expected output remain identical. This did introduce a
design challenge in that we restricted ourselves to operating with index vectors and a
single return buffer. Furthermore, we were restricted to dealing with index vectors in
order to interface with GA on the backend. Multi-dimensional index vectors presented
a design challenge in several feature implementations, as described in the following
sections.
3.2.2 Arbitrary Dimensionality

Being restricted to index vectors became a design challenge when we attempted to add support for arrays of arbitrary dimensionality. The original GRC began with support only for 4 dimensional arrays. When the dimensionality of our index vectors is known at compile time, we can iterate over them with simple loop nesting. This process becomes considerably more complex when our index vectors are not set until runtime. To support this, we developed an algorithm for iterating over the values in an array of unknown dimensionality (see Algorithm 2).

3.2.3 Modular Replacement Policy

In our testing, we applied two different cache replacement policies: Least Recently Used (LRU) and Adaptive Replacement Cache (ARC) [15]. In order to easily switch between the two, we implemented a module interface, defining several necessary functions that the policy must serve. The policy is determined by a flag parameter passed
Algorithm 2: Iterate over values of an array of unknown dimensionality

```
Data: int mov[N], start[N], end[N]
1 while flag = 0 do
2     for i ← N to 1 do
3         if i = N then
4             for mov[i] ← start[i] to end[i] do
5                 mov is the index vector of contiguous elements
6                 flag ← 1
7         else if flag = 1 then
8             flag ← 0
9             mov[i] ← mov[i] + 1
10            if mov[i] > end[i] then
11                mov[i] ← start[i]
12                flag ← 1
13         else
14             break
```

to the cache creation function, causing the appropriate module to be connected. Thus, at runtime, we are able to tell the cache which policy to use.

### 3.2.4 Custom Heap Management

Another challenge to supporting arbitrary dimensionality is that very few resources could be allocated statically, which meant a lot of interaction with the heap. Passing control to the operating system kernel is not ideal for performance, so when the cache is created, we allocate all of our memory up front for both the cache metadata and the memory that will be storing bricks. By doing this, we no longer need to go to the kernel during the performance sensitive portions of the application. However, it also means that we must manage the allocation of our brick memory ourselves.

We do this using a pair of hash queues that contain allocated and free bricks, respectively. When a brick is to be freed, we find the brick in the allocated queue and move it to the tail of the free queue. Likewise, when a brick is to be allocated, we simply move the head of the free queue to the tail of the allocated queue.
3.3 Write Support

GRC began with the intent that it would be strictly read-only (hence the name). We examined the potential use cases and concluded that the majority of our target applications execute in phases, writing all of their data to memory at once and then exclusively reading on that data. However, some do require periodic writes. Under the read-only model, the entire cache would have to be destroyed and recreated any time a write was executed on the GA. To alleviate this constraint, we implemented some rudimentary write support for the cache.

3.3.1 Coherence/Consistency Model

The primary benefit of the original read-only design was the lack of any necessary synchronization overhead. With the introduction of writes into the design, we must now consider the possibility of writes occurring on other nodes in the system. To ensure coherence, we implemented the write-through strategy, immediately calling back to GA upon writing to a region. We rely on GA’s own coherence model to ensure that subsequent reads from any node in the system are delivered up-to-date data.

However, the task still falls to us to communicate to each node’s cache that bricks have been written. We avoid synchronization overhead by leveraging the phase-based access model. We introduce two new collective calls to the API that allow the system to enter and exit a ”write phase”. During a write phase, no reads are allowed on the cache. Likewise, no writes are allowed on the cache outside of a write phase. During a write phase, we pass write operations through to the GA subsystem and collect information on which cache bricks are being written.
When the call is made to exit the write phase, we distribute this information throughout the system such that each node knows every brick that was written to by any node. Each node then checks each brick in their cache against this list and evicts any brick that has been written. The evicted bricks are brought back in on the next read that accesses their data.

### 3.3.2 Bloom Filters

Bloom Filters are a data structure used to efficiently store set information with a possibility of false positives [1]. They are widely used in database and networking applications [3, 12]. In our case, we leverage bloom filters to efficiently track which bricks have been written during a write phase. By storing this information in a bloom filter instead of a more simple data structure (such as a byte array), we significantly reduce the number of bits required. This significantly affects the scalability of the cache during write phases due to the all-to-all reduction of each system node’s dirty brick information. As more nodes enter the system, reducing the size of our data structures produces a significant impact on network traffic.

In the cache, false positives mean that we evict bricks that have not actually been written to during the write phase. For a sufficiently small rate of false positives, evicting clean bricks can be an acceptable performance hit. Evicting a clean brick means one additional cache miss, which can be offset by the performance gains from the reduction in network traffic gained by using bloom filters.

For the first version of GRC’s write support, we are using 2 hash functions (SAX and SDBM) with our bloom filter. From Broder and Mitzenmacher [3], if we let $p$ be the probability of a bit still being 0 after all items in the set have been inserted into
the bloom filter, we know that the probability of a false positive, \( f \), can be minimized with the asymptotic approximation:

\[
p = e^{-kn/m}
\]

\[
f = (1 - p)^k
\]

\[
0 = -\frac{m}{n} \ln(p) \ln(1 - p)
\]

where \( k \) is the number of hash functions, \( n \) is the number of elements in the set we are inserting into the filter, and \( m \) is the number of bits used by the bloom filter to store the set information. Thus, we have the result that \( f \) is minimized when \( p = 1/2 \) and can simply solve for \( m \) to determine the size of our bloom filter. In our cache, \( n \) shall be the total number of bricks in the GA:

\[
n = \frac{ga\_size}{brick\_size}
\]

\[
m = -\frac{2n}{\ln(1/2)}
\]

Using this approximation, we can achieve a false positive probability that is asymptotically close to 0. The alternative solution we implemented instead of bloom filters was to use a simple byte array, with each element representing a brick of the GA. Since we must only set the byte to 1 to represent a written brick or 0 to represent a clean brick, 7 bits are wasted for each brick. If we compare the bits necessary in this solution with the bits necessary using bloom filters:

\[
-\frac{2n}{\ln(1/2)} \ast \frac{1}{8n} \approx 0.36
\]

(3.5)

By using bloom filters, we must use nearly a third the bits of using a basic byte array.
Algorithm 3: Trace generation algorithm

**Data:** $N, miss\_\text{percent}, remote\_\text{percent}$

1. fill the cache with bricks
2. remember a brick that is local and one that is remote
3. $miss\_\text{rate} \leftarrow 1 \div miss\_\text{percent}$
4. $remote\_\text{rate} \leftarrow 1 \div remote\_\text{percent}$
5. $k \leftarrow 1 \ l \leftarrow 1$
6. for $i \leftarrow \text{cached\_bricks} \text{ to } N$ do
   7. $miss \leftarrow (i \geq k \ast miss\_\text{rate})$
   8. $remote \leftarrow (i \geq l \ast remote\_\text{rate})$
   9. if $miss = 1$ then
      10. if $remote = 1$ then
          11. find new remote brick not cached
          12. $l \leftarrow l + 1$
      13. else
          14. find new local brick not cached
          15. evict lru brick
          16. emit index of new brick
          17. $k \leftarrow k + 1$
   18. else if $remote = 1$ then
      19. emit index of remote brick
      20. $l \leftarrow l + 1$
   21. else
      22. emit index of local brick

3.4 Experiment Results

To test the overall performance of the cache, we examined two different approaches. The first was to modify the GA-based Qwalk implementation to use GRC and measure its performance with and without the cache. For the second approach, we simulate access patterns that produce different conditions in the cache and measure its performance under these conditions.

3.4.1 Qwalk Case Study

Design

In this experiment, we modify the real-world application, Qwalk, to use our GRC cache. The modifications required were minimal due to our API design. For our input, Qwalk creates one, 4-dimensional global array of size $106 \times 106 \times 106 \times 432$ with elements of type "complex double". For our brick size, we use $1 \times 1 \times 2 \times 432$ bricks.
in order to fully take advantage of Qwalk’s access pattern. We use a cache size of 128MB and the ARC replacement policy. Qwalk reports the average time spent on each evaluation for each node. For each test, we run one process of Qwalk on each node and record the average evaluation time from all nodes. We test Qwalk on 4, 8, 16, 32, 64, and 128 nodes.

Results

The results, shown in Figure 3.4, show that Qwalk performs better with vanilla GA than with our GRC up to about 100 nodes. As the number of nodes increases, network communication becomes a bigger bottleneck for Qwalk, causing GA to scale poorly on more and more nodes. While GRC does initially suffer from the extra overhead, it scales significantly better in the presence of increasing network traffic.

3.4.2 Simulation Study

Design

In this experiment, we wanted to isolate two key factors that influence our cache’s performance relative to GA: hit rate and the rate at which remote data is accessed.
Hit rate is important because if we are experiencing misses too frequently, then we are not gaining any benefit from the overhead of storing bricks in the cache. The other factor stems from GA’s architecture. It stores a portion of the array on each node, causing each node to have local access to a portion of the array. If each node must only access the portion of the array that it contains locally, the node is already accessing local memory. Our cache stores bricks in local memory as well, so in such a use case, the cache only produces overhead.

In order to isolate these factors, we simulate access patterns that produce the desired hit rate and remote access rates in GA and in the cache. We do this by generating traces according Algorithm 3. For these tests, we use a 2 dimensional global array of size 1000x1000 and of type ”integer”. We choose our brick size of 50x50 and a cache size of 1MB in order that the total number of bricks in the GA is approximately 4 times as many as can be held in the cache. Each trace runs for 10000 iterations. After all iterations, the test gathers the hit rates, remote access rates, and
Table 3.1: Results from the hit rate experiment

Table 3.2: Results from the remote access rate experiment

For these tests, we use 2 nodes with 1 process per node.

Results

Figure 3.5 and Table 3.1 show the results from the hit rate tests. Figure 3.6 and Table 3.2 show the results from the remote access rate tests. For the hit rate tests, we see that the cache is highly dependent on hit rate, requiring approximately 50% of requests to be hits in order to outperform vanilla GA with a remote access rate of a constant 50%. Our simulated access pattern does not produce significant variance in performance between LRU and ARC, likely due to how we emit the same index for local and remote hits each time unless that index happens to be evicted on a
miss iteration. Under these conditions, we observe the expected result that LRU outperforms ARC due to ARC’s greater book-keeping overhead.

The remote access rate tests were performed using a constant 80% hit rate. Here we see that the remote access rate does not have as great an effect on GA or GRC’s performance as the hit rate. This may be an indicator of a design flaw in this experiment in that we may not be transferring large enough amounts of data to fully saturate the network bandwidth between the nodes. We may also be seeing some batch processing of network packets, accounting for the increases in GA performance on some increases of the remote access rate. However, we do see GA’s performance becoming increasingly erratic as the remote access rate increases, while GRC’s performance remains mostly stable. This result compliments the Qwalk test results that show GRC to scale significantly better than GA as network communication becomes a greater portion of the access pattern.
CHAPTER 4

PERFORMANCE RESULTS

4.1 Design

We tested our parallel R-Tree implementation by constructing a tree using NCAR’s GIS data. We entered the data into a global array-based database of 32678 records and constructed the index using our GRT library. We then chose a randomized set of 1048576 range queries. Finally, we distributed the set of queries amongst all processes in the test such that each process executed $\frac{1048576}{N}$ queries.

Each query returned a number of results from the R-Tree and we recorded the query execution time along with the number of results returned from the query. By executing so many queries, we were able to mostly eliminate probabilistic anomalies due to the randomization, as well as showing the impact of the cache upon subsequent accesses to the same portions of the tree. We attempted to show the impact of the cache further by running the tests both with the cache enabled and without.

Our tests were executed on the homogenous Oakley cluster at the Ohio Supercomputer Center running Intel Xeon processors. We utilized 1, 2, 4, 8, 16, 32, and 64 processors and executed 1 process per node. We chose the ARC cache replacement policy for these tests because we believe ARC to be better suited to the access pattern
Figure 4.1: NCAR GIS test throughput as the number of processors increase with cache disabled (left) and the cache enabled (right)

Figure 4.2: NCAR GIS test results from 1 processor run with the cache disabled (left) and the cache enabled (right)

of our queries than LRU. In future tests, we will compare the two policies to see the true difference in their impact on our system.

4.2 Results

Figure 4.1 shows the throughput results from our tests in terms of results per $\mu$sec. The left hand figure shows the throughput with the cache disabled, using vanilla GA.
The right hand figure shows the throughput with the cache enabled. We can see that GA does better without the cache on a single process run, but loses some throughput on 2 processes. This is likely due to the current lack of load-balancing causing a single node to house most, if not all, of the relatively small R-Tree index. GA’s performance improves steadily as the number of processes is increased. However, when we introduce the cache, we see a much more linear improvement in throughput.
as we increase the number of processes and we do not see a drop when going from 1 to 2 processes. This shows that the cache is significantly reducing the impact of network communication on the performance of our R-Tree even without proper load balancing in place.

Figures 4.2 - 4.8 show the plots of the results from each individual test. They show the expected result, that our R-Tree’s query execution time is directly correlated
Figure 4.7: NCAR GIS test results from 32 processor run with the cache disabled (left) and the cache enabled (right)

Figure 4.8: NCAR GIS test results from 64 processor run with the cache disabled (left) and the cache enabled (right)

with the number of results returned by the query. As with the throughput test, we also see that our cache has a significant impact on this correlation. GA’s query performance is highly consistent with uniform variance from the mean. However, query performance degrades much more quickly than with the cache enabled. The cache’s query performance varies to a greater degree than with vanilla GA. However, the mean query execution time is considerably lower and performance degrades more
slowly as the number of query results increase. It is worth noting that some plots show wild variance in the mean toward the top of the x-axis. This is likely due to having relatively few data points in this portion of the plot, causing anomalies to show.
CHAPTER 5

FUTURE WORK

While preliminary testing has shown promising potential for our approach to a parallel R-Tree implementation, there is still much room for improvement. The cache provides some load balancing due to its data-replication trait, but implementing a proper load balancing scheme could still produce significant improvements to performance. Similarly, re-distributing the tree across our array structure holds potential gains from the cache if we are able to do so in such a way that maximizes spatial locality.

Another potential performance improvement would be to follow the work done by Luo et al. [13] in order to parallelize queries on our GRT. Currently, individual queries are executed in serial within a process of the user application. It is likely to be possible to achieve significant gains to throughput on the overall system by leveraging multi-core GPU and CPUs to locally parallelize queries within a process.

One of our design goals at the start of this project was to support parallel insertion and deletion from the tree. This feature turned out to be extremely difficult to implement along with our array-based design and thus was put on hold in the interest of time and more pertinent features. However, it is still a goal of the project to implement this feature and thus will be considered for future work.
Modern databases are scaling to very large sizes and thus the size of their indexes also are becoming very large. One possible technique for increasing the efficiency and scalability of an index structure is to implement clustering of records.

The current version of GRT supports spatial data in 1, 2, or 3 dimensions. It is desirable to expand this support to arbitrary dimensionality. The current version is also missing an API call for deletion of records from the R-Tree.
CHAPTER 6

CONCLUSIONS

In this thesis we have introduced a novel approach to parallel R-Tree design in that we have constructed our R-Tree as an array structure. The benefits of this approach enabled us to leverage a generalized cache to dynamically adapt the distribution of our tree data to the access pattern of the user application.

We have implemented our parallel R-Tree and shown promising throughput using our prototype. There is still much room for improvement, in particular with the introduction of load balancing and local parallelism.
BIBLIOGRAPHY


