A LIBRARY OF LARGE-SCALE DISTRIBUTED SPATIAL DATA STRUCTURES

By

Pieter Hooimeijer

A Thesis Submitted to the

Faculty of Colgate University

in Partial Fulfillment of the

Requirements for

HIGH HONORS IN COMPUTER SCIENCE

Approved:

____
Toshiro K. Ohsumi
Thesis Adviser

Colgate University
Hamilton, New York

April 2006
(For Graduation May 2006)

CONTENTS

LIST OF FIGURES iv

LIST OF LISTINGS v

ABSTRACT vi

1. Introduction and Background 1
   - 1.1 Motivation 1
   - 1.2 Outline 2
   - 1.3 Spatial Data Structures 2
     - 1.3.1 Recursive Decomposition 2
     - 1.3.2 Quadtrees 3
   - 1.4 C++ Template Techniques 5
     - 1.4.1 Generic Programming 5
     - 1.4.2 Template Metaprogramming 6
     - 1.4.3 Using Templates to Improve Efficiency 8
   - 1.5 The MPI Library 8

2. Library Design and Public Interface 10
   - 2.1 The Templated Composite Pattern 10
   - 2.2 Public Interface 12
   - 2.3 Private Classes 15

3. Library Implementation 17
   - 3.1 Data Structures 17
     - 3.1.1 Top-level MPI data structure 17
     - 3.1.2 Bottom-up MX Quadtree 21
   - 3.2 C++ Template Techniques 22
     - 3.2.1 Tag Classes using a Template Typedef 23
     - 3.2.2 Type-safe Constructor Parameters 25
     - 3.2.3 Efficient Callbacks through MPI 29

4. Results and Future Work 33

LITERATURE CITED 35

APPENDICES

A. An example program 37

LIST OF FIGURES

1.1 A *point* quadtree with corresponding spatial representation. 3

1.2 An *MX* quadtree with corresponding spatial representation. 4

2.1 The Composite design pattern 10

3.1 Morton Numbering 21

LIST OF LISTINGS

1.1 Generic programming example: max() 6
1.2 Recursive template example: Factorials 7
3.1 Example: Using MPI 19
3.2 Using tag classes to define a type 23
3.3 A Tag Class 24
3.4 Recursive templated tuples 27
3.5 Predicate inlining across processes 31
A.1 Example: Using DGrid 37

ABSTRACT

DGrid is a generic C++ library that addresses querying of spatially clustered point data. The library is based on the Templated Composite design pattern, which is a novel adaptation of the Composite pattern. DGrid uses template metaprogramming techniques to improve performance and flexibility, and to allow static type checking. The library provides support for queries on distributed data. Use cases are provided as evidence for the library's practical merit.

CHAPTER 1
Introduction and Background

1.1 Motivation

This thesis presents DGrid, a library of data structures designed for very large-scale sets of point data. The library was designed and implemented according to the following requirements:

- flexibility for trade-offs between query time and construction time – Many data structures are optimized for static data, so that inserting and deleting items is relatively expensive. An important goal for the DGrid library was to provide a range of trade-offs between the cost of updates and the cost of queries, making the library suitable for highly dynamic data.

- generic design – The DGrid library is designed to be minimally dependent on a specific data structure. Academic work frequently focuses on implementing a specific data structure (e.g. the Approximate Nearest Neighbor library [4]). DGrid allows any two data structures to be combined, provided they have compatible operations.

- scalability up to very large dynamic datasets – DGrid uses the *Message Passing Interface (MPI)* library to allow data structures to be distributed across multiple concurrent processors.

These requirements are based on a specific type of biology simulation [8] that involves `hosts` (static entities) and `vectors` (dynamic entities that tend to cluster around hosts). A typical example of this involves flowers as the hosts and bees as the vectors. Another applicable type of problem is within transport systems, such as air traffic control (where airplanes cluster around airports).

1.2 Outline

The remainder of this chapter introduces the theoretical and technical concepts used throughout the paper. Chapter 2 offers a high-level overview of the library. Section 2.1 describes the design pattern that underlies most of the library. Section 2.2 provides the public interface of the library, while section 2.3 details several classes that are used internally.

Chapter 3 discusses the library's implementation. Section 3.1 provides additional details for the data structures implemented by the DGrid library. Section 3.2 goes into detail about the techniques used to implement several specific features. Chapter 4 presents our results and discusses possible future work. Finally, the Appendix provides an additional example program that uses the DGrid library.

1.3 Spatial Data Structures

The term *spatial data structure* applies to any data structure that is designed to hold points, lines, rectangles, etc. Spatial data structures can be optimized for a variety of search operations, for example:

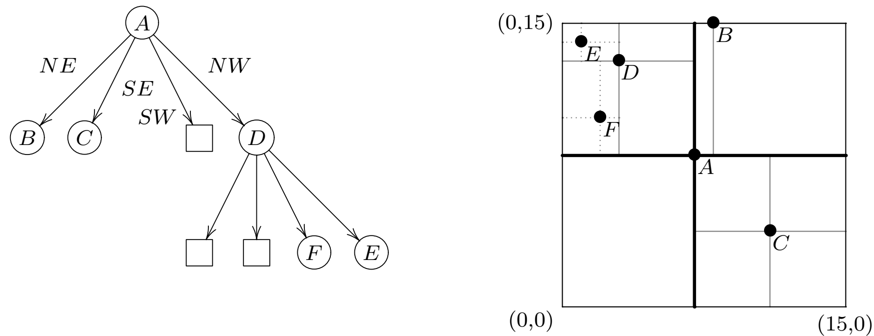
- range query – Find all points within a given rectangle.
- nearest neighbor query – For some set of points, find the nearest neighbor to each.
- intersection query – Given a set of lines or shapes, find all 'collisions.'

Typical trade-offs for spatial data structures are between query complexity, insertion and deletion complexity, and memory requirements. Samet [11, 10] provides an overview of the most common spatial data structures.

1.3.1 Recursive Decomposition

Many spatial data structures are based on the idea of dividing the space into smaller areas. Search operations can then be limited to the relevant 'bucket'

Figure 1.1: A *point* quadtree with corresponding spatial representation.



only, making this method slightly more efficient than searching the entire space. Bucketing can also save memory if we only instantiate the buckets that actually have items in them. This is called *tiling*. The trade-off is that insertion and deletion will be slightly more complex—the data structure needs to compute the correct bucket every time.

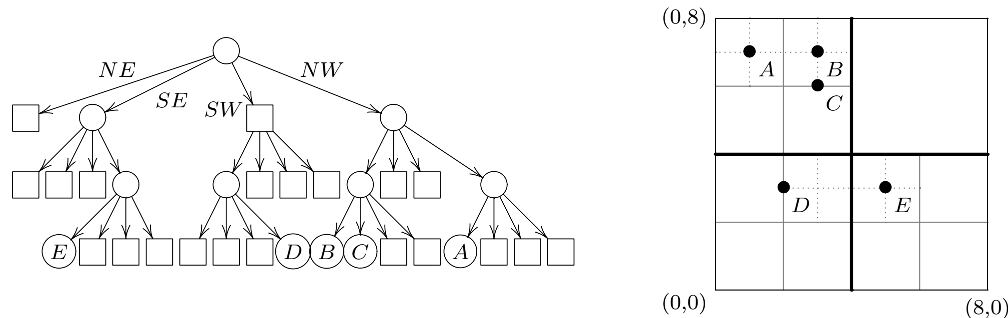
If we further divide each bucket into smaller 'sub-buckets,' the result will be a tree structure of increasingly fine-grained subdivisions. This is called *recursive decomposition*. The DGrid library is designed around this principle.

1.3.2 Quadtrees

The quadtree [11] is a good example of a data structure that is based on recursive decomposition. It is a two dimensional tree-based data structure that is similar to a binary search tree. Each node has four children: *NE*, *SE*, *SW*, and *NW*. There are several types of quadtree that are appropriate for point data. This section discusses two of these: the *point quadtree* and the *matrix quadtree*.

Figure 1.1 shows a *point quadtree* after the following insertions: A $(7,8)$; B $(8,15)$; C $(11,4)$; D $(3,13)$; E $(1,14)$; F $(2,10)$. Each node represents a data point, and points are added by a top-down search. This means that the shape of a point quadtree depends on the insertion order of its points, and that the tree may need balancing. Balancing and deletion are complicated by the fact

Figure 1.2: An *MX* quadtree with corresponding spatial representation.



that data is stored in the internal nodes of the tree. These issues are similar to the ones found in binary search trees, but the implementation is much more complicated.

There are several other types of quadtree. The implementation in DGrid is a *matrix (MX) quadtree*. Figure 1.2 shows an 8×8 MX quadtree for the following points: A $(1,7)$; B $(3,7)$; C $(3,6)$; D $(2,3)$; E $(5,3)$. In an MX quadtree, the data points are stored in leaf nodes only. Each intermediate node simply splits the space into four equally sized quadrants. As a result, the shape of the tree does not depend on the order in which items are inserted, and no balancing needs to be done.

The MX quadtree has some limitations compared to the point quadtree. A point quadtree can have any arbitrary size, since the nodes hold the coordinates that they represent. An MX quadtree must know its size in advance, and the size is limited to powers of two (i.e. $2^n \times 2^n$). Also, because data is stored in the leaves, search operations require $\Theta(h)$ node traversals, where h is the height of the tree, which is dependent on the size of the space. For point quadtrees, the height of the tree depends on the number of items only.

The main advantage of the MX quadtree that its shape is predictable—the same leaf node always represents the same coordinate. The quadtree implementation in DGrid uses this property to ensure efficient insertion and deletion. Section 3.1.2 describes this implementation in more detail.

1.4 C++ Template Techniques

C++ templates are a powerful and complex language feature. Templates are Turing complete [15], which means that they can be used as a (crude) functional programming language that is executed at compile time. In the following sections, we will outline the concepts that are most relevant to the DGrid library. Vandevoorde and Josuttis [14] give a more complete explanation.

1.4.1 Generic Programming

The primary reason templates were introduced into C++ was to support *generic programming*. Consider an object oriented language without templates. A simple `List` data structure might have the following methods:

- `void add(Object item)` – Adds an item to the list.
- `Object get(int index)` – Retrieves an item from the list.

Suppose a `List` contains objects of type `Item`. The following code retrieves the fifth element from the list:

```
List myList;
// Omitted: insert items
Item myItem = (Item)myList.get(4);
```

Since `get()` returns an `Object`, it is necessary to downcast the return value to an `Item`. A downcast might throw an exception if we accidentally inserted a different type of object into `myList`. The problem is that this is a runtime error; inserting the 'wrong' type of object will not yield any compilation errors.

If `List` were implemented using templates, the compiler would be able to catch this type of error. The code would look like this:

```
List<Item> myList;
// Omitted: insert items
Item myItem = myList.get(4);
```

Because a `List<Item>` can only contain instances of `Item`, there is no need to downcast the return value of `myList.get(4)`; it is already an `Item`. Adding some

Listing 1.1: Generic programming example: max()

```
#include <iostream>

template <typename ItemType>
const ItemType& max(const ItemType& a, const ItemType& b) {
  return (a < b) ? b : a;
}

int main(int argc, char** argv) {
  int a    = max(6, 5);
  double b = max(5.0, 5.1);

  std::cout << a << std::endl
            << b << std::endl;
}
```

non-`Item` object to `myList` is simply not possible, because it would result in a compile time error. The only way to achieve this without templates is by manually coding every specialization of the `List` class (e.g. `ObjectList`, `ItemList`, etc.), which is typically not feasible.

Generic programming applies to algorithms as well. It attempts to make algorithms minimally dependent on their data. Listing 1.1 demonstrates this principle. The templated `max` function returns the maximum of two items for any type that has the less than (<) operator defined, including any user defined classes. In a strictly object oriented language, user defined classes would have to implement a common base class or interface (e.g. `Comparable`).

Musser [6] offers a more complete description of generic programming. Several well known libraries make extensive use of generics. This includes the C++ Standard Template Library (STL) [5, 7, 12] and the Boost libraries [3].

1.4.2 Template Metaprogramming

C++ templates can be used to perform computations at compile time. Listing 1.2 demonstrates this by using templates to calculate factorials. The `fact` template is evaluated recursively, so that `fact<5>::value` = `5 * fact<4>::value`

= `5 * 4 * fact<3>::value`, etc. This means that `fact<5>::value` is equivalent to writing `120` as a literal. The `value` member is a **`static const`** so that the compiler can ensure that there is a constant, compile time, value for every `fact<N>`.

The use of templates for computation is generally referred to as compile time programming or *template metaprogramming*. Listing 1.2 is a trivial example of template metaprogramming, since it could easily be replaced by regular C++ without templates. In theory, this is true of any code that uses templates. For example, instead of a templated `List<Item>` class (as described in the preceding section), we could implement a separate `ListItem` class for every type of `Item`. It should be clear, however, that using templates would take much less time.

For more complicated uses of templates, it becomes infeasible to hand-code the equivalent non-templated classes. A good example is Veldhuizen's Blitz++ library [16], which uses template techniques to provide very efficient array data structures.

Listing 1.2: Recursive template example: Factorials

```
#include <iostream>

// default template (inductive case)
template <int N>
struct fact {
  static const int value = N * fact<N - 1>::value;
};

// specialization (base case)
template < >
struct fact<1> {
  static const int value = 1;
};

int main(int argc, char ** argv) {
  std::cout << fact<5>::value << std::endl; // result: 120
  return EXIT_SUCCESS;
}
```

1.4.3 Using Templates to Improve Efficiency

Templates are frequently used to improve the runtime efficiency of the resulting machine code. Because template parameters are evaluated at compile time, the compiler may be able to inline function calls. This eliminates the cost of the actual function call in the resulting code, at the cost of increasing the size of the code (because the method body is duplicated in the calling code). This may be a good trade-off when the inlined code is executed frequently.

Object oriented programming encourages the use of polymorphism through inheritance. In terms of performance, this compounds several issues:

- Methods will frequently be `virtual` to ensure that the correct function call is performed at run time. Virtual function calls typically cannot be analyzed at compile time, which means they cannot be inlined.
- Virtual function calls are more expensive at run time than regular function calls [1]. Virtual calls are typically handled through a virtual method table (*vtable*). When a virtual function is called, the C++ runtime performs a lookup to determine the correct function. This lookup adds to the cost of the function call.

This means that high performance applications should avoid frequent virtual calls to short functions. One way to achieve this is to use templates instead of virtual functions. For example, the calls to `max()` and the call to `operator<()` in listing 1.1 could all be inlined. A strictly object oriented implementation (using a `Comparable` base class) would not allow the calls to `operator<()` to be inlined.

1.5 The MPI Library

DGrid uses the MPI library to allow it to run on multiple concurrent processors. MPI is a low-level library for C/C++ and Fortran that centers around basic `send` and `receive` operations. The MPI library assigns a unique *rank* to each process, which is used explicitly when sending and receiving messages. Programs

that use MPI can run on a variety of architectures, ranging from parallel (super) computers to distributed networks of commodity PCs. Pacheco [9] provides a good introduction to MPI.

CHAPTER 2
Library Design and Public Interface

We implemented DGrid in C++ using template techniques to improve both performance and usability. This chapter focuses on the design decisions that led to the library's current form, while Chapter 3 discusses the implementation in more detail.

2.1 The Templated Composite Pattern

The design of the DGrid library is an adaptation of the Composite design pattern. The Composite pattern is a structural pattern described by Gamma *et alia* [2]. Figure 2.1 shows a UML diagram. The Composite pattern allows the client to treat a tree structure of `Component` objects as a single entity. Such a tree structure is built by nesting instances of `Composite`.

This design has some important consequences. All `Leaf` and `Composite` classes must implement the operations specified in the `Component` interface. These operations must 'make sense' when applied to a tree structure of objects. If the client is allowed to add children to a `Composite`, then the `Component` interface

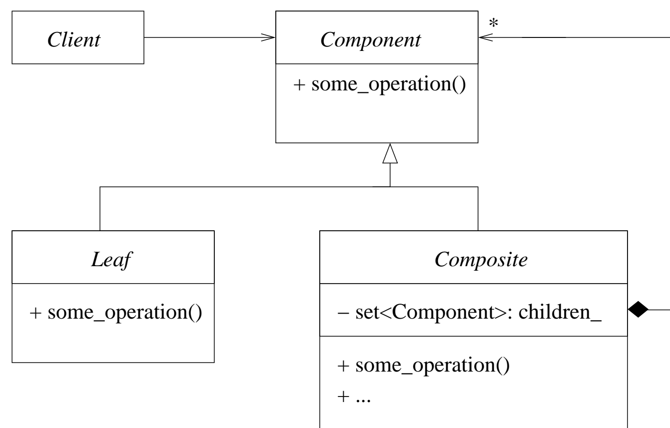


Figure 2.1: The Composite design pattern

needs to have some `add_component()` method. This means that `Leaf` will also need to implement `add_component()`, even though it does not have children.

The DGrid library uses templated code instead of inheritance, so the `Component` interface is implicit. Each data structure acts as a `Composite`, taking another data structure as a template parameter. The leaf class is `location`, which only takes an item type as a template parameter.

Each data structure manages its own children, but the user specifies the children's data type. This is done using *tag classes*. Each tag class (e.g. `full_grid_tag`) represents a specific data structure, and tags can be nested to represent nested data structures. Section 3.2.1 shows how tags work (and why they are used in the DGrid library).

A typical instantiation might look as follows:

```
using namespace dgrid::tags;
typedef dgrid::dgrid<item, partial_grid_tag<
                            full_grid_tag< > > > bucket;

bucket a(0,0, 127, 127, tiles(64,64) <<
                        tiles(1,1));
```

This creates a tiled data structure of two levels. The top level simply divides the space into areas of size 64×64. Each area is instantiated only when needed (and only needs to be searched if instantiated). The areas themselves are handled by `full_grid`, which instantiates 64^2 locations upon construction. This is an example of a simple 'bucketing scheme' that uses tiling, as described in section 1.3.1.

An important consequence of the Templated Composite pattern is that each nested data structure is a type, which means it must be known at compile time. This means the user cannot build more levels at runtime, for example. On the other hand, this setup may allow the compiler to optimize the code extensively. It also allows for thorough type checking of the parameter list (as described in section 3.2.2).

2.2 Public Interface

The interface of the `dgrid` class depends on which data structure handles the top level. For example, using the MPI data structure adds several methods that do not exist when using the other data structures. There is, however, a central set of methods that every data structure accepts. They are as follows:

- `dgrid(int x0, int y0, int x1, int y1, Argument args)` – The constructor instantiates the data structure described by the template parameters. The coordinates are inclusive, and it is required that $x0 < x1$ and $y0 < y1$. The `args` parameter is a list of parameters for each of the data structures, separated by stream operators (<<). The following arguments currently exist:
  - `dgrid::tiles(int width, int height)` – This specifies the tile size for the current data structure's immediate children. The tile size must divide the space exactly. The last argument in the list will typically be `tiles(1,1)` (for individual locations). This argument is valid for data structures described by `full_grid_tag`, `partial_grid_tag`, and `quadtree_tag`.
  - `dgrid::mpi_tiles(int width, int height)` – This specifies the tile size for the MPI data structure (`mpi_grid_tag`). Each tile is assigned to a separate process.
  - `mpi_mapping(int * processes)` – This is an optional argument to the MPI data structure. It must be specified after the `mpi_tiles` argument, separated by a dot (`.`). The parameter is a list of process ranks that should be used to handle individual tiles. For example, if the MPI data structure is used for a 128×128 space, using tiles of size 64×64, then `processes` should be an array of four integers. The tiles are considered in row-major order.
- `int x0() const`–`int y1() const` – These methods return the bounds of the current data structure.

- long count() const – Returns the number of items currently held by the data structure.

- int tile_width() const, int tile_height() const – These methods return the tile width and height for the data structure. Note that these are the values for the top level data structure only. There is no way to obtain this data for the inner data structures.

- void get(const int x, const int y, ContainerType & container) const – The get() method retrieves all items for a specific location, and adds them to the container. The container must provide a push_back() method, as is the case for e.g. std::list and std::vector. If item is the type of data held by the data structure, then the container must be of type container<item *>.

- void get_range(const int x0,const int y0,const int x1, const int y1, ContainerType& container) const – The get_range() method retrieves all items in the range specified by the coordinate pair (which is inclusive). The container must provide push_back(). If item is the type of data held by the data structure, then the container must be of type container<item *>.

  The get_range() method can take one or two *predicate functions* to limit the search results. Predicates are user defined functions that return a bool. Consider the following example:

```
bool even_x_pred(int x, int  y) {
  return x % 2 == 0;
}
bool even_item_pred(long & n) {
  return n % 2 == 0;
}
// Omitted: instantiate data structure and add items

std::list<long *> even_x_list;
std::list<long *> even_id_list;
std::list<long *> even_both_list;
```

```
a.get_range <even_x_pred>   (0,0,5,5, even_x_list);
a.get_range <even_item_pred>(0,0,5,5, even_id_list);
a.get_range <even_x_pred, even_item_pred>
                            (0,0,5,5, even_both_list);
```

The use of predicates with the MPI data structure requires extra steps; this is described below.

- `void insert(const int x, const int y, ItemType & item)` – Insert `item` at (x, y). Duplicate insertions are ignored silently.

- `void remove(const int x, const int y, ItemType & item)` – Remove `item` at (x, y). Returns without errors if the item does not exist at that location. Dynamically allocated items will not be freed automatically.

Some of the methods listed above will work differently for the MPI data structure; section 3.1.1 discusses these differences in detail. The MPI data structure (represented by `mpi_grid_tag`) also has several additional methods. These are necessary because the MPI data structure will perform different activities on different processors (i.e. accepting requests vs. handling them).

- `is_active()` – This method returns `true` for processes that will be responding to user requests. These processes should call `start()`, allowing the data structure to start receiving messages.

- `start()` – Starts the event loop. This method blocks until some other process calls `shutdown()`.

- `register_predicate< >()` – This method 'registers' a predicate function for future use. It must be called in all processes, in the same order for each process. Section 3.1.1 demonstrates how to use this feature, while section 3.2.3 explains how it works.

2.3 Private Classes

The interface description in the preceeding section applies to the data structure classes `basic_grid_impl`, `basic_mpi_grid_impl`, and `quadtree_linear_map_impl`. Their interface is exposed through the `dgrid` class (section 3.2.1). This section describes several additional classes that DGrid uses internally:

- `dgrid_stream` – This class handles serialization of data for the MPI data structure. It declares a large number of overloaded stream operator methods so it can handle objects and primitive types uniformly. User code should always use the stream operators.

  The class currently uses a fixed buffer size, defined as DGRID_MPI_DATA_MSG_LENGTH in dgrid_stream.h. The buffer itself is an array of `char`. By default, the class creates its own buffer. Alternatively, a buffer can be provided to the constructor (e.g. to deserialize an existing buffer). If the class is used with its own buffer, then data can be sent using the following methods:

  - `void send(int procid, int tag, SourceType & data)` – Serialize and send `data` to process with rank `procid`, using the provided tag.
  - `void receive(int procid, int tag, TargetType & data)` – Receive serialized dat from the process with rank `procid` using the provided tag. Deserialize the buffer into `data`.

  `SourceType` and `TargetType` are templates for their respective methods only, so that the same `dgrid_stream` instance can be used to send data of different types.

  When the stream class is used with an external buffer, the buffer can be sent and received without using the stream's methods. This feature is used by the MPI data structure, because it uses different tag numbers to denote different types of requests.

- `morton_table` – This is a lookup table for Morton numbers. It is used internally by the quadtree data structure (section 3.1.2), and is not exposed to the client. The lookup table is expanding; it grows to match the largest request. The constructor takes an `unsigned int`, which is the requested size of the table. Internally, the lookup table is `static`, so all instances of the class share the same data. The only other public method is `unsigned long operator()(unsigned int x, unsigned int y)`, which retrieves the Morton number for that coordinate.

  The Morton table performs bit operations (shifts) to interleave coordinate bits. This means the code depends on the endianness of the machine. The corresponding `define`s are BIG_ENDIAN and LITTLE_ENDIAN, the latter being the default.

CHAPTER 3
Library Implementation

3.1 Data Structures

The DGrid library currently implements four data structures, defined by the following tag classes:

- `full_grid_tag` and `partial_grid_tag` – These data structures are implemented by `basic_grid_impl`, and use a two-dimensional array to store data structures. The 'partial' variant instantiates and frees its slots as needed to minimize memory overhead. The full grid instantiates all of its slots when it is constructed.

- `quadtree_tag` – This is a bottom-up MX quadtree, described in more detail in section 3.1.2.

- `mpi_grid_tag` – This distributed data structure is designed for applications that use the MPI library. It is described in the following section.

3.1.1 Top-level MPI data structure

The MPI data structure has essentially the same interface as the other data structures in DGrid. However, it is not designed to 'hide' the fact that the program is using MPI. This has several consequences:

- The user is responsible for setting up MPI before using an MPI-dependent data structure. This includes calling `MPI_Init()` and `MPI_Finalize()` at the appropriate times, for example.

- Users are required to 'register' any predicate functions they wish to use (see section 3.2.3). This must happen in the same order on all processes.

- Unlike the other data structures, the MPI data structure is *not* 'ready for use' after construction. The user is required to call `start()` in each process that will be handling requests. An `is_active()` method is provided; it returns `true` for all processes that should call `start()`.

- The MPI data structure must be shut down (using `shutdown()`) before the program can terminate. A `shutdown()` request is broadcast to all processes that called `start()` previously. From the user's point of view, the `start()` method blocks until `shutdown()` is called in some other process.

- Because the data is distributed, data items can no longer be compared by reference. This means that the user has to provide the library with a method of comparing two data items. Also, since items need to be copied between processes, the data has to be serializable.

Listing 3.1 contains a simple example of how the MPI data structure can be used. The `item` class uses a unique identifier (`id_`) to allow comparison (line 9). It also specifies `serialize()` and `deserialize()`, both of which take a `dgrid_stream` parameter. These methods are required, in addition to a default constructor (line 13). The stream methods are called whenever an object needs to be transferred to another process. The syntax is as shown (lines 17 and 20)—members are added to the stream using the `<<` operator. The retrieval (using `>>`) should occur in the same order. This serialization scheme is designed for simple 'data holding' classes. It is not suited for graphs of objects that reference each other.

The data structure typedef (lines 30–32) defines an MPI data structure in which each process controls a single quadtree. In this case, the innermost `location` object is specified explicitly, using the `id_location` tag. This is necessary because the default `location_tag` compares references to determine equality. The `id_location` tag takes a comparator as a template parameter. This comparator is a class with `operator()` defined. The `dgrid::id_less` template class that is used here looks like this:

Listing 3.1: Example: Using MPI

```
#include <mpi.h>
#include <iostream>
#include "dgrid.hpp"
#include "dgrid_basic_mpi_grid.hpp"
#include "dgrid_quadtree.hpp"

class item {
private:
  int id_;
  int some_other_var_;
public:
  item(int id) : id_(id), some_other_var_(5) { }
  item() {  }
  int id() { return id_; }

  void serialize(dgrid_stream & stream) {
    stream << id_ << some_other_var_;
  }
  void deserialize(dgrid_stream & stream) {
    stream >> id_ >> some_other_var_;
  }
};

int main(int argc, char ** argv) {
  int my_rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  using namespace dgrid::tags;
  typedef dgrid::dgrid<item, mpi_grid_tag<
                             quadtree_tag<
                             id_location<dgrid::id_less<item *> > > > > grid;

  int processes[4] = { 1, 2, 3, 4};
  grid a(0, 0, 31, 31, dgrid::mpi_tiles(16, 16) . mpi_mapping(processes)
                       << dgrid::tiles(1, 1));

  if(a.is_active()) {
    a.start();
  }

  if (my_rank == 0) {
    for (int i = 4; i < 8; ++i) {
      item temp(i);
      a.insert(i, i, temp);
    }
    list<item *> results;
    a.get_range(0,0, 6, 6, results);

    for(list<item *>::iterator itr = results.begin();
                                itr != results.end(); itr++) {
      std::cout << (*itr)->id() << "\t";
      delete (*itr);
    }

    a.shutdown();
  }
  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

```
template <typename ItemType>
class id_less {
public:
  bool operator()(const ItemType & a, const ItemType & b) {
    return a->id() < b->id();
  }
};
```

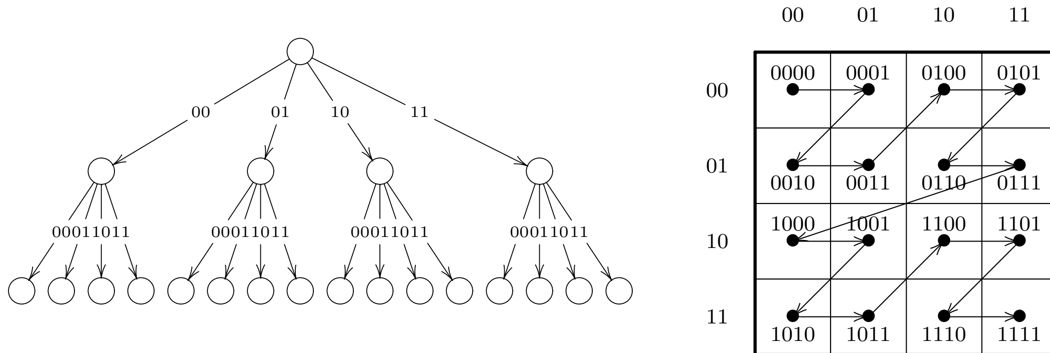
The constructor parameters (line 35) define a space of 32×32 locations, split across four processes in tiles of 16×16. The `processes` array is used to map each tile to an MPI process rank. The tiles are considered in row-major order, so that the top left quadrant $[(0,0) : (15,15)]$ is assigned to process 1, $[(16,0) : (31,15)]$ to process 2, etc. The default values (if `mpi_mapping` had been left out) would be 0–4. Note that the number of entries in `processes` must match the number of tiles.

The code discussed so far will run on all processes. This changes at line 38; `is_active()` will return `true` for processes 1 through 4, and `false` for any other processes. Processes 1–4 will then call `start()`. Those processes will handle requests in a message loop (until `shutdown()` is called from some other process).

The other processes will be free to do 'user work.' In this case, only process 0 is used (line 42). The operations and queries work exactly the same way as for the other data structures in the library. All method calls (e.g. `a.insert()`, line 45) will block until the operation is complete. This means that requests from a single process are always executed in order. However, if more than one process will be used to issue requests, then it is up to the user's code to synchronize these requests as needed.

The last section of listing 3.1 prints the search results (`6 5 4`). Line 53 shows an important detail for working with the MPI data structure: the search results must be freed using `delete`. This is the case because the search results are copies of the items that were actually 'found' on other processes. The MPI data structure only frees up those copies that it made as part of `insert` operations (this is done in the destructor).

Figure 3.1: Morton Numbering



3.1.2 Bottom-up MX Quadtree

The MX quadtree in DGrid is specialized for clustered data. Typical quadtree implementations are top-down; insertion and deletion are performed by doing a top-down search for an item's location. This means that if n is the number of nodes in a balanced quadtree, then any operation starts with a $\Theta(log_4(n))$ search (Samet [11] has a more thorough analysis of various quadtree types).

To improve performance for dynamic data, the bottom-up MX quadtree provides constant-time access to its leaf nodes. This means that the initial top-down search is no longer necessary for the insertion and deletion of items. Access to the leaf nodes is provided using a look-up table of Morton numbers. This is demonstrated in figure 3.1 for a 4×4 area. The Morton number for a particular coordinate can be obtained by *interleaving* the binary values for that coordinate. For example, the Morton number for $(10, 10)$ (row, column) is $r_1\ c_1\ r_2\ c_2 = 1100$.

This number corresponds to a traversal of the 4×4 MX quadtree shown on the left-hand side of figure 3.1. If each edge is numbered $(00 - 11)$, then the Morton number for a leaf can be obtained by concatenating the edge numbers. For this example, the leaf that corresponds to Morton number 1100 is obtained by following path 11–00 from the root node.

The Morton number for any coordinate is independent of the size of the space. This is not the case for other numbering schemes, such as row-major or

column-major order. An important consequence of this property is that only a single lookup table is needed (as long as it is large enough to accommodate the largest quadtree).

The DGrid implementation uses an STL `map` to store leaf nodes by Morton number. When an item is inserted, it is added to the map. The cost for this operation depends on the `std::map` implementation. After the item is inserted into the map, the tree may need to be modified. The internal nodes are represented by a `vector<bool>` of size $\sum_{k=0}^{n-1} 4^k$, where n is the height of the tree. The nodes are represented level by level, with `nodes[0]` as the root. The parent for a node `nodes[c]` is `nodes[p]`, where p is $c/4$ (or $(c/4) - 1$ when 4 divides c).

In the worst case, the new point was inserted in a completely empty quadrant of the space. In that case, a traversal all the way to the root is needed to 'enable' the parent nodes of the new item. If the new item already has a sibling, on the other hand, no traversal is needed. If the items are clustered, then that increases the likelihood that a new item will share a 'recent common ancestor' with an existing item.

For a range query, this quadtree implementation provides the same benefits as a top-down MX quadtree. For example, if a search range overlaps with two completely empty quadrants, then those areas can be ruled out immediately. The bottom-up implementation is more efficient for frequent insertions and deletions, however, because the height of the tree is traversed only once. Additionally, the traversal can frequently be stopped before the root is reached, especially for clustered data.

3.2 C++ Template Techniques

This section describes some of the less common template techniques that were used in DGrid to implement specific features. These techniques were used primarily to improve the usability and performance of the library.

Listing 3.2: Using tag classes to define a type

```
// (a) Definition using the implementation types
using namespace dgrid;
typedef basic_grid_impl<item,
         basic_grid_impl<item,
          quadtree_linear_map_impl<item,
           location<item> >, true>, true> nested_qt;

// (b) Equivalent definition using tag classes
using namespace dgrid::tags;
typedef dgrid::dgrid<item, full_grid_tag<
                           full_grid_tag<
                            quadtree_tag< > > > > nested_qt2;
```

3.2.1 Tag Classes using a Template Typedef

Each data structure in the DGrid library has several template parameters. These include `ItemType` and `Structure`. It is desirable to allow data structures to have additional template parameters. For example, `basic_grid_impl` takes an additional template parameter `FullGrid` that specifies whether the data structure should be fully populated or sparse. However, such additional parameters make it complicated for the user to correctly specify a data structure.

Listing 3.2 (a) shows the description of a data structure. The typedef describes two levels of fully populated arrays that contain quadtrees, which in turn contain location objects. The last two (boolean) template parameters instruct `basic_grid_impl` that it should fully populate (rather than creating data structures 'on demand'). The last two (boolean) template parameters actually belong to `basic_grid_impl`.

There are several ways to make this construction more readable:

- Eliminate the repeated `item` parameter. This is possible, since each data structure can obtain the item type from the innermost data structure. Doing this would force the user to define the entire data structure explicitly, including the `location` type.

- Move the boolean parameter (and any other 'extra' parameters) before the longer `Structure` parameter, so that it appears closer to the data structure it belongs to. Doing this would make it impossible to specify defaults

Listing 3.3: A Tag Class

```
template <class SubTags = location_tag>
class partial_grid_tag {
public:
  template <typename ItemType>
  class gridtype {
  public:
    // Omitted: compiler-specific versions of the following lines
    typedef basic_grid_impl<ItemType, typename SubTags::template
                            gridtype<ItemType>::thetype, false> thetype;

    typedef typename SubTags::template gridtype<ItemType>::thetype subtype;
  };
};
```

for these template parameters, though, forcing the user to specify every parameter explicitly.

DGrid relies on tag classes to simplify the definition of nested types. Listing 3.2 (b) shows the equivalent type definition using tags. The repeated `item` parameter is eliminated, and the boolean parameter to `basic_grid_impl` is taken care of by two separate tags (`full_grid_tag` and `partial_grid_tag`).

Listing 3.3 shows the `partial_grid_tag` template class. The tag class itself is templated only on the 'sub-tags.' The inner `gridtype` class is templated on `ItemType` so that the inner typedefs `thetype` and `subtype` can use both template parameters. The user only specifies the `SubTags` parameter, while the library code (in the `dgrid` class) sets the `ItemType` parameter. This technique is referred to as a 'template typedef.' This feature is currently under consideration for addition to the C++ standard [13].

The `dgrid` template class takes a set of tags as a template parameter. It subclasses the appropriate data structure by using the tag classes. Aside from the constructor, the class is completely empty. It looks as follows:

```
// Omitted: compiler-specific code
template <class ItemType, typename Tags>
class dgrid : public Tags::template
                gridtype<ItemType>::template thetype {
//...
};
```

The constructor delegates to the superclass. It adds an `argument_dummy` to the parameter list, so the user is not required to do this every time.

The tag classes separate the library interface from its implementation. At present, the `dgrid` class simply inherits from the appropriate data structure. This means that data structures that provide non-standard operations (such as `basic_mpi_grid_impl`) can be defined in the same way as other data structures.

3.2.2 Type-safe Constructor Parameters

The data structures in the DGrid library are designed to be nested. An important consequence is that the constructor needs to accept a variable number of parameters—the data structure's own parameters plus the parameters for any number of nested data structures. It is important that each nested data structure receives the correct parameters.

C++ does not permit variable-length parameter lists based on templates. Instead, we are forced to use a single parameter that is a composite. An intuitive solution would be to use an array or list of parameters. If the parameters are derived from a common base class, this would look as follows:

```
class DataStructure {
// Omitted: class skeleton
public:
   DataStructure(std::list<Parameter *> parameters);
};
```

This solution works, and it is type-safe in the sense that only `Parameter` instances can be added to the parameter list. However, there is no compile-time checking to see if each item is the correct subclass of `Parameter`, or even if the list size is correct.

DGrid uses recursive tuples to enable full compile-time checking of the parameters. To the user, the syntax looks as follows:

```
// Omitted: namespace details
grid a(0,0, WIDTH-1, HEIGHT-1, mpi_tiles(TILE_W, TILE_H).
                           mpi_mapping(mapping) <<
                           tiles(32,32) <<
```

```
                          tiles(1,1));
```

The last constructor parameter is a list of parameters, separated by the stream operator (<<). For data structures that take more than one parameter, the dot operator is used. In this example, the grid type is defined as follows:

```
// Omitted: namespace details
typedef dgrid::dgrid<long, mpi_grid_tag<
                          full_grid_tag<
                          full_grid_tag< > > > > grid;
```

If we were to change the tag list (e.g. by taking out the inner full_grid_tag< >), then the parameter list would need to be adjusted accordingly for the code to compile. Both the list size and the types contained in the list must match the definition of the data structure.

The parameter list structure is based on recursive pairs. Each call to operator<< appends a new item to the the list, which means that the 'first' item is always in the right innermost tuple. Listing 3.4 demonstrates the retrieval of an item from a list of four recursive pairs. Both the duo struct and the item class define operator<<(). In both cases this is a templated function, so that any class can be appended to a list.

The << operator is left-associative, which means that a << b << c << d on line 57 associates as follows: (((a << b) << c) << d). The return type for item::operator<<() is duo<OtherType, item> (line 48). The innermost (a << b) evaluates to a duo<item, item> (with a as the *second* element). The operator<<() for duo has return type duo<OtherType, duo<A, B> >, so that ((a << b) << c) evaluates to a struct of type duo<item, duo<item, item> >. Subsequent calls to operator<<() will add additional pairs.

The get_item< >::value() method returns the Nth item from the back of the list. Since N is a template parameter, its value needs to be known at compile time. The data structures in the DGrid library simply use their depth for this index. This way, the top-level data structure gets the innermost element of the list (which corresponds to the element that was inserted first).

Listing 3.4: Recursive templated tuples

```
#include <iostream>

template<typename A, typename B>
struct duo {
  typedef A FirstType;
  typedef B SecondType;

  A first;
  B second;

  duo() { }
  duo(A a, B b) : first(a), second(b) { }

  template<typename OtherType>
  duo<OtherType, duo<A, B> > operator<<(OtherType next_item) {
    return duo<OtherType, duo<A, B> >(next_item, duo<A, B>(first, second));
  }
};

template <int N, typename ItemType>
struct get_item {
    template<typename ListType>
    static ItemType & value(ListType & list) {
      return get_item<N - 1, ItemType>::value(list.second);
    }
};

template <typename ItemType>
struct get_item<1, ItemType> {
  template<typename ListType>
  static ItemType & value(ListType & list) {
    return list.first;
  }
  static ItemType & value(ItemType & list) {
    return list;
  }
};

class item {
private:
  static int id;
public:
  int the_id;
  item() {
    the_id = item::id++;
  }
  template<typename OtherType>
  duo<OtherType, item> operator<<(OtherType next_item) {
    return duo<OtherType, item> (next_item, *this);
  }
};

int item::id = 0;

int main(int argc, int argv) {
  item a;  item b;  item c;  item d;
  duo<item, duo<item, duo<item, item> > > myList = a << b << c << d;

  std::cout << get_item<4, item>::value(myList).the_id << std::endl;
  return EXIT_SUCCESS;
}
```

The method `get_item::value()` is evaluated recursively. The base case `get_item<1, ItemType>::value()` returns `duo::first` of the current tuple. The innermost element is treated as a special case (line 34). That special case is actually used in this particular example, since `get_item<4, item>::value(myList)` (line 59) retrieves the innermost element of the list. The output of the code in listing 3.4 is '0.'

The mechanism described here requires that each data structure have a way to find the type of the parameters taken by its inner data structures. This is done through recursion. Together with the resulting constructor definition it looks as follows:

```
// Omitted: class skeleton

typedef Structure subtype;

typedef typename subtype::template
        TheArgument<tiles>::Type Argument;

template<typename SuperType>
struct TheArgument {
  typedef typename subtype::template
           TheArgument<Duo<tiles,SuperType> >::Type Type;
};

basic_grid_impl(int x0, int y0,
                int x1, int y1, Argument args);
```

In this sample, `Structure` is the template parameter that defines the data structure that is contained within the current data structure. The typedef to `subtype` is there to make this type information publicly available through `typename DataStructure:: subtype` (this is not possible using template names).

The second typedef defines `Argument`, the type that is actually used by the constructor. `Argument` is a shorthand for `subtype`'s `TheArgument< >::Type`. The `Type` member is evaluated recursively from the top-level data structure down. Each data structure uses the `SuperType` template parameter to denote the parameter list 'thus far,' so that the inner data structures can add their parameter type

to the list. This recursion ends with the `dgrid::location` class, which appends `argument_dummy` (an empty class) to the list of requested argument types.

When the constructor is called for any data structure, it removes the innermost argument from a copy of the `args` list. The 'tail' of the list then serves as the argument for any new `substructure`s. The data structure has to take its own argument out of the list, since the `substructure` will not accept it otherwise (i.e. the code would not compile).

The only issue that remains is how to pass more than one argument to a single nested data structure. At present, the only data structure that takes more than one parameter is `dgrid::basic_mpi_impl`, which takes `mpi_tiles(int, int)` and `mpi_mapping(int * mapping)` (see section 3.1.1 for details). Since `mpi_tiles` is a mandatory parameter, `mpi_mapping` is simply defined as a method of the `mpi_tiles` class:

```
mpi_tiles mpi_mapping(int * processes) {
  if (!use_processes_) {
    processes_ = processes;
    use_processes_ = true;
  }

  return *this;
}
```

This way of handling multiple arguments does not scale to large numbers of arguments, where some arguments might be optional. The Boost Graph Library solves this problem in the implementation of its 'named parameters' feature, using inheritance [3]. The solution shown here is simpler to implement for a small number of arguments, however.

3.2.3 Efficient Callbacks through MPI

The DGrid library allows the use of predicate functions to limit search results. The `get_range` query operation can take a function as a template parameter. This function should return `true` for those elements that should be added to the search results. Because the function is a template parameter, it can

be inlined. This eliminates the overhead of repeatedly calling the function.

The `basic_mpi_grid_impl` data structure allows the library to be used in MPI applications. This complicates the use of predicates, since it is not possible to 'send' a template parameter from one process to another. We solve this problem by requiring that users 'register' their predicate functions. The predicates are registered in the same order in each process, and each receives a unique identifier which can be sent from one process to another.

The registration is handled by several methods and classes in the MPI implementation class. The relevant code from `basic_mpi_grid_impl` is shown in listing 3.5. The client code might look as follows:

```
bool even_x_pred(int x, int  y) {
  return x % 2 == 0;
}
// Omitted: instantiate a data structure myGrid
myGrid.register_predicate<even_x_pred>();
```

The `register_predicate` method (line 49) instantiates an object of type `coord-_predicate< >`. The constructor for that class (line 18) simply sets an identifier, using the counter in its base class (`predicate_base`, line 8). Template function parameters are matched based on the function name, so that a separate class (with its own static variables) is generated for each unique call to `register_predicate`. The base class allows these template classes to share a single counter (`ID`, line 10).

The preceeding steps generate a unique identifier for each registered predicate. The next step adds a pointer to `get_range` (line 30) to the `predicates_` map. The static `get_range` method is a placeholder for the `Structure`'s own `do_get_range` method. This way, the pointer is a regular function pointer, which works more reliably than a pointer to member. The pointer to function is added to `predicates_` in line 52.

At this point, the MPI processes share the same unique identifier for that particular function. This identifier can be sent between processes as part of a search request. The recipient would call the appropriate function like this:

Listing 3.5: Predicate inlining across processes

```
// Omitted:
// - basic_mpi_grid_impl class body
// - subclasses of predicate_base (only coord_predicate is shown)
// - various sanity checks
// - initialize static members (e.g. predicate_base::ID)

private:
  class predicate_base {
  protected:
    static int ID;                                                    10
  };

  template<bool T_function(int, int)>
  class coord_predicate : public predicate_base {
  private:
    static int the_id_;
  public:
    coord_predicate() {
      coord_predicate::the_id_ = ++ predicate_base::ID;
    }                                                                 20
    static int the_id() {
      return the_id_;
    }
  };

  template<bool T_function(int, int), bool T_function2(ItemType & item),
           bool USE_COORD_PRED, bool USE_ITEM_PRED, typename Structure_Type>
  class get_range_holder {
  public:
    static void get_range(const int x0, const int y0,                 30
                          const int x1, const int y1,
                          list<ItemType *> & items, Structure_Type * structure) {

      structure->do_get_range<T_function, T_function2,
        USE_COORD_PRED, USE_ITEM_PRED, list<ItemType *> >
        (x0, y0, x1, y1, items);
    }
  };

  // map predicate IDs to function pointers                          40
  typedef void (*get_range_ptr)(const int, const int,
                                const int, const int,
                                list<ItemType *> &, Structure *);

  map<int, get_range_ptr> predicates_;

public:
  template<bool T_function(int, int)>
  void register_predicate() {
    coord_predicate<T_function> cp;                                   50

    predicates_[coord_predicate<T_function>::the_id()] =
     &get_range_holder<T_function, item_pred, true, false, Structure>::get_range;
  }
```

```
(*(predicates_[item_pred_id]))
    (x0, y0, x1, y1, search_result, theitems_);
```

The items stored in `search_result` would then be sent back to the requesting process.

CHAPTER 4
Results and Future Work

The current implementation of the DGrid library is successful in several areas. Informal testing has shown the serial data structures in the library to use significantly less memory than a naïve array-based implementation, while offering better performance. The library also provides a good proof-of-concept for the use of templates as an alternative for the Composite design pattern. The other uses of templates, e.g. for parameter list passing, significantly improve the library's ease of use. Appendix A shows a full-length example that illustrates basic insertion, deletion, and querying operations; section 3.1.1 has a similar example that uses the MPI data structure.

Future work on the library could focus on the following features:

- additional query types – The library offers only orthogonal (i.e. rectangular) range queries. Other query types, such as nearest neighbor search, also have many applications. The design of the library does not preclude it from implementing other types of queries as well.

- more generic design – At present, search operations take a data structure that is used to 'deliver' the results to the client code. A more flexible approach would use input iterators instead, so that the library is minimally dependent on external classes. Stroustrup [12] describes iterator types.

  Another potential improvement would be to further separate the interface from the implementation. Currently the main `dgrid` class simply inherits from the appropriate data structure. Instead, each implementation class could provide separate 'public' interface to the user.

- load balancing – The MPI data structure currently divides the space evenly according to a fixed tile size. For clustered data, it is probable that one

tile is completely empty, while another has a large cluster of items. If each process handles a single tile, this results in an uneven distribution of the workload. Consequently, it would be beneficial to adjust the distribution based on the 'density' of the data. Additionally, many distributed systems are heterogeneous; the processors have different characteristics, and the bandwidth that is available between processors may not be uniform. This would have to be considered as well.

- additional data structures – The only non-standard data structure that is currently supported by the library is the MX quadtree, which is optimized for highly dynamic, clustered data. Adding additional data structures could provide more diverse trade-offs between memory usage, query time, and construction time. The library could be made to support data in higher dimensions as well.

LITERATURE CITED

[1] Karel Driesen and Urz Hölzle. The direct cost of virtual function calls in C++. In *OOPSLA Conference Proceedings, San Jose*, 1996.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of reusable object-oriented software*. Addison Wesley, 1995.

[3] Lie-Quan Lee, Andrew Lumsdaine, and Jeremy G. Siek. *The Boost Graph Library*. Addison - Wesley Publishing Company, Inc., 2001.

[4] David M. Mount and Sunil Arya. ANN: A library for approximate nearest neighbor searching. http://www.cs.umd.edu/˜mount/ANN/.

[5] David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide, Second Edition*. Addison Wesley, 2001.

[6] David R. Musser and Alexander A. Stepanov. Generic programming. In *ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation*, 1989.

[7] David R. Musser and Alexander A. Stepanov. Algorithm-oriented generic libraries. *Software - Practice and Experience*, 24(7):623–642, 1994.

[8] Toshiro K. Ohsumi. Personal communication, March 2005.

[9] Peter S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers, Inc., 1997.

[10] Hanan Samet. *Applications of Spatial Data Structures*. Addison - Wesley Publishing Company, Inc., 1990.

[11] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison - Wesley Publishing Company, Inc., 1990.
Context. Presents many different uses of various spatial data structures, including k-d-trees.

[12] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.

[13] Herb Sutter. Proposed addition to C++: Typedef templates. www.gotw.ca/publications/N1406.pdf.

[14] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates : The Complete Guide*. Addison Wesley, 2002.

[15] Todd L. Veldhuizen. C++ templates are turing complete. http://osl.iu.edu/~tveldhui/papers/2003/turing.pdf.

[16] Todd L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

APPENDIX A
An example program

The following is a full-length example of a program that uses the DGrid library to perform some basic operations. The preceeding chapters contain extensive examples that explain various features. Listing 3.1 is a full-length example of how to use the MPI data structure; the following example uses other (non-parallel) data structures.

Listing A.1: Example: Using DGrid

```
#include <iostream>
#include <list>
#include "dgrid.hpp"
#include "dgrid_basic_grid.hpp"
#include "dgrid_quadtree.hpp"

template <typename SequenceType>
void display_sequence(SequenceType & seq) {
    typedef typename SequenceType::iterator iterator;
                                                                              10
    std::cout << "[" ;
    for (iterator i = seq.begin(); i != seq.end(); ++i) {
        std::cout << **i << "␣";
    }
    std::cout << "]" << std::endl;
}

int main (int argc, char ** args) {
  using namespace dgrid::tags;
  typedef dgrid::dgrid<int, full_grid_tag<                                    20
                           quadtree_tag< > > > grid;

  // 3x3 quadtrees of size 1024x1024 each
  grid myGrid(0,0, 3071, 3071, dgrid::tiles(1024, 1024) << dgrid::tiles(1,1));

  int b = 5;
  int c = 7;

  myGrid.insert(10, 10, b);
  myGrid.insert(11, 10, c);                                                   30

  std::list<int*> search_results;

  myGrid.get_range(0,0, 100,100, search_results);
  display_sequence(search_results); // expect {b,c}

  search_results.clear();
  myGrid.get_range(0,0, 10, 10, search_results);
  display_sequence(search_results); // expect {b}
                                                                              40
  myGrid.remove(10, 10, b);
  myGrid.remove(11, 10, c);
```

```
  search_results.clear();
  myGrid.get_range(0,0, 100, 100, search_results);
  display_sequence(search_results); // expect {}

  return EXIT_SUCCESS;
}
```