ALTERNATIVE APPROACHES TO PARALLEL GIS PROCESSING

by

Nathan Thomas Kerr

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

August 2009

ALTERNATIVE APPROACHES TO PARALLEL GIS PROCESSING

by

Nathan Thomas Kerr

has been approved

August 2009

Graduate Supervisory Committee:

Dr. Daniel Stanzione, Chair
Dr. Robert Pahle
Dr. Yi Chen

ACCEPTED BY THE GRADUATE COLLEGE

ABSTRACT

Geospatial Information Systems were designed to model the world around us. As our abilities to gather data and analyze it in more sophisticated ways have improved we have had larger datasets with more computationally intensive methods for analysis. Even with the increased power of desktop computers, desktop GIS processing applications are unable to perform the desired analysis.

This thesis applies the parallel processing paradigms of map reduce and message passing to GIS processing, creating two implementations of parallel GIS processing environments, one using the open source Hadoop map-reduce framework and the other using MPI. Both implementations are evaluated for scalability and usability.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

CHAPTER 1 INTRODUCTION

Geographic Information Systems (GIS) were designed to model aspects of the world around us. From roads to temperature, GIS data can be used to represent a large range of real-world objects, allowing for sophisticated analysis and processing.

GIS analysis and simulation are used to understand and develop the environment around us. A common use of GIS data is found in the GPS based car navigation systems in common use today. City planners will use GIS data in applications such as population growth models, land use planning, and traffic management.

The amount of data to be processed increases as populations grow, communities become more complex, or the size of the processing area grows. In the year 2000, 1.2 million parcels of land were listed in the 9,224 square miles of Maricopa County, Arizona. Each parcel of land is represented as a georeferenced polygon along with various attributes such as a parcel id, ownership information, and zoning codes. Parcels of land is just one set of data that is kept for Maricopa County. Other datasets include roadways, railways, rivers, schools, voting districts, etc.

When the data required to complete GIS processing grows beyond the memory available to a single processor, the processing method must be adapted to fit within that limitation. A common method is to not load the entire dataset into memory, but to read it off disk as needed. This method works well if the data is only needed once, as reading from disk is slow.

One application, which is looked at in more detail later, associates businesses with parcels of land (the data was geocoded differently). The algorithm is quite simple: for each business, find the nearest parcel of land with a compatible zoning code. In Maricopa County in the year 2000, there were 34,302 businesses and 1,218,130 parcels of land. While these particular datasets fit into memory today, they did not just a few years ago.

Given increased parcel density, or a larger area of land, the memory capabilities of a standard computer are quickly reached. While there are specially made computers with extraordinarily large amounts of memory, they are also extraordinarily expensive.

In addition to memory limitations, the processor also limits the processing that can be done. In this case 41,784,295,260 comparisons between businesses and parcels is made. If a computer is able to make one million comparisons per second, the processing will take approximately 11 hours, 37 minutes. A factor increase in speed to ten million comparisons per second reduces the time to 1 hour 10 minutes. If only there was a single computer that would perform 100 billion of these comparisons per second, then the processing would be done in less than half a second!

As an infinitely fast computer with infinite memory does not exist, more effort is required to perform this processing in a reasonable time.

1.1 Parallel GIS Processing

Using multiple interconnected computers to complete the required GIS processing allows for increased memory capabilities along with increased computation power. Processing algorithms must be reworked to allow for this collaboration between computers.

Programming for multiple machines is not the easiest of tasks. A common paradigm for parallel programming is Single Program, Multiple Data[1] (SPMD), which uses a single program that is run on all the computers included in the parallel computation where each instance of the program operates on different data, for example a different set of businesses. This paradigm simplifies parallel algorithm complexity while providing enough power to increase memory capability and computation power.

The first step in working with an SPMD program is splitting the data. GIS data can be split in several different ways. The specific data splitting method used is dependent on what data is needed on what machine. The data used in this thesis is a set of records; each record contains a geometry with its associated attributes. One method that can be used is to evenly distribute the records among the participating machines. Another method takes into account the varying size of geometries in memory by splitting the data up by size while taking into account record boundaries. Another class of data distribution is to geographically split up the dataset, for instance into quadrants with each machine having responsibility for one or more quadrants. This method is more complicated in the setup required and the exceptions that need to be handled such as how to handle geometries that span more than one quadrant.

After the data is distributed, the processing methodology often needs to be adjusted because access to the full dataset at once is no longer possible. Combined with the data distribution, these two additions are needed for a parallel implementation, but not the serial case. This is the overhead required to take advantage of more than one computer to complete the processing.

Parallel performance is measured through speedup:

$$\text{speedup}(n) = \text{time}(n)/\text{time}(1) \tag{1.1}$$

which compares the execution time of the process on n processors to the execution time on 1 processor. Ideal speedup is n.

The scalability of a parallel solution can be seen by graphing speedup by number of processors used. The best realistic case is to have linear scalability, meaning that speedup is directly proportional to the number of processing elements used.

1.2 Alternative Approaches to Parallel GIS Processing

This thesis implements and evaluates two parallel, dataset centric approaches to processing large geospatial datasets on clusters with the intent of gaining insight into which approach is more suitable for GIS processing, and why. The first approach uses the map-reduce paradigm. The second uses message passing, which is currently the standard approach for programming clusters. This thesis uses existing implementations of both these paradigms, focusing instead on their applicability for GIS processing. Furthermore, the geospatial operations required to handle GIS processing have been distilled into libraries that conform to the Open Geospatial Consortium's Simple Features[2] standard. By applying data parallel programming methods to GIS processing, both these approaches create a fairly easy environment to program in while providing significant speedup and scaleup.

The first approach, called hadoopGIS, uses the open source Hadoop map/reduce framework. Hadoop is based on Google's MapReduce[3]. Map reduce defines a two-phase method to working with data. The first phase, map, applies a function to every record in a data set. The map function can output one or more key-value pairs. Map is an inherently parallel process, as no record is need to process any other record.

After the map phase, the generated key-value pairs are aggregated by key and passed to reduce processes, one reduce process per key. There are generally many fewer reduce operations as compared to map operations. Reduce operations are inherently serial, as the operation must have access to all the key-value pairs associated with the key being processed.

HadoopGIS simply adds a GIS data type to Hadoop, thus enabling for GIS processing. Geospatial processing is provided through the Java Topology Suite[4] (JTS).

The second approach, called clusterGIS, uses the MPI[5] message passing library.

Message passing works by creating a process, called a task, on each participating processor core. Each task operates independently and is assigned a unique address. Tasks can collaborate with each other by passing messages.

ClusterGIS is simply library of functions that employ MPI and the GEOS[6] library to handle common situations such as loading and saving data.

The rest of this thesis is organized as follows: Chapter 2 explores related efforts in GIS processing. Chapter 3 then defines the specific requirements needed for a good parallel GIS processing engine. Chapter 4 details the design choices for HadoopGIS and ClusterGIS. Chapter 5 defines a set of tests and list individual results. Chapter 6 evaluates the two implementations by comparing performance results against the requirements defined in chapter 3. Chapter 8 summarizes conclusions drawn from the evaluation.

CHAPTER 2 RELATED WORK

This thesis applies parallel processing techniques to the field of GIS processing. I will first look at the state of GIS processing, then Parallel Processing in general, then parallel GIS processing.

2.1 GIS Processing

Geographic Information Systems (GIS)[7] have been in use since the 1960's with the Canada Geographic Information System (CGIS)[8] and then moving from the mainframes to current desktop applications like ESRI's ArcGIS[9] product, which began development in the 1980's.

In general, there are two types of GIS data: raster and vector. Raster data is a set of cells, such as pixels in a picture, that have one or more attributes (e.g., temperature, humidity, elevation). Each attribute covers the entire area of the pixel. The entire raster dataset is spatially located and states how much area is covered by each cell.

Vector data is comprised of spatially referenced geometric objects such as points, lines, and polygons. Each object represents something in the real world, and has associated attributes.

Most GIS processing systems are able to handle both raster and vector data sources.

2.1.1 Desktop GIS

Desktop GIS packages such as ArcGIS[9], QuantumGIS[10], and GRASS GIS[11] are commonly used for GIS processing and analysis. While these programs provide graphical interfaces to their GIS capabilities, their capabilities are limited by the computers they run on. Datasets can be too large for their memories and computations can take too long to be practical.

Desktop packages are often supplemented with a database component such as ArcSDE[12] or PostGIS[13] which acts as a centralized repository for GIS data which can be shared between a workgroup. It is important to note that the database is generally used for storage and sharing, not for computation. Computation is performed by the desktop package.

2.1.2 Database GIS

An alternative to performing GIS processing in a desktop program like ArcGIS, is to employ a geospatial database like PostGIS[13], ArcSDE[12], or Oracle Spatial[14]. Geospatial databases allow centralized access to, and processing of, geospatial data through query languages such as SQL. As data is stored and managed by the database software, advanced database features such as indexes can be utilized to speedup data access and processing.

PostGIS is utilized as the core component of the Urban Systems Frame-work[15] (USF) designed by the Digital Phoenix[16] project group at Arizona State University. Digital Phoenix tries to integrate 3D visualization technology with simulated and gathered GIS data to better understand the impacts of urban planning decisions.

2.1.3 GIS Simulation and Analysis

GIS simulation and analysis can also be done using more specialized environments. UrbanSim[17] is a popular simulation tool for growth models.

GeoDa[18] is a spatial analysis tool that includes spatial regressions. PySal[19] is a python library that builds on the work done with GeoDa.

These tools provide additional capabilities for GIS processing, but are limited to working on a single computer. As the data becomes larger and the analysis more complex, they become unable to perform the required processing in a reasonable time, if at all.

2.1.4 GIS Libraries

The Open Geospatial Consortium (OGC) defined a core set of geospatial processing operations in their Simple Features[2] standard. These core operations allow for most geospatial processing needs. One of the main libraries that provides these operations and related data types is the Java Topology Suite[4] (JTS). Though written in Java, the JTS has been used as the basis for ports into other languages. The Geometry Engine, Open Source (GEOS) library is a C++ port of the JTS that also provides a C interface. PostGIS is implemented using the GEOS library. The NetTopologySuite[20] (NTS) is a port of the JTS into .NET.

The Simple Features standard and the libraries that support it have distilled the most important functions for GIS processing. The standard through the experience and discussion employed in the creation process, and the libraries by practical use in other projects and environments.

As quality libraries are available for a variety of languages, there is no need to re-implement the functionality they provide. This thesis makes direct use of the JTS and GEOS libraries.

2.2 Parallel Processing

2.2.1 Serial and Parallel Shared-Memory Applications

Computer programs are executed by processors. The simplest of programs is made up of a series of operations that are executed in order by the processor. As this type of application is comprised by a series of operations it is known as a serial program. Serial programs are not able to take advantage of more than one processor. To utilize more than one processor at a time, a set of serial programs that can communicate with each other is required. Each serial program in this set is referred to as a thread. Thus multi-threaded programs can utilize more than one processing core. The simplest method of communication between threads is to share a common section of memory. Therefore a parallel shared-memory application could utilize at most the number of processors able to be connected to a single section of memory.

Both serial and parallel shared-memory applications are limited to a single computer, where computer means a group of processors that are connected to the same memory. Most modern computers provide this capability.

2.2.2 Parallel Distributed-Memory Applications

One method to overcome the limitations of a single computer is to use multiple computers. By providing a way for the computers to communicate, and therefore collaborate, with each other. By means of collaboration, the individual computers are able to act together to solve a single problem. While the participating computers can communicate, they do not have direct access to each other's memory. Thus the total memory available to the collection of processors is distributed between them, not shared.

Because each computer only has direct access to its own memory, meaning that finding out what is in the memory of another computer is harder, the main task in designing a parallel program is figuring out how to split up the work between computers. One method is to split up the processing between computers. Each task would generally have the same data and perform variations of an operation on the data; for instance, running multiple scenarios. This methodology is called Task Parallel. An example of task parallelism is simulating the effects of various weather patterns on an urban environment. Each task would have a copy of the environment and run its variety of weather on it. The capability of executing each scenario is limited to the capabilities of a single processing element, but many scenarios can be executed at the same time.

Data Parallel methods split the data up between computers and perform the same, or similar, operations on each piece of data. To calculate the effects of a weather pattern on an urban environment, the environment would first be divided between the tasks with each task responsible for one part of the environment. Each task would then calculate the effects of the weather on its section of the environment, communicating with the other computers as needed to share information related to edge conditions, etc.

2.2.3 Message Passing Interface

The Message Passing Interface[5] (MPI) standard has become the normal way of programming parallel distributed-memory applications. MPI works by starting processing tasks on each of the computers alloted to the MPI process. These tasks are then able to send and receive messages between themselves allowing for inter-task communication.

As MPI defines a simple paradigm for inter-task communication, any parallelism in the application must be explicitly specified and programmed. Thus MPI programming is not simple or easy, though it is not without benefits.

MPI libraries are able to utilize advanced network layers such as InfiniBand without changes to the application, except for recompilation against the library. This design allows for advances in technology to be passed onto parallel application with little effort.

MPI is also able to take advantage of parallel filesystems such as Lustre[21]. Parallel filesystems spread file data between several file servers. Client programs are then able to access each part of the file in parallel, speeding file reading and writing while enabling each task to read or write a portion of data while not conflicting with the other tasks in the same MPI process.

MPI is generally deployed on clusters, making programs that are based on MPI able to run on a variety of machines.

2.2.4 Map Reduce

Map reduce is a functional programming idiom that has recently become popular due to Google's extensive use first described in a 2004 paper[3]. While Google's MapReduce environment is proprietary, the concepts and idea have passed into the open source Hadoop[22] framework developed by the Apache Software Foundatation with major support by Yahoo! and Cloudera.

Map reduce is relatively simple to program for. The programmer only needs to supply two functions: map and reduce. The map function takes as its input a record of the input data, record splitting is handled by the framework, and outputs zero or more key-value pairs. The reduce function takes a key and a set of values associated with that key and outputs zero or more key-value pairs.

Unlike MPI, parallelism is handled by the map-reduce framework which loads a portion of the entire input dataset on each of the machines participating processing elements, splits the data into records, executes the map function on each record, aggregates all the key-value pairs produced by the map functions and runs the reduce function on each unique key, passing the associated values along. At last the output from the mappers is collected into the final output of the program. Parallel operations are done implicitly, and therefore do not need to be created by the programmer.

Map reduce was designed on the assumptions that disk is cheap, networking is expensive, and that large systems can be built with inexpensive hardware, as long as no piece of hardware in indispensable. With these restrictions map reduce makes use of a distributed file system that replicates data between several of these inexpensive computers. Map reduce tasks are then ran on one of the computers that has the block of data it needs. The blocking used to spread files on the distributed file system are also used as the basis for parallelizing the map reduce process.

Map reduce is basically a two phase solution for parallel computing. The first phase applies a function, map, to each record in the input dataset. This phase is inherently parallel as each call to map is completely independent, requiring no more data than just the record that only it is responsible for. The second phase, reduce, is inherently not parallel. Some parallelization is given by Hadoop in this phase by limiting a reduce function from seeing all the data produced by the map function to only the data produced with the same key (each output being in the form of a key-value pair). Because of this limitation, reduce has some parallelism, but much much less than in map.

Hadoop itself is comprised of two major parts: a distributed filesystem, and a parallel execution engine. The filesystem, Hadoop Distributed Filesystem (HDFS), splits files into blocks which are then spread among the participating computers. Blocks can be automatically replicated, such that the loss of any participating computer will not cause data loss. The execution engine is friendly with HDFS, in that it knows where file blocks, and there replicas, are located and can place computation on the same computer that contains the data. The idea here is that data is larger than the program and that therefore moving the program to the data will improve efficiency. Hadoop may also start additional computation tasks that replication the computation being done elsewhere on one of the replicated blocks.

Basic Hadoop programs require three parts. The first part is some startup code that tells Hadoop which input files to use, and which map and reduce functions to execute. The Hadoop environment will look at the inputs and decide on the number of mappers to create. Each mapper is responsible for channeling the data from a file block to the map function, and then taking that output and passing it onto the reducers. The number of mappers used is based on two criteria. Each file will get a mapper. If a file consists of more than one block of data, a mapper will be created for each additional block. This process is meant to be done automatically, but has real-world effects on program performance.

2.3 Parallel GIS Processing

Attempts have been made to overcome the limitations of single machine implementations. Of primary interest are those extending current methods to use multiple computers.

2.3.1 Problems with Desktop Programs on Clusters

Current desktop approaches to GIS processing such as ArcGIS are unable to make use of the multi-machine processing environment that compute clusters provide. While reworking these programs to utilize these extended resources is possible, it is non-trivial.

GRASS GIS was reworked[23] to use its collaboration features to distribute subqueries among computers. The method described in this paper utilizes multiple instances of GRASS in a master-slave configuration where all participants access a shared data repository or filesystem. The geometries are portioned between the various nodes. Operations are done on the subsets, and the results are merged to produce the final result.

While the method used to extend GRASS GIS will in fact speed up GIS processing, it has two flaws. First, GRASS is designed to be used in an interactive mode rather than a batch or script driven approach. Second, the entire set of data used must still fit on a single computer to move it in or out of the environment.

2.3.2 Parallel Databases

Parallel databases such as TeraData[24] and Oracle[14] use data parallel methods. Paradise[25, 26] spreads data between computers using round-robin, hash, or spatial partitioning. Because the data is able to be distributed between multiple computers, the processing is able to scale to larger datasets.

When a query is processed across the database, a task is created for each fragment of the data. Thus as the data grows larger, the processing capabilities of the system also increase. If a particular processing operation requires relatively less computation for each data record this is fine. However, after the amount of computation per data record increases beyond a certain point, which is dependent on the speed of the machine used, the processing operation can be sped up by utilizing more processors. Major factors in determining how much data should be processed on each machine, and therefore how the data should be spread between machines, are memory, computation, and communication overhead to move the data and computation to another machine. The ratio of computation to memory and commutation requirements is often referred to as grain size. Coarser grained processes have more computation per data record, while finer grained processes have little computation for each data record.

Databases excel at working with indexed data while allowing multiple users to interact with the data in a concurrently safe manner through the use of atomic transactions. The requirements placed upon database systems to handle these situations slow down computations that don't utilize indexes or work on an entire dataset at once. The processing operations this research examines do not require these restrictions, and as such a more efficient system can be created.

Many universities and research institutions already have significant investment in compute clusters. These clusters are groups of computer linked together with high speed network interconnects and high performance parallel filesystems such as Lustre[21]. By separating compute and storage resources at the cost of a high speed network, compute clusters are able to separate computation from data storage.

Parallel filesystems allow the data to be separated from the computer where the processing will be executed by spreading files across multiple network connected fileservers allowing access that can be faster than utilizing a computer’s local disk for storage while also enabling processing to spread across the available compute resources based entirely on the process’ grain size.

2.3.3 Problems with Current Parallel Approaches

While desktop GIS programs are relatively easy to use, their current implementations are fundamentally unable to make use of parallel computation resources. The attempt to use GRASS was not entirely successful due to its lack of a non-interactive mode and inability to work on datasets larger than could be handled by a single computer.

Parallel databases require dedicated resources and expertise to be useful. Even so, database are limited in their ability to work with a large range of process granularity as data redistribution is expensive in terms of rebuilding indexes and configuration changes. Furthermore optimizing SQL often requires knowledge of the data distribution and configuration of the database. Most companies that use databases as a core technology have dedicated staff to manage these systems.

Chapter 3 details the requirements of a good parallel approach to GIS processing.

CHAPTER 3 REQUIREMENTS

Chapter 2 related the current state of parallel GIS processing, from which the limitations of current approaches can be seen. From the capabilities and limitations of current approaches, the requirements of a good parallel GIS processing engine can be defined. Some of these requirements are derived from what makes good cluster based software, such as batch mode processing and scalability.

The main requirements of a parallel GIS processing engine are that is supports standard geospatial operations, can be executed in a batch environment, makes effective use of the additional resources provided by the cluster environment, and is not too hard to use.

3.1 Standard Geospatial Operations

Any GIS processing application must have at least a core set of GIS processing operations. Without the capability to perform the required processing, such an engine would be useless. The Open Geospatial Consortium (OGC) defines a set of such operations in their Simple Features[2] standard.

The Open Geospatial Consortium, Inc. (OGC) is a non-profit, international, voluntary consensus standards organization that is leading the development of standards for geospatial and location based service. The OGC maintains a variety of standards that support GIS processing such as cataloging, KML, WMS, and others.

The most important standard for this thesis is the Simple Features Standard[2] (SFS). The SFS defines ways of storing and processing data including defining the Well Known Text (WKT) and Well Known Binary (KWB) representations of geospatial data. Along with data formats, a set of operations for working with geospatial data are specified.

The OGC SFS is just a standard, and so it needs to be implemented. The main open-source library implementing the standard is the Java Topology Suite[4] (JTS). While the

JTS is implemented in Java, it has been ported to other languages. For example PostGIS uses the C/C++ GEOS library[6].

Compliance to this requirement can be tested by checking an independent implementation against the standard, or assumed thought the use of a compliant implimentation such as the JTS or GEOS libraries, assuming full access to the library functionality is allowed.

3.2 Batch Mode Processing

As many clusters only allow non-interactive, or batch, processing modes, the parallel GIS processing engine must fit this criteria.

To be compliant with this requirement, an implementation must support at least the standard geospatial operations while in batch mode.

3.3 Scalable

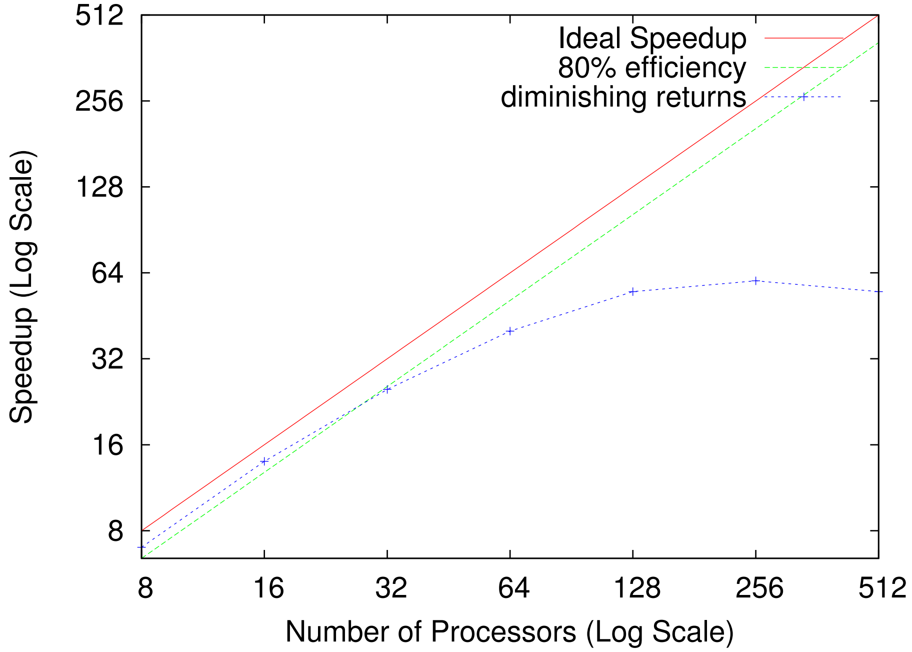
The goal of a scalable application is utilize the resources given it effectively. There are two aspects to scalability: capacity and speedup. Capacity scalability means that an implementation is able to solve a bigger problem than would otherwise be possible using a single machine. Speedup measures the impact of using more resources as compared to not using them.

Speedup is measured by comparing the processing time on a specific number of processors (n) to the processing time on a single processor:

$$\text{speedup}(n) = t(n)/t(1)$$

To see how scalable an application is, a speedup plot is generated which shows ideal

Figure 3.1: Example Speedup Graph



speedup for the application based on a serial run. Figure 3.3 shows a sample speedup graph with three speedup lines plotted. The Ideal Speedup line shows what the ideal speedup would look like on the graph enabling easy comparison of the other line(s). The 80% efficiency line shows an application that has achieved linear speedup. This application scales very well. The diminishing returns line shows an application that increases speedup to a point around 256 processors, at which time adding more resources actually increases the program runtime. This limitation could occur because of limited data size or some other concern.

Graphs like this will be used throughout this thesis to evaluate scalability.

3.4 Ease of Use

The last requirement is the most ambiguous: ease of use. The main thing to think of here is that most people doing GIS processing are interested in the results of the processing and would like the method to obtain those results to be as easy as possible. In addition, most GIS processors are not computer scientists, and so if a programmable interface is the point of interaction, it must be as easy to use as possible so that the learning curve is not excessive.

Ease of use is a subjective measure. No user studies will be performed here, but it is a point of discussion that is important in evaluating a GIS processing engine implementation.

CHAPTER 4 DESIGN

To discover the differences between using message passing and map reduce as approaches for parallel GIS processing, this thesis implements two environments for manipulating GIS data. The first approach, hadoopGIS, is based the Hadoop map-reduce framework. The second, clusterGIS, uses MPI. Both implementations build on these existing technologies and utilize standards compliant geospatial libraries. The main work is combining the underlying parallel technology with geospatial processing capabilities in a way that works for that paradigm.

One point of simplification made for both implementations is supporting only one file format. Both hadoopGIS and clusterGIS use CSV formatted data with geospatial data stored in the Well Known Text (WKT)[2] format. HadoopGIS adds an additional file which stores the column names, one on each line. ClusterGIS just uses column numbers.

4.1 HadoopGIS

HadoopGIS provides a Hadoop native GIS datatype. This allows for GIS data to be used in Hadoop just like any other supported format. Geospatial processing support is provided through the Java Topology Suite[4] (JTS). Adding a new datatype to the Hadoop environment required overcoming three data transfer boundaries: file to map, map to reduce, and reduce to file.

The first gap to overcome is getting data into the mappers. Hadoop uses two classes in addition to the core GIS class to load data from HDFS blocks. The process starts with GISInputFormat which uses GISRecordReader. GISRecordReader is responsible for extracting the records from a file block. In this case, GISRecordReader wraps the Hadoop supplied LineRecordReader, which manages to solve the problem of connecting lines in a file split between blocks. In this phase the geospatial data is converted from

WKT to a Geometry object provided by the Java Topology Suite.

To bridge the gap between mappers and reducers, Hadoop created an interface called Writable. The GIS class implements writable by providing functions that serialize the object state into a byte stream, and reconstitute an object from that byte stream.

The last gap to cross is taking data from the reducers and putting it back on HDFS. The process here is nearly a mirror image of loading data, except that the data is being written. The process starts with GISOutputFormat which utilizes GISRecordWriter. GISRecordWrite converts the GIS object back into CSV format and sends the resulting byte stream through the supplied DataOutputStream.

This design builds upon the basic architecture of Hadoop and adds as little as possible to get a GIS datatype to work in the environment. The user of this new functionality has access to the full power of both Hadoop's task and data management which allows for parallel processing, and the capabilities of a JTS geometry object which implements all the standard geospatial operations. All of this is done for the user by them stating that the input data should use the GISInputFormat and outputted data uses the GISOutputFormat.

This enhancement does not address Hadoop's architecture. This means the use of a secondary dataset is still problematic. The map function only is provided with one record. The framework expects most computation that requires access to more than just one record to be accomplished in the reduce phase. There is provision, however, to load extra data when a mapper is created. These limitation will be discussed further in the experimental setup and results section of this thesis.

4.2 ClusterGIS

ClusterGIS makes use of MPI to manage communication between multiple tasks and the GEOS library to provide the required geospatial operations.

The main task allotted to the clusterGIS library is to load one or more geospatial datasets in such a way that the GIS records are distributed between participating tasks. MPI tasks are grouped into communicators. By passing a specific communicator to a helper function, the tasks participating in the processing done by that function can be limited. MPI provides a default communication, MPI_COMM_WORLD, which contains all the tasks controlled by the MPI runtime.

Datasets can be loaded into participating tasks from a file by calling one of the clusterGIS load functions. Two main load functions exist: distributed and replicated. The replicated load function makes a copy the dataset on each task. The distributed load function splits the datasets's records among the participating tasks. The split is currently done byte-wise, meaning that each task is responsible for the records contained within a range of bytes in the source file. Records that overlap these boundaries are handled by each task dropping the first partial record and completed the final partial record. In this way no records are lost and the splitting process only requires a single complete read of the dataset with the addition of the overlapping sections.

Both the load functions take advantage of MPI-I/O. MPI-I/O is a collective I/O interface which optimizes how the data in a file is read from disk by communicating between the tasks participating in the collective operation. As disks are slow, this collaboration allows for decreased read time. MPI-I/O is also used for the write functions.

No matter if the load was distributed or replicated, the resulting local dataset is stored as a linked list. Each node in the linked list has an array containing the columns of the represented record. There is also a pointer for the geometry of the record. After the load, the geometry has not yet been created. ClusterGIS provides a function that will create all the geometries for a dataset.

After the data is loaded and the geometries created, the user can now proceed with the desired processing activities taking advantage of the full power of both MPI and GEOS.

When the user is ready to save a dataset, the write functions of clusterGIS are available. Like the load functions, the write functions come in distributed and replicated varieties. The replicated write function will write just one complete copy of the dataset out (each task has its own copy, but only one will be written to disk). The distributed write function writes all of the records for all of the participating tasks into one file.

The main idea of clusterGIS is to provide a few essential library functions that handle some of the basic, yet tricky, functions that a parallel program must implement. With these provided functions, the user can easily get to writing the more interesting and relevant parts of the processing program.

CHAPTER 5 EXPERIMENTAL SETUP

Chapter 3 defined the requirements for a good parallel GIS processing engine. Chapter 4 discussed the designs used in hadoopGIS and clusterGIS. Given these requirements and designs, the implementations can now be evaluated and compared against each other. As both these implementations are experimental, PostGIS will be used as a reference point to represent current methods. As such, it will also be evaluated as much as possible in the same manner as hadoopGIS and clusterGIS.

5.1 Standard Geospatial Operations

The first requirement is that an implementation can perform a set of standard geospatial operations. Allowing full access to a compliant geospatial library fulfills this requirement a priori. PostGIS uses, and in fact developed, the GEOS library. HadoopGIS makes use of JTS, from which the GEOS library was ported. ClusterGIS uses the GEOS library.

All three implementations fulfill this requirement, no further experimentation is needed.

5.2 Batch Mode Processing

Batch mode operation is essential to running operations on a cluster.

PostGIS extends the PostgreSQL database. PostgreSQL can be accessed through many different methods including programming interfaces in C and other languages, or through the provided command line client, “psql”. This client can be used interactively, or a sql script can be passed to it. That leaves the problem of the server. Either a dedicated server can be used, or a script can be written to create a server as needed to do the processing. This thesis sets a PostgreSQL server up as needed. PostGIS can perform batch mode processing.

Hadoop executes user jobs from a command line interface. The only hindrance for Hadoop to batch mode processing is the same as PostGIS: Hadoop is designed to have a persistent server environment setup. Projects like Hadoop on Demand allow a Hadoop environment to be created as needed. This thesis accomplishes a similar solution through a custom designed script. With this setup, HadoopGIS is capable of batch mode processing.

ClusterGIS only operates in a batch mode, there is no interactive mode.

5.3 Scalability

The first two requirements did not require any experimentation to evaluate. Scalability requires experimentation. This section details the experiments and specific implementation details. Chapter 6 presents and discusses the results of these experiments.

Verification of operations will be done by comparing output of each operation for HadoopGIS and ClusterGIS with the output from a PostGIS reference implementation.

As HadoopGIS and ClusterGIS are processing engines, scalability depends on the individual processing that is done. To achieve this, a set of processing operations is defined. After the operations are discussed in general, specific details for each implementation are discussed.

5.3.1 Operation set

One of the main tasks in parallelizing an algorithm is solving the problem of data access. HadoopGIS and clusterGIS split up the data that is being processed between different computers. The OGC SFS standard defines the operations that need to be supported. Table 5.1 shows a count of the number of operations in each section of the standard that defines operations along with the number of geometries required to execute that operation. One geometry is required for 42 of the operations, while 24 operations require two. There

Section	1 Geometry	2 Geometries
6.1.2 (Geometry)	16	15
6.1.4 (Point)	4	0
6.1.6 (Curve)	5	0
6.1.7 (LineString, Line, LinearRing)	2	0
6.1.8 (MultiCurve)	2	0
6.1.10 (Surface)	3	0
6.1.11 (Polygon, Triangle)	3	0
6.1.12 (PolyhedralSurface)	4	0
6.1.13 (MultiSurface)	3	0
6.1.15 (Relational Operators)	0	9
Total	42	24

Table 5.1: OGC Methods by Number of Required Geometries

are no operations that require more than two geometries.

The problem faced by HadoopGIS and ClusterGIS then is to get the data required to perform the geospatial operations required by what ever processing is required. The operations defined here are meant to be representative of what actual processing requirements would demand. The first four operations deal with individual record access. The following three operations deal with getting the data needed for a geospatial operation where it is needed.

Specific details on the datasets follow the operation descriptions. For now all that is needed to know is that there are two datasets, one of 34 thousand employers and one of 1.2 million parcels of land.

5.3.1.1 Create

The create operation adds a new record to an existing dataset. Adding a record is the basic operation required to build a new dataset. Record-centric systems merely add a record to the record store, while dataset-centric systems output a new dataset that contains the data of the input dataset with the new record included in it. Expected output is the original dataset with the addition of a single record.

The create operation implementations add a parcel of land to the parcels dataset.

5.3.1.2 Read

The read operation extracts a single record from a dataset based on a unique identifier. Record-centric systems are able to employ indexes and other methods to quickly locate a record whereas dataset-centric systems must scan an entire dataset to produce the single record. Expected output is a single record.

The read operation extracts a parcel from the parcels dataset.

5.3.1.3 Update

The update operation finds a single record from a dataset and changes some attribute of the record. Record-centric systems are able to employ indexes to locate and modify the record. Dataset-centric systems generate a new dataset with the changed record. Expected output is a dataset that contains all the records from the input dataset with exception that the specified record has been changed in the specified manner.

The update operation changes the land use code for a parcel in the parcels dataset.

5.3.1.4 Destroy

The destroy operation finds and removes a record from a dataset. Record-centric systems are able to find the record to be removed and then removing it from the record store. Dataset-centric systems generate a new dataset without the specified record. Expected output is a dataset with all the records from the input dataset except for the one that was removed.

The destroy operation removes a parcel from the parcels dataset.

5.3.1.5 Filter

The filter operation removes all records that don't fulfill a certain requirement, in this case all the records that don't intersect with a defined geometry. Record-centric systems do not have much of an advantage here as indexing the results of some geospatial operation is not usually worth the indexing cost, so both the record-centric and dataset-centric systems must scan the entire dataset. Expected output is a dataset containing all of the records from the input dataset that fulfill the filtering requirements.

The filter operation removes all parcels of land that don't overlap a defined region in the parcels dataset.

5.3.1.6 Nearest

The nearest operation is derived from an actual processing operation required for the Digital Phoenix Project[16]. This operation uses two datasets. The first is a set of points representing employers in the Phoenix metro-area. The second is a set of polygons representing parcels of land in the same area. Digital Phoenix needed to match the employer with the parcel of land where the business should be. Thus the operation is for each employer, find the nearest parcel of land with a compatible zoning code. The datasets used in this thesis have been simplified to make this matching appear more straight forward. Expected output is a list of employer ids with associated parcel ids.

The nearest operation uses both the employers and parcels datasets.

5.3.1.7 Chaining

The chaining operation combines the filter and nearest operations with the intent of showing how multiple operations can be performed one after the other. This operation is an obvious optimization of the nearest operation in that it first removes all residential parcels before running the nearest operation on the remaining parcels. As employers cannot (in this simplified world) exist on residential parcels, and since residential parcels make up a large portion of the entire parcel dataset, the number of distances calculated for each employer will be significantly decreased thus decreasing processing time. Expected output is the same as for the nearest operation.

The chaining operation uses both the employers and parcels datasets.

5.3.2 Dataset Description

Two datasets are used in evaluating these operations. The first is a set of employers where each record contains an employer id, the place where the employer is located represented as a point, and the business classification: commercial, industrial, or governmental. The employer dataset contains 34,302 records.

The second dataset is a set of parcels where each record contains a parcel id, a multipolygon representing the parcel coverage, and a land use code: residential, commercial, industrial, or governmental. The parcel dataset contains 1,218,130 records.

Both datasets use R, C, I, and G to represent residential, commercial, industrial, and governmental use codes. These datasets are simplified versions of real datasets for Maricopa County, Arizona. The datasets were simplified by adding the simplified land use code attribute and removing the other attributes not used in these operations.

5.4 Execution Environment

All operations will be executed on ASU's Saguaro cluster. The Saguaro cluster is comprised of several generations of hardware. To simplify comparisons, all operations will be executed on similar hardware with similar network interconnects, using the one most natural for use.

Saguaro is a CentOS 5.3 based cluster running on Intel Xeon processors. Each node used in these experiments has two harpertown processors each with four cores running at 2.83GHz and has 16GB of RAM. In addition each node has one gigabit ethernet connection and one DDR InfiniBand connection. The shared filesystem for these experiments is Luster 1.6.6 accessed through the InifiniBand connection.

5.5 PostGIS Implementation

The PostGIS experiments were ran using PostgreSQL 8.4.1 and PostGIS 1.3. Both pieces of software were compiled for these tests and used default options.

Each of the following operations was executed on a freshly setup PostGIS instance, running on local disk with default options. Data was loaded, indexed, and vacuumed before the experiment was executed. Runtimes were gathered by using the linux ‘time’ command. When a query’s result did not modify an existing table, its results were stored into another table.

Full SQL statements for each operation are listed below.

5.5.1 Create

The create operation for PostGIS as shown in listing 5.1 is a simple SQL insert statement.

```
insert into parcels values (97123897, ST_GeomFromText('
    POLYGON((-112.0859375 33.4349975585938,-112.0859375
    33.4675445556641,-112.059799194336
    33.4675445556641,-112.059799194336
    33.4349975585938,-112.0859375 33.4349975585938))', 4326)
    , 'C');
```

Listing 5.1: PostGIS Create Operation

The most complicated portion of this operation is creating the geometry type. PostGIS provides several functions that can create geometry types from text representations. The one used here, ST_GeomFromText, creates a geometry from the standard Well Know Text format.

5.5.2 Read, Update, and Destroy

The read, update, and destroy operations use the basic SQL select, update, and delete statements. Listing 5.2 shows the update operation which is representative of all three of these operations.

5.5.3 Update

```
update parcels set devtype = 'C' where id = 1008130;
```

Listing 5.2: PostGIS Update Operation

5.5.4 Filter

The Filter operation shown in Listing 5.3 is the first of these operations to take advantage of the processing capabilities of PostGIS.

```
create table output as
select parcels.* from parcels, (select ST_GeomFromText('
    POLYGON((-112.0859375 33.4349975585938,-112.0859375
    33.4675445556641,-112.059799194336
    33.4675445556641,-112.059799194336
    33.4349975585938,−112.0859375 33.4349975585938))', 4326)
     as the_geom) as box
where ST_Intersects(parcels.the_geom, box.the_geom);
```

Listing 5.3: PostGIS Filter Operation

The operation returns all the records that intersect with a predefined geometry, in this case created using ST_GeomFromText. ST_Intersects was chosen to perform the intersection because it does not utilize the bounding box based index provided by PostGIS. As the operation is not influenced by that index, which can only be used for a few operations, this query is more representative of real-world conditions and more complicated filtering queries.

5.5.5 Nearest and Chaining

The chaining operation is defined as comparing all employers to all parcels except for those parcels with a residential land-use code. PostGIS automatically performs this optimization when the land-use code field is indexed, therefore the nearest and chained operations are equivalent.

```
begin;

—— Figure out the distance to all applicable parcels
create index jobs_devtype_index on jobs using btree ("
    devtype");
create index parcels_devtype_index on parcels using btree
    ("devtype");
create temp table distances on commit drop as
```

```
select jobs.id as job_id, parcels.id as parcel_id, distance
   (jobs.the_geom, parcels.the_geom)
from jobs, parcels
where jobs.devtype = parcels.devtype;

-- Find the distance to the closest parcel
create temp table min on commit drop as
select job_id, min(distance) as distance
from distances
group by job_id;

-- Match the job to the closest parcels
create index distances_distance_index on distances using
   btree (”distance”);
create index min_distance_index on min using btree (”
   distance”);
create temp table nearest on commit drop as
select distances.job_id, distances.parcel_id
from distances, min
where distances.job_id = min.job_id
and distances.distance = min.distance;

-- Only use unique job_id's
create index nearest_job_id_index on nearest using btree (”
   job_id”);
```

```
create table ”jobs_parcels” as
select nearest.job_id, nearest.parcel_id
from nearest
where nearest.parcel_id = (select n.parcel_id from nearest
    as n where n.job_id = nearest.job_id limit 1);

commit;
```

Listing 5.4: PostGIS Nearest and Chained Operations

When developing this operation, it was hoped that a single nested query would be sufficient. Unfortunately the nested query performed significantly slower than the optimized query shown in Listing 5.4. Indexes beyond the standard primary key index are created as part of this query.

The first step in the query process is to create a distance table between every job and ever parcel with matching devtypes. The next step determines what the minimum distance is for each employer. The third step matches the previously found minimum distance with all the parcels that are that distance away. The final step reduces the number of parcels that are equidistant (using the minimum distance) from each employer to one.

5.6 HadoopGIS Implementation

The hadoopGIS experiments were executed using Hadoop 0.19.0, data was accessed on an HDFS running on the same nodes off local disk. Internode communication was accomplished through Gigabit Ethernet.

Default values were mostly used. Changes were made to increase the number of mappers per node to 8, increasing the memory available to each JVM, and altering the HDFS block size to produce the desired number of mappers.

Core algorithm details are described below. In all cases the reduce function is an identity function where all input is passed through as output, therefore the including listings in this section display code that is part of the map function. Full listings are available in the appendix.

5.6.1 Create

The Hadoop architecture makes it difficult to arbitrarily output a record. The processing method is fully dependent on the input records. Since it was not possible to add a record to the output dataset without relying on some input, like is possible for PostGIS and clusterGIS, this operation creates the new record when triggered by a record that already exists in the dataset. The resulting map function is shown in Listing 5.5.

```
if(key == 1008130) {
        GIS created = new GIS();
        GIS.update(”\”97123897\”,\”POLYGON((−112.0859375
           33.4349975585938,−112.0859375
           33.4675445556641,−112.059799194336
           33.4675445556641,−112.059799194336
           33.4349975585938,−112.0859375 33.4349975585938))
           \”,\”C\””);
        output.collect(new LongWritable(new Integer
           (97123897)), created);
}
output.collect(key, value);
```

Listing 5.5: HadoopGIS Create Operation

5.6.2 Read, Update, and Destroy

The read, update, and destroy operations are very similar. Listing 5.6 shows the simplicity of the update operation. All three operations use a little bit of logic to determine if a record is passed onto the reduce phase. The read operation only keeps the requested record; update modifies a specific record before passing it on, and destroy passes on all the records except for the one specified.

5.6.3 Update

```
if(key.equals(new LongWritable(1008130))) {
        value.attributes.put(”devtype”, ”C”);
}
output.collect(key, value);
```

Listing 5.6: HadoopGIS Update Operation

5.6.4 Filter

The filter operation works in a similar manner as the read, update, and destroy operations as shown in Listing 5.7.

```
if(value.geometry.intersects(box)) {
        output.collect(key, value);
}
```

Listing 5.7: HadoopGIS Filter Operation

The intersects function requires a geometry. To keep from recreating the geometry filtered against, the geometry is created in the configure method which is ran at the creation of the mapper, and therefore is only referenced in the map function.

5.6.5 Nearest and Chaining

The nearest operation uses the employers dataset as input for the map functions. As the operations compares every employer against every parcel, the parcel dataset is loaded in its entirety in the configure method (ran when creating each mapper). This way of using a secondary dataset is not optimal, but it is the only method allowed in the map-reduce framework. Being forced to include a full copy of the parcels dataset increases the memory requirement for each mapper, thus limited the number of mappers that can be ran on a single machine and the maximum size of the parcel dataset. The clusterGIS implementation presents a solution for this problem that is possible in MPI but not Hadoop.

Even with that problem, the resulting map function is quite simple as shown in Listing 5.8.

```
double minDistance = Double.MAX_VALUE, currDistance;
int closestParcel = -1;

Iterator it = parcels.iterator();
while (it.hasNext())
{
        GIS parcel = (GIS) it.next();

        currDistance = value.geometry.distance(parcel.
           geometry);
        if (currDistance < minDistance)
        {
                minDistance = currDistance;
                closestParcel = new Integer(parcel.
                    attributes.get(”id”));
        }
}

LongWritable lngClosestParcel = new LongWritable(
    closestParcel);
output.collect(key, lngClosestParcel);
```

Listing 5.8: HadoopGIS Nearest Operation

The map function uses a simple algorithm to find the minimum distance to a parcel and then outputs the id of both the employer and the parcel.

The chaining operation reduces the memory problems encountered in the nearest operation implementation by removing all residential parcels. This optimization works for this specific problem but cannot be generally applied. As this optimization is applied during the mapper's configure function, the algorithm used to find the nearest parcel does not change.

5.7 ClusterGIS Implementation

ClusterGIS was compiled with the Intel C Compiler (icc) 10.1 20080312 against MVAPICH 1.0.1. MVAPICH makes use of the DDR InfiniBand network connection for communication. Dataset storage is done on Lustre, which is connected through the same InifiBand connections.

The specific main portion of each operations algorithm is show below. Dataset loading and writing is accomplished using the distributed method including all tasks in the operation unless otherwise stated. Full code listings are available in the appendix.

5.7.1 Create

Unlike the hadoopGIS create operation, it is easy to arbitrarily add a record using clusterGIS. In this case the first tasks creates a new record using a helper function provided by clusterGIS and then inserts it into its local dataset. When the dataset is written collectively, the new record is included.

```
if(rank == 0) {
        int start = 0;
        record = clusterGIS_Create_record_from_csv
           (”97123897,POINT(0 0),C\n”, &start);
        record->next = dataset->data;
        dataset->data = record;
}
```

Listing 5.9: ClusterGIS Create Operation

5.7.2 Read, Destroy

The read and destroy operations are implemented in a similar manner. Listing 5.10 shows the main portion of the read operation.

```
record = dataset->data;
head = &(dataset->data);
/* keep records that match the criteria, otherwise delete
   them */
while(record != NULL) {
        if(atoi(record->data[0]) == 1008130) {
                head = &(record->next);
                record = record->next;
        } else {
                *head = record->next;
                clusterGIS_Free_record(record);
                record = *head;
        }
}
```

Listing 5.10: ClusterGIS Read Operation

This implementation looks more complicated than the corresponding hadoopGIS operations, but performs the same processing. ClusterGIS provides all the records in a dataset as a linked list, which is not as pretty to manipulate as the Java container objects.

This code loops through each record in the linked list. The record variable points to the current record. The head variable points to the pointer in the previous record that points to the current record. To keep a record, such as is done in the then block of the if statement, it is only necessary to update these variable to the next record. To delete a record, head is set to point to the next record and then the current record is deleted from memory. Record is then advanced to the next record.

5.7.3 Update

The update operation as shown in Listing 5.11 shows the simplest way to loop through a linked list. Each record is checked to see if it matches the criteria, and then the record is updated.

```
record = dataset->data;
```

```
/* keep records that match the criteria, otherwise delete
   them */
while(record != NULL) {
        if(atoi(record->data[0]) == 1008130) {
                record->data[2] = ”C”;
        }
        record = record->next;
}
```

Listing 5.11: ClusterGIS Update Operation

5.7.4 Filter

The filter operation as shown in Listing **??** builds on the structure used in the read operation. Instead of comparing against an id number, the filter operation uses a geometric intersection to determine if the record should be kept. The shown code can easily be extended to use an arbitrarily complex filtering parameter.

```
box = GEOSWKTReader_read(reader, ”POLYGON((-112.0859375
   33.4349975585938,-112.0859375
   33.4675445556641,-112.059799194336
   33.4675445556641,-112.059799194336
   33.4349975585938,-112.0859375 33.4349975585938))”);

record = dataset->data;
head = &(dataset->data);
```

```
/* keep records that match the criteria, otherwise delete them */
while(record != NULL) {
        char intersects;

        intersects = GEOSIntersects(record->geometry, box);

        if(intersects == 1) { /* record overlaps with box */
                head = &(record->next);
                record = record->next;
        } else { /* no overlap */
                *head = record->next;
                clusterGIS_Free_record(record);
                record = *head;
        }
}
```

Listing 5.12: ClusterGIS Filter Operation

The two main differences show how to create a new geometry object using GEOSWKTReader and how to use the GEOSIntersects function. Beyond these differences the loop structure is the same.

5.7.5 Nearest and Chaining

Upto this point, all the operations for clusterGIS have only had to work on one dataset. The nearest and chained operations work on two. Unlike hadoopGIS, clusterGIS is able to split up both datasets between participating tasks. The tested implementation places a full copy of the parcels dataset on each computer, which has eight tasks running on it, where each of those tasks has the same set of employers. This means that each task will find the closest parcel for each employer of the parcels that it has (i.e., a local minimum). The tasks sharing the same employers then talk amongst themselves to determine the globally nearest parcel. This communication is done through MPI's reduce mechanism. The employer id now associated parcel id are added to the output dataset.

The nearest operation implementation and the chaining operation implementation are very similar. Listing 5.13 shows the chaining operation.

```
/* remove all residential parcels */
parcel = parcels->data;
head = &(parcels->data);
while(parcel != NULL) {
        if(strncmp(parcel->data[2], ”R”, 1) == 0) {
                head = &(parcel->next);
                parcel = parcel->next;
        } else {
                *head = parcel->next;
                clusterGIS_Free_record(parcel);
                parcel = *head;
        }
}

/* Find the min distance */
employer = employers->data;
```

```
min = malloc(sizeof(double)*2);
global_min = malloc(sizeof(double)*2);
output = clusterGIS_Create_dataset();
while(employer != NULL) {

        /* find the local min */
        parcel = parcels->data;
        GEOSDistance(employer->geometry, parcel->geometry,
           &min_distance);
        min_distance_parcel = parcel;
        while(parcel != NULL) {
                if(strncmp(employer->data[2], parcel->data
                   [2], 1) == 0) {
                        GEOSDistance(employer->geometry,
                           parcel->geometry, &distance);
                        if(distance < min_distance) {
                                min_distance = distance;
                                min_distance_parcel =
                                   parcel;
                        }
                }
                parcel = parcel->next;
        }

        /* find the global min */
        min[0] = atoi(min_distance_parcel->data[0]);
        min[1] = min_distance;
        MPI_Allreduce(min, global_min, 2, MPI_DOUBLE,
            min_distance_op, parcels_comm);

        /* Add the min to the output dataset using front
            insertion */
        sprintf(output_csv, ”\”%s\”,\”%d\”\n”, employer->
            data[0], (int) global_min[0]);
        start = 0;
        output_record = clusterGIS_Create_record_from_csv(
            output_csv, &start);
        output_record->next = output->data;
        output->data = output_record;

        employer = employer->next;
}
```

Listing 5.13: ClusterGIS Chaining Operation

The difference between nearest and chaining is the addition of the first loop, which removes all residential parcels from the parcel dataset. With the exception of this loop, the rest of the listing reflects the nearest operation's implementation.

The main loop finds the globally nearest parcel for each employer. Each task first finds a locally nearest parcel using the GEOSDistance function to calculate distance. The nearest parcel id and distance are store in the min array, which is then used to find the global minimum.

The MPI_Allreduce command with the help of the min_distance_op, determines the minimum distance of all the min arrays for the tasks which share the same set of employers. After the function finishes, each task in the group have the same values their min array. This is the global minimum.

The last section of this code creates new records which are added to the output dataset. This dataset can then be used just like any other dataset loaded in clusterGIS.

CHAPTER 6 RESULTS

The main purpose of pursuing parallel methods for GIS processing is to overcome the limitations of GIS processing using desktop applications. This thesis uses PostGIS, the GIS extension for the PostgreSQL relational database, to represent the capabilities of more traditional GIS processing applications.

Given the capabilities and sophistication of a relational database, record level operations performed in PostGIS are expected to outperform the simpler algorithms used in hadoopGIS and clusterGIS. While there are not limitations against using similarly sophisticated algorithms in the parallel implementations, this thesis attempts to show what can be done with relatively simple and naive algorithms, such as could be created by people not possessing computer science degrees.

On the other hand, hadoopGIS and clusterGIS should outperform PostGIS in operations requiring access to entire datasets. This is expected because unless the dataset level operation can be done entirely using indexes, which in the operations selected here is not the case, PostGIS is forced to work with every record in the dataset; leaving it to a basic serial algorithm. HadoopGIS and clusterGIS attempt to

This chapter evaluates these assumptions and compares the relative performance and scalability of hadoopGIS and clusterGIS by first discussing characteristics of the record level operation experiments and then the dataset level operations. After these assumptions are evaluated, some remarks will be made on the ease of using each of these three environments for GIS processing.

6.1 Record Level Operations

The classic operations to determine ability to perform record level operations are to create a new record, read an existing record, update an existing record, and delete an existing

Implementation	Time (seconds)	Processor Cores
PostGIS	0.012	1
hadoopGIS	77.453	0
clusterGIS	6.160	128

Table 6.1: Fastest Execution Time for Create Operation

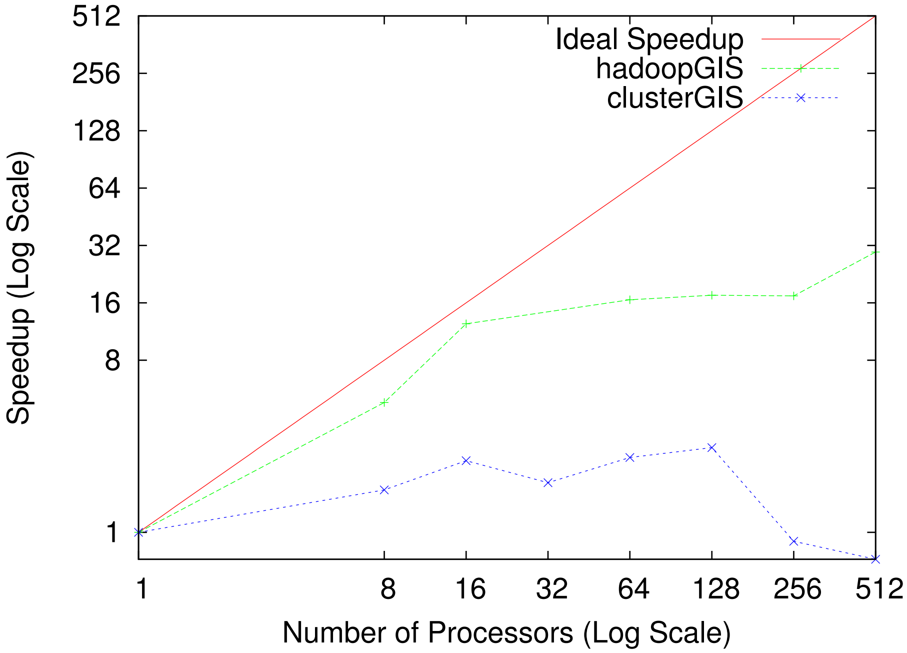


Figure 6.1: Create Speedup

record from a dataset. Of these four operations, create, update, and delete exhibited similar performance characteristics in both PostGIS and the parallel implementations. Read performance in the parallel implementations was markedly different.

As performance characteristics were clustered in this manner, the create operation will be used to represent the performance of the create, update, and delete operations. The read operation will represent itself.

6.1.1 Create

The create operation inserts a record into a dataset. In the case of PostGIS, an existing dataset is mutated to include the new record. HadoopGIS and clusterGIS, as they work on the dataset level, each create a new dataset including all the records from the original dataset and the new record.

Table 6.1.1 lists the fastest times that were executed in the series of tests along with the number of processors used. PostGIS is markedly faster than either of the parallel implementations. This is largely due to the fact that PostGIS is not required to both read and write the entire dataset, but only has to add the single record, confirm that the record does not violate any constraints of the dataset, and update any indexes. Even with the additional meta-work required, the significant difference between execution times allows for a large amount of additional work to be performed.

This number is for the addition of one record to a dataset. The parallel implementations have a large amount of overhead in reading and writing an entire dataset of 1.2 million records. Assuming that creating additional records was free, meaning each additional record added 0.000 seconds to the execution time, one would have to create 514 records using the faster of the parallel implementations, clusterGIS, to amortize the overhead.

Figure 6.1.1 shows how hadoopGIS and clusterGIS perform with the addition of more computers. While the raw speed of hadoopGIS and clusterGIS differ greatly as shown in Table 6.1.1, the speedup graph helps us to ignore the differences in implementations of these these two technologies and see how additional resources can improve performance.

Both implementations are limited by I/O. Neither implementation has much computation to do, as they just read the data in and then write it back out with the additional record.

Implementation	Time (seconds)	Processor Cores
PostGIS	0.023	1
hadoopGIS	49.324	0
clusterGIS	2.015	16

Table 6.2: Fastest Execution Time for Read Operation

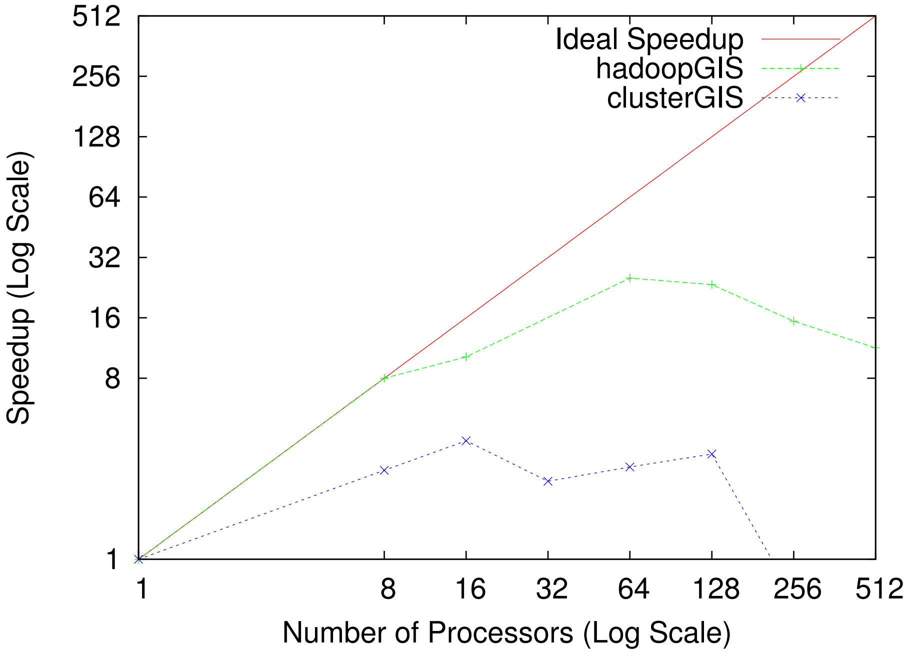


Figure 6.2: Read Speedup

As was expected, PostGIS is able to outperform hadoopGIS and clusterGIS in creating records. While not tested, it appears that at least 514 records would have to be created in order to give the parallel implementations a chance of outperforming PostGIS.

6.1.2 Read

The read operation is similar to the create operation for the parallel implementations in that the entire dataset must be read in. The difference is that only one record must be written to disk. PostGIS is able to use an index to quickly locate the requested record and then read just that data from disk and then output it to a new table. Table 6.1.2 shows PostGIS to still be the fastest, as compared to the create operation, but that the parallel implementations are significantly faster even with the increased computation requirement of checking every single record to see if it was the record.

The speedup shown in figure 6.1.2 is higher than for the create operation due to the decreased I/O and slightly increased computation requirements. While the operations are still I/O bound, the difference is significant. Also note the point of diminishing returns is met much more quickly, at 16 cores for the clusterGIS implementation, than for the create operation. All of the experimental runs for clusterGIS have a peak at 16 cores, which indicates the datasize, distribution of data blocks on the Lustre filesystem setup, and network setup somehow have the best setup for redistributing the data.

As expected, PostGIS is able to outperform hadoopGIS and clusterGIS in reading a single record from a dataset while using less resources. This is once again due to more sophisticated algorithms, data storage methods, and indexes. Again it is not unreasonable that such methods could be used to increase the performance of the parallel implementations.

6.2 Dataset Operations

Now that basic performance and capabilities for record level operations have been established, and have been shown to be weaknesses for both parallel implementations, at least with the current naive and simple algorithms used, we come to the operations where the parallel implementations are expected to outperform PostGIS. These operations utilize entire datasets, first by filtering a dataset using a geometric computation, and then by utilizing two datasets in their entirety.

A scaling issue in hadoopGIS for the second dataset was encountered while running

Implementation	Time (seconds)	Processor Cores
PostGIS	1.123	1
hadoopGIS	50.408	0
clusterGIS	2.957	128

Table 6.3: Fastest Execution Time for Filter Operation

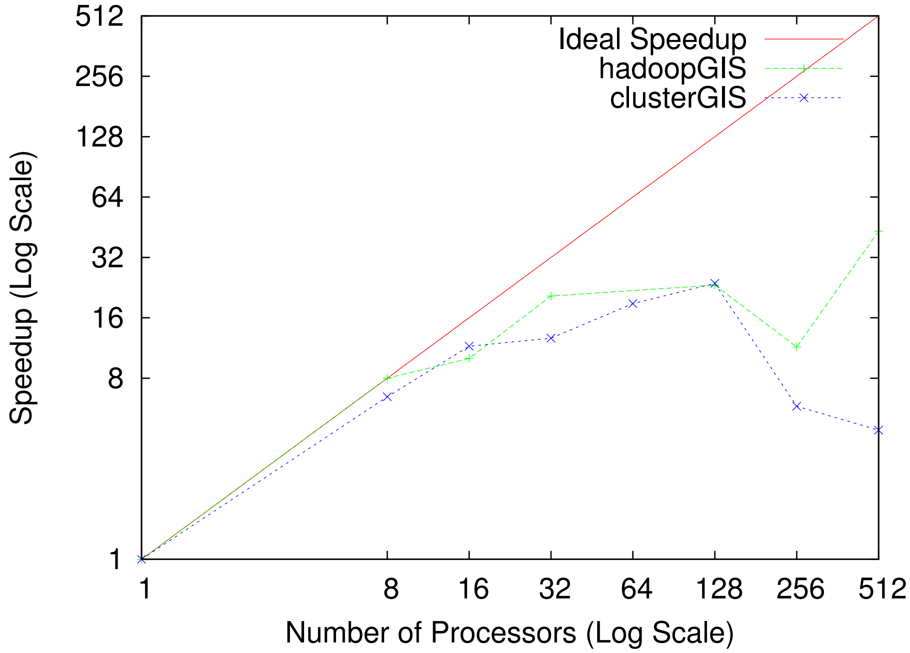


Figure 6.3: Filter Speedup

the nearest and chained operations. As a result, only the chained operation is show here with a description of the issue and how the clusterGIS implementation solves the same problem. ClusterGIS exhibits a similar speedup line for both the nearest and chained operations, and it is therefore assumed that if more memory were available that the hadoopGIS implementation's speedup would be similar to that of the chained operation. As such, only the chained operation is discussed here.

Implementation	Time (seconds)	Processor Cores
PostGIS	198000.00	1
hadoopGIS	87621.129	0
clusterGIS	41.995	1024

Table 6.4: Fastest Execution Time for Chained Operation

6.2.1 Filter

The algorithms used for the filter operation are quite similar to those used in the read operation, but that instead of doing a simple comparison of record ids, a geometric computation is required. The intersection operation was chosen to represent any generic geometric operation in that it cannot utilize indexes currently available in PostGIS.

Comparing the execution times in table 6.2.1 to those of the read operation (table 6.1.2) we see that clusterGIS is able to absorb the additional computation requirements by adding more processing cores while only increasing computation time by one second. HadoopGIS performs in similar manner. PostGIS is the most hard hit by its inability to utilize additional resources, thus significantly increasing processing time.

Speedup, as shown in figure 6.2.1 is much more near ideal speedup than the record level operations. The increased computation is able to offset the I/O requirements, reducing the effect of the I/O bottleneck and increasing efficiency.

As more computation is required to process each record, the parallel implementations are able to utilize more and more resources to reduce total computation time. PostGIS's execution time can only increase when more computation is required per record.

6.2.2 Chained

Finding the nearest parcel for each employer is a fairly normal GIS question to ask. The fastest execution times shown in table 6.2.2, however, show significant differences in the

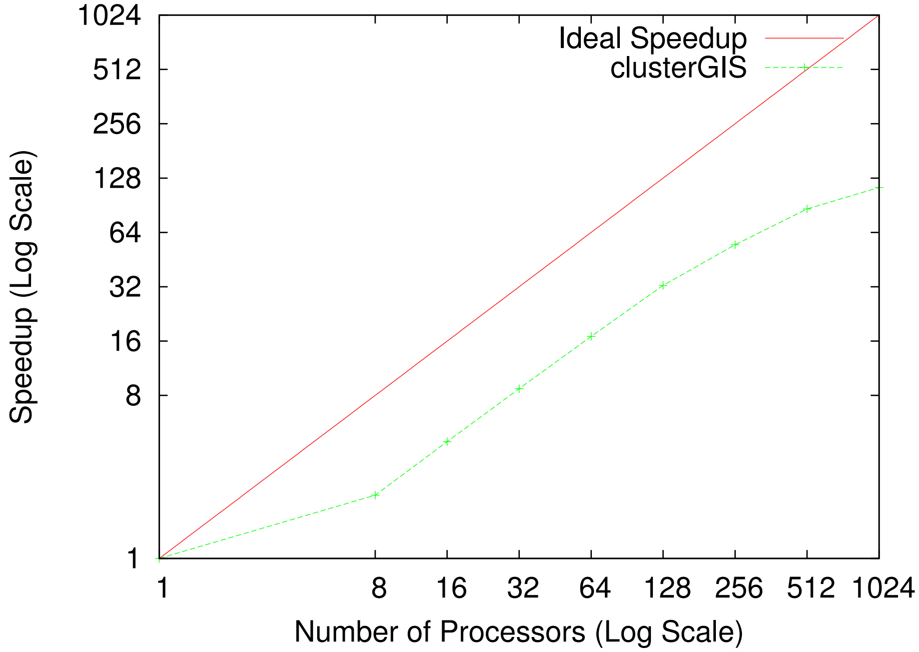


Figure 6.4: Chained Speedup

capabilities of each implementation. ClusterGIS by far outperforms the other implementations.

A significant scaling problem was discovered the in architecture of the hadoopGIS implementation. The map reduce paradigm expects apply a map function to ever record in the input dataset. The nearest and chained operations require the use of a second dataset, the parcel dataset, to compare against for each employer. Therefore each execution of the map function requires full access to the parcel dataset. Hadoop provides the capability to load a copy of the parcel dataset into a mapper. Mappers are the part of the Hadoop architecture that executes the map functions. While the parcel dataset is not required to be read from disk for each map invocation, it is required to be available in the mapper. The problem comes in that the parcel dataset is quite large, with each mapper requiring 2.2 GB of RAM to store all the parcels except for residential parcels in memory. Because of the simplicity of the land use code matching this optimization was available for this case and was originally designed as the chained experiment.

While 2.2GB of RAM is not really that much, the computers used have a total of 16GB of RAM that is shared between 8 cores. This averages to 2GB of RAM per core (assuming no memory utilization by the operating system), which is less than the 2.2GB required. As each mapper executes a single thread of computation, 2 processors cores on each machine are left idle because of this memory requirement.

While it is possible to read in the entire parcel dataset for each map function, this is much slower due to the slow speed of disk as compared to memory. Another possible method to reduce memory usage, and thus increase the compute capabilities in this computation bound operation would be to read in the parcel data in chunks, finding local minimum distances for each employer in that chunk and then processing the next chunk and so on. This method could only be performed using the map-reduce paradigm by running multiple map-reduce processes. The first process would split the parcel dataset into manageable chunks, after which a process for each chunk would find local minimum for that chunk, and then a final process would reduce the local minimums to a global minimum. Such a process is possible to implement, but is somewhat unwieldily.

The clusterGIS solution to this same scaling problem is to modify how the data is split. The implementation used creates blocks of eight MPI tasks. Each block has a full copy of the parcel dataset, with each task in the block having just a portion. Instead of dividing the employer dataset between all the tasks, the tasks in each block contain the same set of employers. The block size is tunable such that the block size can be adjusted allowing for a specific implementation to be optimized for memory usage and problem granularity. This implementation was not tuned for optimal performance, but for explainability in that each computer has a full copy of the parcel dataset and a subset of the employers where each task has the same employers.

The speedup shown in figure 6.2.2 for clusterGIS is nearly linear for most of its run. The speedup starts to drop off starting at 256 cores, probably due to the small size of the employer dataset. At 1024 cores the employer dataset's 34,000 records are split into 128 sections. The increased work to split and the recombine the dataset starts to affect the granularity of the problem at that scale. A larger employer dataset or more computation per record would increase the number of cores required before this effect would be noticed. The performance is still impressive, converting each hour of execution time used by PostGIS into less than one second.

Though hadoopGIS experiences a severe limitation on the size of the parcel dataset, it is also able to produce results more quickly than PostGIS while having a reasonable speedup line.

As was expected, the parallel implementations are able to outperform PostGIS. Of the two, clusterGIS is able to scale computation capabilities along with the size of both datasets.

CHAPTER 7 CONCLUSION

This thesis evaluated two parallel GIS processing paradigms, map reduce and message passing by creating an implementation in each paradigm and then comparing the performance of each implementation.

The map reduce paradigm applies a map function to every record in a dataset and then reduces the output of those function calls to produce a final result. The Hadoop map reduce framework was extended to include support for GIS data creating the hadoopGIS implementation.

The message passing paradigm uses a set of tasks that are able to communicate with each other by passing messages. The clusterGIS processing environment was created using MPI.

A set of operations was developed to assist in the evaluation. Two major types of operations were made, record centric and dataset centric. The record centric operations only affected a single record in a dataset, while dataset centric operations worked on every record in one or more datasets. As these operations were performed in through the lens of geometric processing, and as the maximum number of geometries required by the basic geometric operations as defined by the Open Geospatial Consortiums's Simple Features standard[2] is two, any operation could be accomplished if two datasets were available.

Traditional serial GIS processing was represented by PostGIS, which extends the PostgreSQL relational database to be able to perform geospatial processing.

As expected, PostGIS was able to perform the record-centric operations more quickly than either hadoopGIS or clusterGIS. This is due to the fact that both the parallel implementations must read in and then write out the entire dataset to affect any one record. It is possible that this overhead could be amortized should enough new records be created.

The record centric operations formed two performance clusters. Creating, updating, and deleting records output basically the entire dataset. Reading records, on the other hand, significantly reduced the amount of data output, thereby increasing speedup.

Dataset centric operations were performed more quickly by the parallel implementations, hadoopGIS and clusterGIS. In the filter operation, where a subset of the records was retrieved using a geometric operation, hadoopGIS and clusterGIS scaled similarly. Severe scalability problems were discovered in hadoopGIS when using more than one dataset. The secondary dataset is unable to be larger than available memory. ClusterGIS was able to solve this problem by distributing both datasets and combining the partial answers into the final answer.

In terms of usability, PostGIS had the shortest implementations, with the next largest being clusterGIS and then hadoopGIS. PostGIS uses SQL and a relational database, both of which require specialized skills to utilize effectively. Though simple SQL statements can be written, getting performance out of the database requires knowledge of indexes and processing paths. Both hadoopGIS and clusterGIS are more explicit in how the processing is done, and therefore it is easier to see what is being done. ClusterGIS is more flexible is how processing can be accomplished, allowing for easier handling of additional datasets processing methods. HadoopGIS provides a fairly easy to understand processing method, but lacks the ability to work in ways that are not provided for in that processing method. The main example of this is its inability to scale on the secondary dataset as using a secondary dataset is not provided for in the map reduce paradigm.

HadoopGIS and clusterGIS show that parallel GIS processing is possible and can afford significant decreases in processing time while using simple algorithms in cases where one or more datasets must be processed in their entirety. More sophisticated algorithms could be developed to provide greater benefits.

7.1 Future Work

The operations created for this thesis provide examples of how to use hadoopGIS and clusterGIS. These examples can be used as building blocks to create programs that solve more complicated problems to to enable parallel simulation methods.

The algorithms presented in this thesis were intentionally simplistic and naive. More advanced methods than just looping through the datasets such as using indexes have the potential to further improve the provided capabilities.

Another improvement that can be made is the addition of alternate decomposition methods. These decomposition method used in this thesis was very simple and just split up the records between computers. More sophisticated decomposition methods such as a geospatial decomposition where record are distributed by locality would be beneficial for many processing methods.

CHAPTER Bibliography

[1] M. J. Flynn, “Very high-speed computing systems,” *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[2] O. S. G. Consortium, “Opengis implementation specification for geographic information - simple feature access - part 1: Common architecture,” standard, Open Geospatial Consortium, Inc., 2006.

[3] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] V. Solutions, “Java topology suite,” *http://www.vividsolutions.com/jts/jtshome.htm*, 2009.

[5] M. P. I. Forum, “Mpi: A message-passing interface standard,” *International Journal of Supercomputer Applications*, 1994.

[6] O. S. G. Foundataion, “Geos - geometry engine, open source,” *http://trac.osgeo.org/geos/*, 2009.

[7] M. D. Michael Worboys, *GIS: a computing perspective*. CRC PRess, 2004.

[8] P. Schut, “Back from the brink: the story of the remarkable resurrection of the canada land inventory data,” *http://www.igs.net/ schut/cli.html*, 2000.

[9] T. Ormsby and R. Burke, *Getting to Know ArcGIS Desktop: Basics of ArcView, ArcEditor, and ArcInfo*. ESRI, Inc., 2nd ed., 2004.

[10] J. Gray, “Quantum gis: the open-source geographic information system,” *Linux J.*, vol. 2008, no. 172, p. 8, 2008.

[11] M. Neteler and H. Mitasova, *Open source GIS: a grass GIS approach*. Springer, 3rd ed., 2002.

[12] R. West, *Understanding ArcSDE*. ESRI, Inc., 2001.

[13] D. Blasby, “Building a spatial database in postgresql,” in *Open Source Database Summit*, 2001.

[14] S. Ravada and J. Sharma, “Oracle8i spatial: Experiences with extensible databases,” in *SSD ’99: Proceedings of the 6th International Symposium on Advances in Spatial Databases*, (London, UK), pp. 355–359, Springer-Verlag, 1999.

[15] R. Pahle and N. Kerr, “A datacentric framework for research in planning,” in *UPE8, The 8th International Symposium (UPE 8) of the International Urban Planning and Environment Association*, 2009.

[16] S. Guhathakurta, Y. Kobayashi, M. Patel, J. Holston, T. Lant, J. Crittenden, K. Li, G. Konjevod, and K. Date, “Digital phoenix project: A multidimensional journey through time.” Not published, 2006.

[17] P. Waddell, A. Borning, M. Noth, N. Freier, M. Becke, and G. Ulfarsson, “Microsimulation of urban development and location choices: Design and implementation of urbansim,” *Networks and Spatial Economics*, vol. 3, pp. 43–67, 2003.

[18] Y. K. Luc Anselin, Ibnu Syabri, “Geoda: An introduction to spatial data,” *Geographical Analysis*, vol. 38, pp. 5–22, 2006.

[19] L. A. Sergio J. Rey, “Pysal: A python library of spatial analytical methods,” *The Review of Regional Studies*, vol. 37, no. 1, pp. 5–27, 2007.

[20] D. Guidi, “Nettopologysuite,” *http://code.google.com/p/nettopologysuite/*, 2009.

[21] P. J. Braam, “Lustre file system,” white paper, Cluster File Systems, Inc., 2007.

[22] G. Mackey, S. Sehrish, J. Bent, J. Lopez, S. Habib, and J. Wang, “Introducing mapreduce to high end computing,” in *Petascale Data Storage Workshop, 2008. PDSW ’08. 3rd*, pp. 1–6, Nov. 2008.

[23] F. Huang, D. Liu, P. Liu, S. Wang, Y. Zeng, G. Li, W. Yu, J. Wang, L. Zhao, and L. Pang, “Research on cluster-based parallel gis with the example of parallelization on grass gis,” in *Grid and Cooperative Computing, 2007. GCC 2007. Sixth International Conference on*, pp. 642–649, Aug. 2007.

[24] T. Corp., “Dbc/1012 data base computer concepts & facilities,” tech. rep., Teradata Corp Document No C02-0001-00, 1983.

[25] J. Patel, J. Yu, N. Kabra, K. Tufte, B. Nag, J. Burger, N. Hall, K. Ramasamy, R. Lueder, C. Ellmann, J. Kupsch, S. Guo, J. Larson, D. De Witt, and J. Naughton, “Building a scaleable geo-spatial dbms: technology, implementation, and evaluation,” in *SIGMOD ’97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 336–347, ACM, 1997.

[26] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J.-B. Yu, “Client-server paradise,” in *VLDB ’94: Proceedings of the 20th International Conference on Very Large Data Bases*, (San Francisco, CA, USA), pp. 558–569, Morgan Kaufmann Publishers Inc., 1994.

APPENDIX A HADOOPGIS SOURCE

The complete source for hadoopGIS is included in this appendix. HadoopGIS is also available at `http://github.com/nathankerr/hadoopGIS`.

The source is divided into two sections, the core code and the examples.

A.1 Core

These listings comprise the core components of hadoopGIS, implementing the GIS datatype and the facilities to extract GIS records from a file into the mapper, pass GIS data from mappers to reducers, and from reducers back to a file.

A.1.1 GIS.java

```
package hadoopGIS;

import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;

import java.util.ArrayList;
import org.apache.hadoop.io.BinaryComparable;
import com.vividsolutions.jts.geom.Coordinate;
import java.io.DataInput;
import java.io.DataOutput;
import com.vividsolutions.jts.geom.Geometry;
import com.vividsolutions.jts.geom.GeometryFactory;
import java.util.HashMap;
import java.io.IOException;
import java.util.List;
import java.util.Map;
import java.util.Iterator;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import com.vividsolutions.jts.geom.Geometry;
import com.vividsolutions.jts.io.WKTReader;

public class GIS extends BinaryComparable implements
   Writable
{
```

```
public Geometry geometry;
public HashMap<String, String> attributes;
private List<String> columns;

public GIS() {
        geometry = new GeometryFactory().
           createPoint(new Coordinate(0,0));
        attributes = new HashMap<String, String
           >(32);
        columns = (List<String>) new ArrayList<
           String>();
}

public Text toText()
{
        return new Text(hashToString(attributes) +
           ”\n”);
}

// For BinaryComparable
public byte[] getBytes()
{
        return toText().getBytes();
}

public int getLength()
{
        return toText().getLength();
}

public boolean update(Text value, List<String>
   columnList)
{
        columns.clear();
        attributes.clear();
        String[] splits = value.toString().split
           (”\”,\””);

        columns.addAll(columnList);
        for (int i=0; i < splits.length; i++)
        {
                // Erase begining and ending quotes
                   /commas
                splits[i] = splits[i].replaceAll
                   (”ˆ\””,””);
                splits[i] = splits[i].replaceAll
                   (”\”$”,””);

                attributes.put(columns.get(i),
                   splits[i]);

                if (columnList.size () == 0)
                        columns.add (String.valueOf
                            (i));
        }

        try {
                geometry = new WKTReader().read(
                   new String ((String) attributes.
                   get(”the_geom”)));
        }
        catch (com.vividsolutions.jts.io.
           ParseException e) { }

        return true;
}

public String toString()
{
        StringBuilder finalString = new
           StringBuilder ();
        for (int i=0; i<columns.size (); i++)
        {
                finalString.append (”\””);
                finalString.append (attributes.get
                   (columns.get (i)));
                finalString.append (”\””);
                if (i != columns.size ()−1)
                        finalString.append(”,”);
        }

        //finalString.append (”\n”);
        return finalString.toString ();
```

```
}

public void write(DataOutput out) throws
   IOException {
        Text value = toText();
        value.write(out);
}

public void readFields(DataInput in) throws
   IOException {
        Text value = new Text();
        value.readFields(in);
        attributes.putAll (stringToHash (value.
           toString()));
        try {
                geometry = new WKTReader().read(
                   new String ((String) attributes.
                   get(”the_geom”)));
        }
        catch (com.vividsolutions.jts.io.
           ParseException e) { }
}

// Outputs ”key1”=”value1”,”key2”=”value2” after
   escaping \”,= characters
private String hashToString(HashMap<String,String>
   h)
{
        h.put (”the_geom”, geometry.toText ());
        String key, value, finalString = ””;

        for (int i=0; i<columns.size (); i++)
        {
                // Escape string
                key = columns.get (i);
                value = attributes.get(key);

                key = key.replace(”ˆ\””,””);
                value = value.replace(”\”$”,””);

                key = key.replace(”\\”, ”\\\\”);
                key = key.replace(”\””, ”\\\””);
        key = key.replace(”=”, ”\\=”);
        key = key.replace(”,”, ”\\,”);

        value = value.replace(”\\”, ”\\\\”)
           ;
        value = value.replace(”\””, ”\\\””)
           ;
        value = value.replace(”=”, ”\\=”);
        value = value.replace(”,”, ”\\,”);

        finalString += ”\”” + key + ”\”=\””
            + value + ”\””;
        if(i != columns.size ()−1)
            finalString += ”,”;
    }

    return finalString;
}

private HashMap<String,String> stringToHash(String
   str)
{
    if(str == null)
        return null;

    columns.clear ();
    str = str.substring (0, str.length ()−1);
    HashMap<String,String> h = new HashMap<
       String,String>();
    String [] splits = str.split(”\”,\””);
    for(int i=0; i<splits.length; i++)
    {
        String [] pair = splits [i].split
           (”\”=\””);
        if(pair.length != 2)
        {
            h.put (pair [0], ””);
            columns.add (pair [0]);
            continue;
        }

        // Unescape string
```

```
            pair[0] = pair[0].replace(”\\\\”,
              ”\\”);
            pair[0] = pair[0].replace(”\\\””,
              ”\””);
            pair[0] = pair[0].replace(”\\=”,
              ”=”);
            pair[0] = pair[0].replace(”\\,”,
              ”,”);

            pair[1] = pair[1].replace(”\\\\”,
              ”\\”);
            pair[1] = pair[1].replace(”\\\””,
              ”\””);
            pair[1] = pair[1].replace(”\\=”,
              ”=”);
            pair[1] = pair[1].replace(”\\,”,
              ”,”);

            pair[1] = pair[1].replace(”\””,””);
            pair[0] = pair[0].replace(”\””,””);

            h.put(pair [0], pair [1]);
            columns.add (pair [0]);
        }

        return h;
    }
}
```

A.1.2 GISInputFormat.java

```
package hadoopGIS;

import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileSplit;
import hadoopGIS.GISRecordReader;
import org.apache.hadoop.mapred.InputSplit;
import java.io.IOException;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reporter;
```

```
public class GISInputFormat extends FileInputFormat<
   LongWritable, GIS> {

        // From FileInputFormat
        public RecordReader<LongWritable, GIS>
           getRecordReader(InputSplit split, JobConf job,
           Reporter reporter) throws IOException
        {
                return new GISRecordReader(job, (FileSplit)
                    split);
        }

}
```

A.1.3 GISRecordReader.java

```
package hadoopGIS;

import java.util.List;
import java.util.ArrayList;
import java.io.BufferedReader;
import java.io.FileReader;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.mapred.FileSplit;
import hadoopGIS.GIS;
import java.io.IOException;
import org.apache.hadoop.mapred.LineRecordReader;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.io.Text;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.filecache.DistributedCache;

class GISRecordReader implements RecordReader<LongWritable,
    GIS>
{
        private List<String> columnList;
        private LineRecordReader reader;
```

```
public GISRecordReader(Configuration job, FileSplit
    split) throws IOException
{
        columnList = new ArrayList<String> ();
        reader = new LineRecordReader(job, (
           FileSplit) split);

        String columnFilename = job.get (”
           columnNames”);
        Path[] distCacheFiles = new Path[0];
        try { distCacheFiles = DistributedCache.
           getLocalCacheFiles(job); }
        catch (IOException e)   { return; }

        String line = new String ();
        for (int i=0; i<distCacheFiles.length; i++)
        {
                if (distCacheFiles[i].getName ().
                   equals (columnFilename))
                {
                        BufferedReader reader = new
                            BufferedReader(new
                           FileReader(
                           distCacheFiles[i].
                           toString ()));
                        while ((line = reader.
                           readLine()) != null)
                                columnList.add (
                                   line);

                        break;
                }
        }
}

public float getProgress()
{
        return reader.getProgress();
}

public synchronized void close() throws IOException
{
```

```
                reader.close();
        }

        public  synchronized long getPos() throws
           IOException
        {
                return reader.getPos();
        }

        public GIS createValue()
        {
                return new GIS();
        }

        public LongWritable createKey() {
                return new LongWritable();
        }

        public synchronized boolean next(LongWritable key,
           GIS value) throws IOException
        {
                Text textValue = new Text();
                if (reader.next(key, textValue)) {
                        value.update(textValue, columnList)
                           ;
                        key.set(new Long(value.attributes.
                           get(”id”)));
                        return true;
                }
                return false;
        }
}
```

A.1.4 GISOutputFormat.java

```
package hadoopGIS;

import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.FSDataOutputStream;
import hadoopGIS.GIS;
import java.io.IOException;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.util.Progressable;
import org.apache.hadoop.mapred.RecordWriter;

public class GISOutputFormat extends FileOutputFormat<
   LongWritable, GIS>
{

        public RecordWriter<LongWritable, GIS>
           getRecordWriter(FileSystem ignored, JobConf job,
            String name, Progressable progress) throws
           IOException
        {
                Path file = getTaskOutputPath(job, name);
                FileSystem fs = file.getFileSystem(job);
                FSDataOutputStream fileOut = fs.create(file
                   , progress);
                return new GISRecordWriter<LongWritable,
                   GIS>(fileOut);
        }

}
```

A.1.5 GISRecordWriter.java

```
package hadoopGIS;

import java.io.DataOutputStream;
import hadoopGIS.GIS;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapred.RecordWriter;
import org.apache.hadoop.mapred.Reporter;
import java.io.UnsupportedEncodingException;

public class GISRecordWriter<LongWritable, GIS> implements
   RecordWriter<LongWritable, GIS>
{
        DataOutputStream out;
        public GISRecordWriter(DataOutputStream out) {
                this.out = out;
        }

        public synchronized void write(LongWritable key,
           GIS value) throws IOException {
                out.writeBytes(value.toString());
                out.writeBytes(”\n”);
        }

        public synchronized void close(Reporter reporter)
           throws IOException {
                out.close();
        }
}
```

A.2 Operations

These listings are the complete code for the required operations. Each operation is contained in one source file.

A.2.1 Create

```
package hadoopGIS.examples;

import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.TextOutputFormat;
import org.apache.hadoop.io.Text;

import hadoopGIS.GIS;
import hadoopGIS.GISInputFormat;
import hadoopGIS.GISOutputFormat;

import java.io.IOException;
import java.io.FileNotFoundException;
import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Iterator;
import java.util.HashMap;
import java.util.Map;
import java.util.ArrayList;

import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.filecache.DistributedCache;

import com.vividsolutions.jts.geom.Geometry;
import com.vividsolutions.jts.io.WKTReader;
import com.vividsolutions.jts.io.ParseException;

public class create extends Configured implements Tool,
   Mapper<LongWritable, GIS, LongWritable, GIS>, Reducer<
   LongWritable, GIS, LongWritable, GIS>
{
        // For Mapper interface
        public void map(LongWritable key, GIS value,
           OutputCollector<LongWritable, GIS> output,
           Reporter reporter) throws IOException
        {
                output.collect(key, value);
        }

        // For Reducer interface
        public void reduce(LongWritable key, Iterator<GIS>
           values, OutputCollector<LongWritable, GIS>
           output, Reporter reporter)
        {
                while(values.hasNext()) {
                        try {
                                output.collect(key, values.
                                   next());
                        } catch (IOException e) {}
                }
        }

        // For Mapper (via JobConfigurable) interface
        public void configure(JobConf job)
        {
        }

        // For Mapper (via Closeable) interface
        public void close() {}

        // For Tool interface
        public int run(String[] args) throws Exception
        {
                JobConf job = new JobConf(new Configuration
                   (), this.getClass());

                GISInputFormat.setInputPaths(job, new Path
                   (”/user/alaster/gis/parcels.gis”));
                Path p = new Path (”/user/alaster/gis/
                   parcels.names”);
                DistributedCache.addCacheFile (p.toUri (),
                   job);
                job.set (”columnNames”, p.getName ());

                GISOutputFormat.setOutputPath(job, new Path
                   (”output”));

                job.setJobName(”hadoopGIS.examples.create”)
                   ;

                job.setMapperClass(this.getClass());
                //job.setCombinerClass(this.getClass());
                job.setReducerClass(this.getClass());

                job.setInputFormat(GISInputFormat.class);
                //job.setOutputFormat(TextOutputFormat.
                   class);
                job.setOutputValueClass(GIS.class);
                job.setOutputFormat(GISOutputFormat.class);
```

```
                return JobClient.runJob(job).getJobState();
        }

        // Hadoop runner requires this to be a static void!
        // Thus must use exit instead of return
        // Also must directly use the class name instead of
             figuring it out
        public static void main(String[] args) throws
           Exception {
                System.exit(ToolRunner.run(new
                   Configuration(), new hadoopGIS.examples.
                   create(), args));
        }
}
```

A.2.2 Read

```
package hadoopGIS.examples;

import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;

import hadoopGIS.GIS;
import hadoopGIS.GISInputFormat;
import hadoopGIS.GISOutputFormat;

import java.io.IOException;
import java.io.FileNotFoundException;
import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Iterator;
import java.util.HashMap;
import java.util.Map;
import java.util.ArrayList;

import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.filecache.DistributedCache;

import com.vividsolutions.jts.geom.Geometry;
import com.vividsolutions.jts.io.WKTReader;
import com.vividsolutions.jts.io.ParseException;

public class read extends Configured implements Tool,
   Mapper<LongWritable, GIS, LongWritable, GIS>, Reducer<
   LongWritable, GIS, LongWritable, GIS>
{
        // For Mapper interface
        public void map(LongWritable key, GIS value,
           OutputCollector<LongWritable, GIS> output,
           Reporter reporter) throws IOException
        {
                // emit only the record with the correct
                   key
                if(key.equals(new LongWritable(1008130))) {
                        output.collect(key, value);
                }
        }

        // For Reducer interface
        public void reduce(LongWritable key, Iterator<GIS>
           values, OutputCollector<LongWritable, GIS>
           output, Reporter reporter)
        {
                while(values.hasNext()) {
                        try {
                                output.collect(key, values.
                                   next());
                        } catch (IOException e) {}
                }
        }

        // For Mapper (via JobConfigurable) interface
```

```
public void configure(JobConf job)
{
}

// For Mapper (via Closeable) interface
public void close() {}

// For Tool interface
public int run(String[] args) throws Exception
{
        JobConf job = new JobConf(new Configuration
           (), this.getClass());

        GISInputFormat.setInputPaths(job, new Path
           (”/user/alaster/gis/parcels.gis”));
        Path p = new Path (”/user/alaster/gis/
           parcels.names”);
        DistributedCache.addCacheFile (p.toUri (),
           job);
        job.set (”columnNames”, p.getName ());

        GISOutputFormat.setOutputPath(job, new Path
           (”output”));

        job.setJobName(”hadoopGIS.examples.read”);

        job.setMapperClass(this.getClass());
        //job.setCombinerClass(this.getClass());
        job.setReducerClass(this.getClass());

        job.setInputFormat(GISInputFormat.class);
        //job.setOutputFormat(TextOutputFormat.
           class);
        job.setOutputValueClass(GIS.class);
        job.setOutputFormat(GISOutputFormat.class);

        return JobClient.runJob(job).getJobState();

}

// Hadoop runner requires this to be a static void!
// Thus must use exit instead of return
```

```
        // Also must directly use the class name instead of
            figuring it out
        public static void main(String[] args) throws
           Exception {
                System.exit(ToolRunner.run(new
                   Configuration(), new hadoopGIS.examples.
                   read(), args));
        }
}
```

A.2.3 Update

```
package hadoopGIS.examples;

import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;

import hadoopGIS.GIS;
import hadoopGIS.GISInputFormat;
import hadoopGIS.GISOutputFormat;

import java.io.IOException;
import java.io.FileNotFoundException;
import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Iterator;
import java.util.HashMap;
import java.util.Map;
import java.util.ArrayList;

import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.filecache.DistributedCache;

import com.vividsolutions.jts.geom.Geometry;
import com.vividsolutions.jts.io.WKTReader;
import com.vividsolutions.jts.io.ParseException;

public class update extends Configured implements Tool,
   Mapper<LongWritable, GIS, LongWritable, GIS>, Reducer<
   LongWritable, GIS, LongWritable, GIS>
{
        // For Mapper interface
        public void map(LongWritable key, GIS value,
           OutputCollector<LongWritable, GIS> output,
           Reporter reporter) throws IOException
        {
                // Change record 1008130 to a Commercial
                   parcel
                if(key.equals(new LongWritable(1008130))) {
                        value.attributes.put(”devtype”, ”C
                           ”);
                }
                output.collect(key, value);
        }

        // For Reducer interface
        public void reduce(LongWritable key, Iterator<GIS>
           values, OutputCollector<LongWritable, GIS>
           output, Reporter reporter)
        {
                while(values.hasNext()) {
                        try {
                                output.collect(key, values.
                                   next());
                        } catch (IOException e) {}
                }
        }

        // For Mapper (via JobConfigurable) interface
        public void configure(JobConf job)
        {
        }
```

```
// For Mapper (via Closeable) interface
public void close() {}

// For Tool interface
public int run(String[] args) throws Exception
{
        JobConf job = new JobConf(new Configuration
           (), this.getClass());

        GISInputFormat.setInputPaths(job, new Path
           (”/user/alaster/gis/parcels.gis”));
        Path p = new Path (”/user/alaster/gis/
           parcels.names”);
        DistributedCache.addCacheFile (p.toUri (),
           job);
        job.set (”columnNames”, p.getName ());

        GISOutputFormat.setOutputPath(job, new Path
           (”output”));

        job.setJobName(”hadoopGIS.examples.update”)
           ;

        job.setMapperClass(this.getClass());
        //job.setCombinerClass(this.getClass());
        job.setReducerClass(this.getClass());

        job.setInputFormat(GISInputFormat.class);
        //job.setOutputFormat(TextOutputFormat.
           class);
        job.setOutputValueClass(GIS.class);
        job.setOutputFormat(GISOutputFormat.class);

        return JobClient.runJob(job).getJobState();
}

// Hadoop runner requires this to be a static void!
// Thus must use exit instead of return
// Also must directly use the class name instead of
    figuring it out
```

```
        public static void main(String[] args) throws
           Exception {
                System.exit(ToolRunner.run(new
                   Configuration(), new hadoopGIS.examples.
                   update(), args));
        }
}
```

A.2.4 Delete

```
package hadoopGIS.examples;

import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;

import hadoopGIS.GIS;
import hadoopGIS.GISInputFormat;
import hadoopGIS.GISOutputFormat;

import java.io.IOException;
import java.io.FileNotFoundException;
import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Iterator;
import java.util.HashMap;
import java.util.Map;
import java.util.ArrayList;

import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.filecache.DistributedCache;
import com.vividsolutions.jts.geom.Geometry;
import com.vividsolutions.jts.io.WKTReader;
import com.vividsolutions.jts.io.ParseException;

public class delete extends Configured implements Tool,
   Mapper<LongWritable, GIS, LongWritable, GIS>, Reducer<
   LongWritable, GIS, LongWritable, GIS>
{
        // For Mapper interface
        public void map(LongWritable key, GIS value,
           OutputCollector<LongWritable, GIS> output,
           Reporter reporter) throws IOException
        {
                // emit only the record with the correct
                   key
                if(!key.equals(new LongWritable(1008130)))
                   {
                        output.collect(key, value);
                }
        }

        // For Reducer interface
        public void reduce(LongWritable key, Iterator<GIS>
           values, OutputCollector<LongWritable, GIS>
           output, Reporter reporter)
        {
                while(values.hasNext()) {
                        try {
                                output.collect(key, values.
                                   next());
                        } catch (IOException e) {}
                }
        }

        // For Mapper (via JobConfigurable) interface
        public void configure(JobConf job)
        {
        }

        // For Mapper (via Closeable) interface
        public void close() {}
```

```
// For Tool interface
public int run(String[] args) throws Exception
{
        JobConf job = new JobConf(new Configuration
           (), this.getClass());

        GISInputFormat.setInputPaths(job, new Path
           (”/user/alaster/gis/parcels.gis”));
        Path p = new Path (”/user/alaster/gis/
           parcels.names”);
        DistributedCache.addCacheFile (p.toUri (),
           job);
        job.set (”columnNames”, p.getName ());

        GISOutputFormat.setOutputPath(job, new Path
           (”output”));

        job.setJobName(”hadoopGIS.examples.delete”)
           ;

        job.setMapperClass(this.getClass());
        //job.setCombinerClass(this.getClass());
        job.setReducerClass(this.getClass());

        job.setInputFormat(GISInputFormat.class);
        //job.setOutputFormat(TextOutputFormat.
           class);
        job.setOutputValueClass(GIS.class);
        job.setOutputFormat(GISOutputFormat.class);

        return JobClient.runJob(job).getJobState();
}

// Hadoop runner requires this to be a static void!
// Thus must use exit instead of return
// Also must directly use the class name instead of
    figuring it out
public static void main(String[] args) throws
   Exception {
        System.exit(ToolRunner.run(new
           Configuration(), new hadoopGIS.examples.
```

```
                        delete(), args));
        }
}
```

A.2.5 Filter

```
package hadoopGIS.examples;

import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;

import hadoopGIS.GIS;
import hadoopGIS.GISInputFormat;
import hadoopGIS.GISOutputFormat;

import java.io.IOException;
import java.io.FileNotFoundException;
import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Iterator;
import java.util.HashMap;
import java.util.Map;
import java.util.ArrayList;

import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.filecache.DistributedCache;

import com.vividsolutions.jts.geom.Geometry;
import com.vividsolutions.jts.io.WKTReader;
import com.vividsolutions.jts.io.ParseException;
```

```
public class filter extends Configured implements Tool,
   Mapper<LongWritable, GIS, LongWritable, GIS>, Reducer<
   LongWritable, GIS, LongWritable, GIS>
{
        private Geometry box;

        // For Mapper interface
        public void map(LongWritable key, GIS value,
           OutputCollector<LongWritable, GIS> output,
           Reporter reporter) throws IOException
        {
                // Keep records that intersect with the box
                if(value.geometry.intersects(box)) {
                        output.collect(key, value);
                }
        }

        // For Reducer interface
        public void reduce(LongWritable key, Iterator<GIS>
           values, OutputCollector<LongWritable, GIS>
           output, Reporter reporter)
        {
                while(values.hasNext()) {
                        try {
                                output.collect(key, values.
                                   next());
                        } catch (IOException e) {}
                }
        }

        // For Mapper (via JobConfigurable) interface
        public void configure(JobConf job)
        {
                ArrayList<String> columns = new ArrayList<
                   String>();

                // Create the column list that is used
                columns.add(”id”);
                columns.add(”the_geom”);
                columns.add(”devtype”);
                // Create the box
                try {
                        box = new WKTReader().read(”POLYGON
                           ((−112.0859375
                           33.4349975585938,−112.0859375
                           33.4675445556641,−112.059799194336

                           33.4675445556641,−112.059799194336
                            33.4349975585938,−112.0859375
                           33.4349975585938))”);
                } catch(com.vividsolutions.jts.io.
                   ParseException e) {}
        }

        // For Mapper (via Closeable) interface
        public void close() {}

        // For Tool interface
        public int run(String[] args) throws Exception
        {
                JobConf job = new JobConf(new Configuration
                   (), this.getClass());

                GISInputFormat.setInputPaths(job, new Path
                   (”/user/alaster/gis/parcels.gis”));
                Path p = new Path (”/user/alaster/gis/
                   parcels.names”);
                DistributedCache.addCacheFile (p.toUri (),
                   job);
                job.set (”columnNames”, p.getName ());

                GISOutputFormat.setOutputPath(job, new Path
                   (”output”));

                job.setJobName(”hadoopGIS.examples.filter”)
                   ;

                job.setMapperClass(this.getClass());
                //job.setCombinerClass(this.getClass());
                job.setReducerClass(this.getClass());

                job.setInputFormat(GISInputFormat.class);
```

```
                //job.setOutputFormat(TextOutputFormat.
                    class);
                job.setOutputValueClass(GIS.class);
                job.setOutputFormat(GISOutputFormat.class);

                return JobClient.runJob(job).getJobState();
        }

        // Hadoop runner requires this to be a static void!
        // Thus must use exit instead of return
        // Also must directly use the class name instead of
             figuring it out
        public static void main(String[] args) throws
            Exception {
                System.exit(ToolRunner.run(new
                    Configuration(), new hadoopGIS.examples.
                    filter(), args));
        }
}
```

A.2.6 Nearest

```
package hadoopGIS.examples;

import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;

import hadoopGIS.GIS;
import hadoopGIS.GISInputFormat;
import hadoopGIS.GISOutputFormat;

import java.io.IOException;
import java.io.FileNotFoundException;
import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Iterator;
import java.util.HashMap;
import java.util.Map;
import java.util.ArrayList;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.filecache.DistributedCache;

import com.vividsolutions.jts.geom.Geometry;
import com.vividsolutions.jts.io.WKTReader;
import com.vividsolutions.jts.io.ParseException;

public class chained extends Configured implements Tool,
   Mapper<LongWritable, GIS, LongWritable, LongWritable>,
   Reducer<LongWritable, LongWritable, LongWritable,
   LongWritable>
{
        ArrayList<GIS> parcels;

        // For Mapper interface
        public void map(LongWritable key, GIS value,
           OutputCollector<LongWritable, LongWritable>
           output, Reporter reporter) throws IOException
        {
                double minDistance = Double.MAX_VALUE,
                   currDistance;
                int closestParcel = -1;

                Iterator it = parcels.iterator();
                while (it.hasNext())
                {
                        GIS parcel = (GIS) it.next();

                        currDistance = value.geometry.
                           distance(parcel.geometry);
                        if (currDistance < minDistance)
                        {
```

```
                        minDistance = currDistance;
                        closestParcel = new Integer
                           (parcel.attributes.get(”
                           id”));
                }
        }

        LongWritable lngClosestParcel = new
           LongWritable (closestParcel);
        output.collect(key, lngClosestParcel);
}

// For Reducer interface
public void reduce(LongWritable key, Iterator<
   LongWritable> values, OutputCollector<
   LongWritable, LongWritable> output, Reporter
   reporter)
{
        while(values.hasNext()) {
                try {
                        output.collect(key, values.
                           next());
                } catch (IOException e) {}
        }
}

// For Mapper (via JobConfigurable) interface
public void configure(JobConf job)
{
        String columnFilename = job.get (”
           parcelColumnNames”);
        String dataFilename = job.get (”parcelData
           ”);
        Path[] distCacheFiles = new Path[0];
        try { distCacheFiles = DistributedCache.
           getLocalCacheFiles(job); }
        catch (IOException e)   { return; }

        ArrayList<String> parcelColumnList = new
           ArrayList<String >();
        parcels = new ArrayList<GIS>();
```

```
BufferedReader reader = null;
String line;
for (int i=0; i<distCacheFiles.length; i++)
{
        if (distCacheFiles [i].getName ().
           equals (columnFilename))
        {
                try
                {
                        reader = new
                           BufferedReader(
                           new FileReader(
                           distCacheFiles [
                           i].toString ()))
                           ;
                        while ((line =
                           reader.readLine
                           ()) != null)
                                parcelColumnList
                                   .add (
                                   line);
                }
                catch (Exception e) { }
                break;
        }
}

GIS myGIS;
for (int i=0; i<distCacheFiles.length; i++)
{
        if (distCacheFiles [i].getName ().
           equals (dataFilename))
        {
                try
                {
                        reader = new
                           BufferedReader(
                           new FileReader(
                           distCacheFiles [
                           i].toString ()))
                           ;
```

```
                                        while ((line =
                                           reader.readLine
                                           ()) != null)
                                        {
                                                myGIS = new
                                                    GIS();
                                                myGIS.
                                                   update (
                                                   new Text
                                                    (line),

                                                   parcelColumnList
                                                   );
                                                parcels.add
                                                    (myGIS)
                                                   ;
                                        }
                                }
                                catch (Exception e) { }

                                break;
                        }
                }
        }

        // For Mapper (via Closeable) interface
        public void close() {}

        // For Tool interface
        public int run(String[] args) throws Exception
        {
                JobConf job = new JobConf(new Configuration
                   (), this.getClass());

                GISInputFormat.setInputPaths(job, new Path
                   ("/user/alaster/gis/jobs.gis"));
                GISOutputFormat.setOutputPath(job, new Path
                   ("output"));

                job.setJobName("hadoopGIS.examples.nearest
                   ");
```

```
          job.setMapperClass(this.getClass());
          //job.setCombinerClass(this.getClass());
          job.setReducerClass(this.getClass());

          job.setInputFormat(GISInputFormat.class);
          //job.setOutputFormat(TextOutputFormat.
            class);
          job.setOutputValueClass(LongWritable.class)
            ;

          Path p = new Path (”/user/alaster/gis/jobs.
            names”);
          DistributedCache.addCacheFile (p.toUri (),
            job);
          job.set (”columnNames”, p.getName ());

          p = new Path (”/user/alaster/gis/parcels.
            names”);
          DistributedCache.addCacheFile (p.toUri (),
            job);
          job.set (”parcelColumnNames”, p.getName ())
            ;

          p = new Path (”/user/alaster/gis/parcels.
            gis”);
          DistributedCache.addCacheFile (p.toUri (),
            job);
          job.set (”parcelData”, p.getName ());

          return JobClient.runJob(job).getJobState();
    }

    // Hadoop runner requires this to be a static void!
    // Thus must use exit instead of return
    // Also must directly use the class name instead of
       figuring it out
    public static void main(String[] args) throws
      Exception {
          System.exit(ToolRunner.run(new
            Configuration(), new hadoopGIS.examples.
            nearest(), args));
    }
```

```
}
```

A.2.7 Chained

```
package hadoopGIS.examples;

import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;

import hadoopGIS.GIS;
import hadoopGIS.GISInputFormat;
import hadoopGIS.GISOutputFormat;

import java.io.IOException;
import java.io.FileNotFoundException;
import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Iterator;
import java.util.HashMap;
import java.util.Map;
import java.util.ArrayList;

import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.filecache.DistributedCache;

import com.vividsolutions.jts.geom.Geometry;
import com.vividsolutions.jts.io.WKTReader;
import com.vividsolutions.jts.io.ParseException;
```

```
public class chained extends Configured implements Tool,
   Mapper<LongWritable, GIS, LongWritable, LongWritable>,
   Reducer<LongWritable, LongWritable, LongWritable,
   LongWritable>
{
        ArrayList<GIS> parcels;

        // For Mapper interface
        public void map(LongWritable key, GIS value,
           OutputCollector<LongWritable, LongWritable>
           output, Reporter reporter) throws IOException
        {
                double minDistance = Double.MAX_VALUE,
                   currDistance;
                int closestParcel = -1;

                Iterator it = parcels.iterator();
                while (it.hasNext())
                {
                        GIS parcel = (GIS) it.next();

                        currDistance = value.geometry.
                           distance(parcel.geometry);
                        if (currDistance < minDistance)
                        {
                                minDistance = currDistance;
                                closestParcel = new Integer
                                   (parcel.attributes.get(”
                                   id”));
                        }
                }

                LongWritable lngClosestParcel = new
                   LongWritable (closestParcel);
                output.collect(key, lngClosestParcel);
        }

        // For Reducer interface
        public void reduce(LongWritable key, Iterator<
           LongWritable> values, OutputCollector<
           LongWritable, LongWritable> output, Reporter
           reporter)
```

```
{
        while(values.hasNext()) {
                try {
                        output.collect(key, values.
                           next());
                } catch (IOException e) {}
        }
}

// For Mapper (via JobConfigurable) interface
public void configure(JobConf job)
{
        String columnFilename = job.get (”
           parcelColumnNames”);
        String dataFilename = job.get (”parcelData
           ”);
        Path[] distCacheFiles = new Path[0];
        try { distCacheFiles = DistributedCache.
           getLocalCacheFiles(job); }
        catch (IOException e)   { return; }

        ArrayList<String> parcelColumnList = new
           ArrayList<String >();
        parcels = new ArrayList<GIS>();

        BufferedReader reader = null;
        String line;
        for (int i=0; i<distCacheFiles.length; i++)
        {
                if (distCacheFiles [i].getName ().
                   equals (columnFilename))
                {
                        try
                        {
                                reader = new
                                   BufferedReader(
                                   new FileReader(
                                   distCacheFiles [
                                   i].toString ()))
                                   ;
                                while ((line =
                                   reader.readLine
                           ()) != null)
                                parcelColumnList
                                   .add (
                                   line);
                }
                catch (Exception e) { }
                break;
        }
}

GIS myGIS;
for (int i=0; i<distCacheFiles.length; i++)
{
        if (distCacheFiles [i].getName ().
           equals (dataFilename))
        {
                try
                {
                        reader = new
                           BufferedReader(
                           new FileReader(
                           distCacheFiles [
                           i].toString ()))
                           ;
                        while ((line =
                           reader.readLine
                           ()) != null)
                        {
                                myGIS = new
                                    GIS();
                                myGIS.
                                   update (
                                   new Text
                                    (line),

                                   parcelColumnList
                                   );
                                if (!myGIS.
                                   attributes
                                   .get ("
                                   devtype
                                   ").
```

```
                      equals(”
                      R”))
                    {
                        parcels
                          .
                          add

                          (
                          myGIS
                          )
                          ;

                    }
                }
            }
            catch (Exception e) { }

            break;
        }
    }
}

// For Mapper (via Closeable) interface
public void close() {}

// For Tool interface
public int run(String[] args) throws Exception
{
    JobConf job = new JobConf(new Configuration
       (), this.getClass());

    GISInputFormat.setInputPaths(job, new Path
       (”/user/alaster/gis/jobs.gis”));
    GISOutputFormat.setOutputPath(job, new Path
       (”output”));

    job.setJobName(”hadoopGIS.examples.chained
       ”);

    job.setMapperClass(this.getClass());
    //job.setCombinerClass(this.getClass());
    job.setReducerClass(this.getClass());
```

```
        job.setInputFormat(GISInputFormat.class);
        //job.setOutputFormat(TextOutputFormat.
          class);
        job.setOutputValueClass(LongWritable.class)
          ;

        Path p = new Path (”/user/alaster/gis/jobs.
          names”);
        DistributedCache.addCacheFile (p.toUri (),
          job);
        job.set (”columnNames”, p.getName ());

        p = new Path (”/user/alaster/gis/parcels.
          names”);
        DistributedCache.addCacheFile (p.toUri (),
          job);
        job.set (”parcelColumnNames”, p.getName ())
          ;

        p = new Path (”/user/alaster/gis/parcels.
          gis”);
        DistributedCache.addCacheFile (p.toUri (),
          job);
        job.set (”parcelData”, p.getName ());

        return JobClient.runJob(job).getJobState();
    }

    // Hadoop runner requires this to be a static void!
    // Thus must use exit instead of return
    // Also must directly use the class name instead of
       figuring it out
    public static void main(String[] args) throws
      Exception {
        System.exit(ToolRunner.run(new
          Configuration(), new hadoopGIS.examples.
          chained(), args));
    }
}
```

CHAPTER B CLUSTERGIS SOURCE

The complete source for clusterGIS is included in this appendix. ClusterGIS is also available at `http://github.com/nathankerr/clusterGIS`.

The clusterGIS listings are comprised of two sections: the library and the example implementations.

B.1 Library

The core clusterGIS library is contained in clustergis.c. To use clusterGIS, include clusterGIS.h in your processing implementation and link clusterGIS with your code.

B.1.1 clustergis.h

```
#ifndef CLUSTERGIS_H
#define CLUSTERGIS_H

#define CLUSTERGIS_BUFFERSIZE 2*1024*1024

#include ”stdio.h”
#include ”stdlib.h”
#include ”mpi.h”
#include ”geos_c.h”

/* variables */
int clusterGIS_started;

/* datatypes */
struct clusterGIS_record_el {
        char** data;
        int columns;
        GEOSGeometry* geometry;
        struct clusterGIS_record_el * next;
};
typedef struct clusterGIS_record_el clusterGIS_record;
struct clusterGIS_dataset {
        clusterGIS_record* data;
};
typedef struct clusterGIS_dataset clusterGIS_dataset;

/* startup and shutdown */
```

```
void clusterGIS_Init(int* argc, char*** argv);
void clusterGIS_Finalize(void);

/* dataset operations */
clusterGIS_dataset* clusterGIS_Create_dataset(void);
clusterGIS_dataset* clusterGIS_Load_csv_distributed(
   MPI_Comm comm, char* filename);
clusterGIS_dataset* clusterGIS_Load_csv_replicated(MPI_Comm
    comm, char* filename);
void clusterGIS_Write_csv(char* filename,
   clusterGIS_dataset* dataset);
void clusterGIS_Write_csv_distributed(MPI_Comm comm, char*
   filename, clusterGIS_dataset* dataset);
void clusterGIS_Free_dataset(clusterGIS_dataset* dataset);

/* record operations */
clusterGIS_record* clusterGIS_Create_record_from_csv(char*
   csv, int* size);
void clusterGIS_Free_record(clusterGIS_record* record);

/* MPI operations */
MPI_Comm clusterGIS_Create_chunked_communicator(MPI_Comm
   comm, int size);
MPI_Comm clusterGIS_Create_strided_communicator(MPI_Comm
   comm, int stride);

/* Geometry operations */
void clusterGIS_Create_wkt_geometries(clusterGIS_dataset*
   dataset, int geometry_column);
void clusterGIS_Create_wkt_geometry(clusterGIS_record*
   record, int geometry_column);

#endif
```

B.1.2 clustergis.c

```
#include ”clustergis.h”
#include ”string.h”
#include ”sys/stat.h”
#include ”assert.h”

/* clusterGIS_Init
```

```
 *
 * Sets up the clusterGIS environment
 *
 * argc - count of arguments in argv
 * argv - char** of arguments
 */
void clusterGIS_Init(int* argc, char*** argv) {
        MPI_Init(argc, argv);
        initGEOS(NULL, NULL);

        clusterGIS_started = 1;
}

/* clusterGIS_Finalize
 *
 * Closes out the clusterGIS environment
 */
void clusterGIS_Finalize(void) {
        MPI_Finalize();
        finishGEOS();
}

/* dataset operations */

/* clusterGIS_Create_dataset
 *
 * Creates a new clusterGIS_dataset
 */
clusterGIS_dataset* clusterGIS_Create_dataset(void) {
        clusterGIS_dataset* dataset = malloc(sizeof(
           clusterGIS_dataset));
        dataset->data = NULL;

        return dataset;
}

/* clusterGIS_Load_csv_distributed
 *
 * Loads a portion of a dataset on each task
 *
 * comm - MPI communicator to use
 * filename - path to the dataset
 * dataset - the dataset which will be created
 */
clusterGIS_dataset* clusterGIS_Load_csv_distributed(
   MPI_Comm comm, char* filename) {
        MPI_File file;
        int err;
        char* buffer;
        MPI_Status status;
        clusterGIS_record** record;
        MPI_Offset offset;
        MPI_Offset chunkstart;
        MPI_Offset chunkend;
        int count;
        MPI_Offset filesize;
        int i;
        int start;
        int end;
        clusterGIS_dataset* dataset;
        int comm_rank;
        int comm_size;

        MPI_Comm_rank(comm, &comm_rank);
        MPI_Comm_size(comm, &comm_size);

        err = MPI_File_open(MPI_COMM_WORLD, filename,
           MPI_MODE_RDONLY, MPI_INFO_NULL, &file);
        assert(err == MPI_SUCCESS);

        buffer = (char*) malloc(CLUSTERGIS_BUFFERSIZE);
        MPI_File_get_size(file, &filesize);
        offset = 0;
        dataset = (clusterGIS_dataset*) malloc(sizeof(
           clusterGIS_dataset));
        record = &dataset->data;

        /* determine chunksizes, last task picks up the
           slack */
        chunkstart = comm_rank * (filesize / comm_size);
        if (comm_rank == comm_size - 1) {
                chunkend = filesize;
        } else {
```

```
        chunkend = chunkstart + (filesize /
           comm_size) − 1;
}

offset = chunkstart;
while(offset < chunkend − 1) {
        MPI_File_read_at(file, offset, buffer,
           CLUSTERGIS_BUFFERSIZE, MPI_CHAR, &status
           );
        MPI_Get_count(&status, MPI_CHAR, &count);

        /* determine the start of this record set
           */
        start = 0;
        if (offset == chunkstart && comm_rank != 0)
            {
                while(buffer[start] != ’\n’ &&
                   start < count) {
                        start++;
                }
                if(start == count) {
                        fprintf(stderr, ”%d: buffer
                            is too small\n”,
                           comm_rank);
                        MPI_Abort(MPI_COMM_WORLD,
                           1);
                }
                start++;
        }

        /* determine the end of this record set */
        end = count − 1;
        while(end > 0 && buffer[end] != ’\n’) {
                end−−;
        }
        if(end == 0) {
                fprintf(stderr, ”%d: \\n not found
                   from end\n”, comm_rank);
                MPI_Abort(MPI_COMM_WORLD, 1);
        }
        end++;
        if(chunkend - offset < count) {
                /* Buffer overruns the
                   responsibility of this task */
                end = chunkend - offset + 1;
                while(end < count && buffer[end] !=
                     '\n') {
                        end++;
                }
                end++;
                if (end > count - 1) {
                        end = chunkend - offset +
                           1;
                        while(end > 0 && buffer[end
                           ] != '\n') {
                                end--;
                        }
                        if(end == 0) {
                                fprintf(stderr, "%d
                                   : \\n not found
                                   from end\n",
                                   comm_rank);
                                MPI_Abort(
                                   MPI_COMM_WORLD,
                                   1);
                        }
                        end++;
                }
        }

        /* Put the records into the dataset */
        i = start;
        while (i < end) {
                (*record) =
                   clusterGIS_Create_record_from_csv
                   (buffer, &i);
                (*record)->next = NULL;
                record = &(*record)->next;
                i++;
        }

        offset += end;
}
```

```
        free(buffer);
        MPI_File_close(&file);

        return dataset;
}

/* clusterGIS_Load_csv_replicated
 *
 * Loads an entire copy of a csv data source on each task
    included in comm
 *
 * comm - MPI communicator of which all members will get a
    copy of this dataset
 * filename - path to the csv formatted dataset
 *
 * returns a pointer to the dataset
 */
clusterGIS_dataset* clusterGIS_Load_csv_replicated(MPI_Comm
    comm, char* filename) {
        MPI_File file;
        int err;
        char* buffer;
        int buffersize = 2*1024*1024;
        MPI_Status status;
        clusterGIS_record** record;
        MPI_Offset offset;
        int count;
        int last_full_record_end;
        MPI_Offset filesize;
        int i;
        clusterGIS_dataset* dataset;
        int comm_rank;

        MPI_Comm_rank(comm, &comm_rank);

        err = MPI_File_open(comm, filename, MPI_MODE_RDONLY
            , MPI_INFO_NULL, &file);
        if(err != MPI_SUCCESS) {
                fprintf(stderr, ”%d: Error opening file %s\
                    n”, comm_rank, filename);
                MPI_Abort(comm, err);
}

buffer = (char*) malloc(buffersize);
MPI_File_get_size(file, &filesize);
offset = 0;
dataset = (clusterGIS_dataset*) malloc(sizeof(
   clusterGIS_dataset));
record = &dataset->data;

while(offset < filesize) {
        MPI_File_read_at_all(file, offset, buffer,
           buffersize, MPI_CHAR, &status);
        MPI_Get_count(&status, MPI_CHAR, &count);

        /* find where the last full record ends */
        last_full_record_end = count - 1;
        while(buffer[last_full_record_end] != ’\n’
           && last_full_record_end >= 0) {
                last_full_record_end--;
        }
        if(last_full_record_end < 0) {
                fprintf(stderr, ”%d: Error in
                   clusterGIS_Load_data_replicated:
                    buffersize is too small, %d\n”,
                    comm_rank, count);
                MPI_Abort(comm, 1);
        }

        /* Put the records into the dataset */
        i = 0;
        while (i < last_full_record_end) {
                (*record) =
                   clusterGIS_Create_record_from_csv
                   (buffer, &i);
                (*record)->next = NULL;
                record = &(*record)->next;
                i++;
        }

        offset = offset + last_full_record_end + 1;
}
```

```
        free(buffer);
        MPI_File_close(&file);

        return dataset;
}

/* clusterGIS_Write_csv
 *
 * Writes a dataset out to a file as csv
 *
 * filename - file to write to
 * dataset - dataset to write
 */
void clusterGIS_Write_csv(char* filename,
    clusterGIS_dataset* dataset) {
        FILE* file;
        clusterGIS_record* record;
        int i;

        remove(filename);
        file = fopen(filename, ”w”);

        record = dataset->data;
        while(record != NULL) {
                fprintf(file, ”\”%s\””, record->data[0]);
                for(i = 1; i < record->columns; i++) {
                        fprintf(file, ”,\”%s\””, record->
                            data[i]);
                }
                fprintf(file, ”\n”);
                record = record->next;
        }

        fclose(file);
}

/* clusterGIS_Write_csv_distributed
 *
 * Writes a distributed dataset to disk using MPI-IO
 *
 * comm - MPI communicator of the participants of the
     distributed dataset
 * filename - path of the file to write to
 * dataset - dataset to write
 */
void clusterGIS_Write_csv_distributed(MPI_Comm comm, char*
   filename, clusterGIS_dataset* dataset) {
        char* filename_part;
        MPI_Offset filesize;
        MPI_Offset offset;
        int i;
        int tmp;
        FILE* file_part;
        char* buffer;
        int buffersize=1024*1024;
        MPI_File file;
        struct stat *file_part_stat;
        int size;
        MPI_Status status;
        int comm_rank;
        int comm_size;

        MPI_Comm_rank(comm, &comm_rank);
        MPI_Comm_size(comm, &comm_size);

        /* Write local part of dataset to individual files
           */
        filename_part = (char*) malloc(strlen(filename) +
           5);
        sprintf(filename_part, ”%s.%d”, filename, comm_rank
           );
        clusterGIS_Write_csv(filename_part, dataset);

        /* Figure out offset by talking with other tasks */
        file_part_stat = malloc(sizeof(struct stat));
        stat(filename_part, file_part_stat);
        filesize = (int) file_part_stat->st_size;
        free(file_part_stat);

        offset = 0;
        for(i = 0; i < comm_size; i++) {
                tmp = filesize;
                MPI_Bcast(&tmp, 1, MPI_INT, i, comm);
                if (i < comm_rank) {
                        offset += tmp;
                }
        }

        /* Write local parts back together into a single
           large file */
        file_part = fopen(filename_part, ”r”);
        remove(filename);
        MPI_File_open(comm, filename, MPI_MODE_WRONLY |
           MPI_MODE_CREATE, MPI_INFO_NULL, &file);
        buffer = (char*) malloc(buffersize);

        size = fread(buffer, 1, buffersize, file_part);
        while(size != 0) {
                MPI_File_write_at(file, offset, buffer,
                   size, MPI_CHAR, &status);
                offset += size;
                size = fread(buffer, 1, buffersize,
                   file_part);
        }
        free(buffer);
        fclose(file_part);
        remove(filename_part);
        MPI_File_close(&file);

        free(filename_part);
}

/* clusterGIS_Free_dataset
 *
 * Frees all memory associated with a dataset (all
   associated records, etc)
 *
 * dataset - the dataset to be freed
 */
void clusterGIS_Free_dataset(clusterGIS_dataset* dataset) {
        clusterGIS_record* head;
        clusterGIS_record* current;

        head = dataset->data;
        current = head;
        while(current != NULL) {
                head = current->next;
                clusterGIS_Free_record(current);
                current = head;
        }

        free(dataset);
}

/* clusterGIS_Create_record_from_csv
 *
 * Creates a record from the given csv formatted char*
 *
 * csv - csv formatted representation of record
 * start - index of the start of the record in csv,
   returned with the end index
 *
 * Returns generated record
 */
clusterGIS_record* clusterGIS_Create_record_from_csv(char*
   csv, int* start) {
        int end = *start;
        int field_end;
        int field_count;
        int field_start;
        int i;
        int j;
        clusterGIS_record* record;

        struct item {
                char* data;
                int len;
                struct item* next;
        };
        struct item* head;
        struct item* current;

        /* find end of record */
        while(csv[end] != '\n') {
                end++;
        }
```

```
/* Pull the fields out of the record. Assumes
   fields are comma delimited. Quotes surround
   fields with commas or quotes (escaped in the
   field) */
i = *start;
field_count = 0;
head = (struct item*) malloc(sizeof(struct item));
current = head;
current->next = NULL;
while(i < end) {
        /* get the start and stop indexes of the
           field */
        if(csv[i] == '"') {
                /* escaped field */
                i++;
                field_start = i;
                while (csv[i] != '"' && csv[i-1] !=
                    '\\') {
                        i++;
                }
                field_end = i;
                i++; /* moves to the , or \n that
                   follows the " */
        } else {
                /* non escaped field */
                field_start = i;
                while(csv[i] != ',' && csv[i] != '\
                   n') {
                        i++;
                }
                field_end = i;
        }

        /* add field to the linked list and move to
            the next field */
        current->data = (char*) malloc(field_end -
           field_start + 1);
        for(j = 0; j < field_end - field_start; j
           ++) {
                current->data[j] = csv[field_start
                   + j];
        }
```

```
                current->data[j] = '\0';

                current->next = (struct item*) malloc(
                   sizeof(struct item));
                current = current->next;
                current->next = NULL;
                field_count++;
                i++;
        }

        /* Convert the linked list to an array of strings
           */
        current = head;
        i = 0;
        record = (clusterGIS_record*) malloc(sizeof(
           clusterGIS_record));
        record->data = malloc(field_count * sizeof(char*));
        record->columns = field_count;
        record->next = NULL;
        record->geometry = NULL;
        for(i = 0; i < field_count; i++) {
                record->data[i] = current->data;
                head = current->next;
                free(current);
                current = head;
        }

        (*start) = end;
        return record;
}

/* clusterGIS_Free_record
 *
 * Frees all memory associated with a record
 *
 * record - the record to free
 */
void clusterGIS_Free_record(clusterGIS_record* record) {
        if(record != NULL) {
                free(record->data);
                free(record);
        }
```

```
}

/* MPI operations */
/* clusterGIS_Create_sub_communicator
 *
 * Creates a new MPI_Comm of size contiguous tasks
 *
 * comm - The communicator which to create a subset from
 * size - The size of the sub comminicator
 * new_comm - The address of the new communicator
 */
MPI_Comm clusterGIS_Create_chunked_communicator(MPI_Comm
   comm, int size) {
        int* members;
        int start;
        int end;
        int rank;
        int tasks;
        MPI_Group old_group;
        MPI_Group new_group;
        int i;
        MPI_Comm new_comm;

        MPI_Comm_rank(comm, &rank);
        MPI_Comm_size(comm, &tasks);
        MPI_Comm_group(comm, &old_group);

        /* Determine start and end of this group */
        start = rank - (rank % size);
        end = start + size - 1;
        if(end >= tasks) end = tasks - 1;
        size = end - start + 1;

        /* List the members in this group */
        members = malloc(sizeof(int) * size);
        for(i = 0; i < size; i++) {
                members[i] = start + i;
        }

        /* Create the new group, and then from it the new
           communicator */
        MPI_Group_incl(old_group, size, members, &new_group
           );
        MPI_Comm_create(comm, new_group, &new_comm);

        free(members);
        return new_comm;
}

MPI_Comm clusterGIS_Create_strided_communicator(MPI_Comm
   comm, int stride) {
        int* members;
        int comm_rank;
        int comm_size;
        MPI_Group old_group;
        MPI_Group new_group;
        int i;
        MPI_Comm new_comm;
        int rank;
        int position;

        MPI_Comm_rank(comm, &comm_rank);
        MPI_Comm_size(comm, &comm_size);
        MPI_Comm_group(comm, &old_group);

        /* Determine which stride group we are in */
        position = comm_rank % stride;

        /* List the members in this group */
        i = 0;
        members = malloc(sizeof(int) * comm_size);
        for(rank = 0; rank < comm_size; rank++) {
                if(rank % stride == position) {
                        members[i] = rank;
                        i++;
                }
        }

        /* Create the new group, and then from it the new
           communicator */
        MPI_Group_incl(old_group, i, members, &new_group);
        MPI_Comm_create(comm, new_group, &new_comm);
        free(members);
        return new_comm;
}

/* clusterGIS_Create_wkt_geometries
 *
 * Creates geometries in the dataset from the WKT formatted
     data in geometry_column
 *
 * dataset - dataset to be modified
 * geometry_column - column of the dataset the WKT
    formatted geometry is located in
 */
void clusterGIS_Create_wkt_geometries(clusterGIS_dataset*
   dataset, int geometry_column) {
        clusterGIS_record* record;

        record = dataset->data;
        while(record != NULL) {
                clusterGIS_Create_wkt_geometry(record,
                   geometry_column);
                record = record->next;
        }
}

/* clusterGIS_Create_wkt_geometry
 *
 * Creates a geometry in the record from the WKT formatted
    datas in geometry_column
 *
 * record - the record to be modified
 * geometry_column - the column in record->data containing
    the WKT formatted geometry data
 */
void clusterGIS_Create_wkt_geometry(clusterGIS_record*
   record, int geometry_column) {
        GEOSWKTReader* reader = GEOSWKTReader_create();
        record->geometry = GEOSWKTReader_read(reader,
           record->data[geometry_column]);
        GEOSWKTReader_destroy(reader);
}
```

B.2 Operations

These listings implement the operations used for this thesis.

B.2.1 Create

```
/* File: create.c
 * Author: Nathan Kerr
 *
 * Copies a dataset, adding a new record in the process.
 */

#include ”clustergis.h”

int main(int argc, char** argv) {
        clusterGIS_dataset* dataset;
        clusterGIS_record* record;
        int rank;

        /* Process local arguments */
        if (argc != 3) {
                fprintf(stderr, ”Usage: %s input output\n”,
                    argv[0]);
                exit(1);
        }

        clusterGIS_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        dataset = clusterGIS_Load_csv_distributed(
           MPI_COMM_WORLD, argv[1]);

        /* adds a new record to the beginning of the
           dataset */
        if(rank == 0) {
                int start = 0;
                record = clusterGIS_Create_record_from_csv
                   (”97123897,POINT(0 0),C\n”, &start);
                record->next = dataset->data;
                dataset->data = record;
        }

        /* writes the records to disk */
        clusterGIS_Write_csv_distributed(MPI_COMM_WORLD,
           argv[2], dataset);

        /* Finalize */
        clusterGIS_Finalize();
        return 0;
}
```

B.2.2 Read

```
/* File: read.c
 * Author: Nathan Kerr
 *
 * Outputs the specified record (by id)
 */

#include ”clustergis.h”
#include ”string.h”

int main(int argc, char** argv) {
        clusterGIS_dataset* dataset;
        clusterGIS_record* record;
        clusterGIS_record** head;

        /* Process local arguments */
        if (argc != 3) {
                fprintf(stderr, ”Usage: %s input output\n”,
                   argv[0]);
                exit(1);
        }

        /* Init */
        clusterGIS_Init(&argc, &argv);
        dataset = clusterGIS_Load_csv_distributed(
           MPI_COMM_WORLD, argv[1]);

        record = dataset->data;
        head = &(dataset->data);
        /* keep records that match the criteria, otherwise
           delete them */
        while(record != NULL) {
                if(atoi(record->data[0]) == 1008130) {
                        head = &(record->next);
                        record = record->next;
                } else {
                        *head = record->next;
                        clusterGIS_Free_record(record);
                        record = *head;
                }
        }

        clusterGIS_Write_csv_distributed(MPI_COMM_WORLD,
           argv[2], dataset);

        /* Finalize */
        clusterGIS_Finalize();
        return 0;
}
```

B.2.3 Update

```
/* File: read.c
 * Author: Nathan Kerr
 *
 * Changes record 1008130 from R to C
 */

#include ”clustergis.h”
#include ”string.h”

int main(int argc, char** argv) {
        clusterGIS_dataset* dataset;
        clusterGIS_record* record;

        /* Process local arguments */
        if (argc != 3) {
                fprintf(stderr, ”Usage: %s input output\n”,
                   argv[0]);
                exit(1);
        }

        /* Init */
        clusterGIS_Init(&argc, &argv);
        dataset =  clusterGIS_Load_csv_distributed(
            MPI_COMM_WORLD, argv[1]);

        record = dataset->data;
        /* keep records that match the criteria, otherwise
            delete them */
        while(record != NULL) {
                if(atoi(record->data[0]) == 1008130) {
                        record->data[2] = ”C”;
                }
                record = record->next;
        }

        clusterGIS_Write_csv_distributed(MPI_COMM_WORLD,
            argv[2], dataset);

        /* Finalize */
        clusterGIS_Finalize();
        return 0;
}
```

B.2.4 Delete

```
/* File: read.c
 * Author: Nathan Kerr
 *
 * Deletes the specified record (by id)
 */

#include ”clustergis.h”
#include ”string.h”

int main(int argc, char** argv) {
        clusterGIS_dataset* dataset;
        clusterGIS_record* record;
        clusterGIS_record** head;

        /* Process local arguments */
        if (argc != 3) {
                fprintf(stderr, ”Usage: %s input output\n”,
                    argv[0]);
                exit(1);
        }

        /* Init */
        clusterGIS_Init(&argc, &argv);
        dataset = clusterGIS_Load_csv_distributed(
           MPI_COMM_WORLD, argv[1]);

        record = dataset->data;
        head = &(dataset->data);
        /* keep records that match the criteria, otherwise
           delete them */
        while(record != NULL) {
                if(atoi(record->data[0]) == 1008130) {
                        *head = record->next;
                        clusterGIS_Free_record(record);
                        record = *head;
                } else {
                        head = &(record->next);
                        record = record->next;
                }
        }

        clusterGIS_Write_csv_distributed(MPI_COMM_WORLD,
           argv[2], dataset);

        /* Finalize */
        clusterGIS_Finalize();
        return 0;
}
```

B.2.5 Filter

```
#include ”clustergis.h”
#include ”geos_c.h”

int main(int argc, char** argv) {
        GEOSGeometry* box;
        GEOSWKTReader* reader;
        clusterGIS_dataset* dataset;
        clusterGIS_record* record;
        clusterGIS_record** head;
        GEOSGeometry* record_geometry;
```

```
int rank;
double startprocessing;

if(argc != 3) {
        fprintf(stderr, ”Usage %s input output”,
           argv[0]);
        exit(1);
}

clusterGIS_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
reader = GEOSWKTReader_create();

box = GEOSWKTReader_read(reader, ”POLYGON
   ((-112.0859375 33.4349975585938,-112.0859375
   33.4675445556641,-112.059799194336
   33.4675445556641,-112.059799194336
   33.4349975585938,-112.0859375 33.4349975585938))
   ”);
dataset = clusterGIS_Load_csv_distributed(
   MPI_COMM_WORLD, argv[1]);
clusterGIS_Create_wkt_geometries(dataset, 1);

startprocessing = MPI_Wtime();
record = dataset->data;
head = &(dataset->data);
/* keep records that match the criteria, otherwise
   delete them */
while(record != NULL) {
        char intersects;

        /*record_geometry = GEOSWKTReader_read(
           reader, record->data[1]);
        if(record_geometry == NULL) {
                fprintf(stderr, ”%d: parse error\n
                   ”, rank);
                MPI_Abort(MPI_COMM_WORLD, 2);
        }*/

        intersects = GEOSIntersects(record->
           geometry, box);
        if(intersects == 2) {
```

```
                        fprintf(stderr, ”%d: error with
                           overlap function\n”, rank);
                        MPI_Abort(MPI_COMM_WORLD, 2);
                }

                if(intersects == 1) { /* record overlaps
                   with box */
                        head = &(record->next);
                        record = record->next;
                } else if(intersects == 0) { /* no overlap
                   */
                        *head = record->next;
                        clusterGIS_Free_record(record);
                        record = *head;
                } else {
                        /* should never get here */
                        MPI_Abort(MPI_COMM_WORLD, 2);
                }
        }
        printf(”%d: processing time %5.2fs\n”, rank,
           MPI_Wtime() - startprocessing);

        clusterGIS_Write_csv_distributed(MPI_COMM_WORLD,
           argv[2], dataset);

        clusterGIS_Finalize();
        return 0;
}
```

B.2.6 Nearest

```
#include ”clustergis.h”
#include ”string.h”
#include ”nearest.h”

#define BLOCK_SIZE 8
#define EMPLOYERS_GEOMETRY_COLUMN 1
#define PARCELS_GEOMETRY_COLUMN 1

/* reduce function for min distances */
void min_distance_function (double *invec, double* outvec,
   int *len, MPI_Datatype *datatype) {
        if(len == NULL || datatype == NULL) {
                MPI_Abort(MPI_COMM_WORLD, 2);
        }

        /* outvec[i] = invec[i] op outvec[i] */
        if(invec[1] < outvec[1]) {
                outvec[0] = invec[0];
                outvec[1] = invec[1];
        }
}

int main(int argc, char** argv) {
        char* employers_filename;
        char* parcels_filename;
        MPI_Comm employers_comm;
        MPI_Comm parcels_comm;
        clusterGIS_dataset* employers;
        clusterGIS_dataset* parcels;
        clusterGIS_record* employer;
        clusterGIS_record* parcel;
        double distance;
        double min_distance;
        clusterGIS_record* min_distance_parcel;
        double *min;
        double *global_min;
        int world_rank;
        MPI_Op min_distance_op;
        char* output_csv;
        clusterGIS_dataset* output = NULL;
        clusterGIS_record* output_record = NULL;
        int start = 0;
        char* output_filename;

        clusterGIS_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
        MPI_Op_create((MPI_User_function*)
           min_distance_function, 1, &min_distance_op);

        if(argc != 4 && world_rank == 0) {
                printf(”Usage: %s employers parcels output\
                   n”, argv[0]);
                MPI_Abort(MPI_COMM_WORLD, 1);
```

```
}
employers_filename = argv[1];
parcels_filename = argv[2];
output_filename = argv[3];


/* Load data into appropriate communicators and
   create their geometries */
employers_comm =
   clusterGIS_Create_strided_communicator(
   MPI_COMM_WORLD, BLOCK_SIZE);
employers = clusterGIS_Load_csv_distributed(
   employers_comm, employers_filename);
clusterGIS_Create_wkt_geometries(employers,
   EMPLOYERS_GEOMETRY_COLUMN);
parcels_comm =
   clusterGIS_Create_chunked_communicator(
   MPI_COMM_WORLD, BLOCK_SIZE);
parcels = clusterGIS_Load_csv_distributed(
   parcels_comm, parcels_filename);
clusterGIS_Create_wkt_geometries(parcels,
   PARCELS_GEOMETRY_COLUMN);

/* Find the min distance */
employer = employers->data;
min = malloc(sizeof(double)*2);
global_min = malloc(sizeof(double)*2);
output_csv = malloc(sizeof(char)*128);
output = clusterGIS_Create_dataset();
while(employer != NULL) {

        /* find the local min */
        parcel = parcels->data;
        GEOSDistance(employer->geometry, parcel->
           geometry, &min_distance);
        min_distance_parcel = parcel;
        while(parcel != NULL) {
                if(strncmp(employer->data[2],
                   parcel->data[2], 1) == 0) {
                        GEOSDistance(employer->
                           geometry, parcel->
                           geometry, &distance);
```

```
                                if(distance < min_distance)
                                    {
                                        min_distance =
                                           distance;
                                        min_distance_parcel
                                           = parcel;
                                }
                        }
                        parcel = parcel->next;
                }

                /* find the global min */
                min[0] = atoi(min_distance_parcel->data[0])
                   ;
                min[1] = min_distance;
                MPI_Allreduce(min, global_min, 2,
                   MPI_DOUBLE, min_distance_op,
                   parcels_comm);

                /* Add the min to the output dataset */
                sprintf(output_csv, ”\”%s\”,\”%d\”\n”,
                   employer->data[0], (int) global_min[0]);
                start = 0;
                output_record =
                   clusterGIS_Create_record_from_csv(
                   output_csv, &start);
                output_record->next = output->data;
                output->data = output_record;

                employer = employer->next;
        }

        if(world_rank % BLOCK_SIZE == 0) {
                clusterGIS_Write_csv_distributed(
                   employers_comm, output_filename, output)
                   ;
        }

        MPI_Op_free(&min_distance_op);
        clusterGIS_Finalize();
        return 0;
}
```

B.2.7 Chained

```
#include ”clustergis.h”
#include ”string.h”
#include ”chained.h”

#define BLOCK_SIZE 8
#define EMPLOYERS_GEOMETRY_COLUMN 1
#define PARCELS_GEOMETRY_COLUMN 1

/* reduce function for min distances */
void min_distance_function (double *invec, double* outvec,
   int *len, MPI_Datatype *datatype) {
        if(len == NULL || datatype == NULL) {
                MPI_Abort(MPI_COMM_WORLD, 2);
        }
        /* outvec[i] = invec[i] op outvec[i] */
        if(invec[1] < outvec[1]) {
                outvec[0] = invec[0];
                outvec[1] = invec[1];
        }
}

int main(int argc, char** argv) {
        char* employers_filename;
        char* parcels_filename;
        MPI_Comm employers_comm;
        MPI_Comm parcels_comm;
        clusterGIS_dataset* employers;
        clusterGIS_dataset* parcels;
        clusterGIS_record* employer;
        clusterGIS_record* parcel;
        double distance;
        double min_distance;
        clusterGIS_record* min_distance_parcel;
        double *min;
        double *global_min;
        int world_rank;
        MPI_Op min_distance_op;
        char* output_csv;
        clusterGIS_dataset* output = NULL;
        clusterGIS_record* output_record = NULL;
```

```
int start = 0;
char* output_filename;
clusterGIS_record** head;

clusterGIS_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Op_create((MPI_User_function*)
   min_distance_function, 1, &min_distance_op);

if(argc != 4 && world_rank == 0) {
        printf(”Usage: %s employers parcels output\
           n”, argv[0]);
        MPI_Abort(MPI_COMM_WORLD, 1);
}
employers_filename = argv[1];
parcels_filename = argv[2];
output_filename = argv[3];


/* Load data into appropriate communicators and
   create their geometries */
employers_comm =
   clusterGIS_Create_strided_communicator(
   MPI_COMM_WORLD, BLOCK_SIZE);
employers = clusterGIS_Load_csv_distributed(
   employers_comm, employers_filename);
clusterGIS_Create_wkt_geometries(employers,
   EMPLOYERS_GEOMETRY_COLUMN);
parcels_comm =
   clusterGIS_Create_chunked_communicator(
   MPI_COMM_WORLD, BLOCK_SIZE);
parcels = clusterGIS_Load_csv_distributed(
   parcels_comm, parcels_filename);
clusterGIS_Create_wkt_geometries(parcels,
   PARCELS_GEOMETRY_COLUMN);

/* remove all residential parcels */
parcel = parcels->data;
head = &(parcels->data);
while(parcel != NULL) {
        if(strncmp(parcel->data[2], ”R”, 1) == 0) {
                head = &(parcel->next);
                parcel = parcel->next;
        } else {
                *head = parcel->next;
                clusterGIS_Free_record(parcel);
                parcel = *head;
        }
}

/* Find the min distance */
employer = employers->data;
min = malloc(sizeof(double)*2);
global_min = malloc(sizeof(double)*2);
output_csv = malloc(sizeof(char)*128);
output = clusterGIS_Create_dataset();
while(employer != NULL) {

        /* find the local min */
        parcel = parcels->data;
        GEOSDistance(employer->geometry, parcel->
           geometry, &min_distance);
        min_distance_parcel = parcel;
        while(parcel != NULL) {
                if(strncmp(employer->data[2],
                   parcel->data[2], 1) == 0) {
                        GEOSDistance(employer->
                           geometry, parcel->
                           geometry, &distance);
                        if(distance < min_distance)
                            {
                                min_distance =
                                   distance;
                                min_distance_parcel
                                   = parcel;
                        }
                }
                parcel = parcel->next;
        }

        /* find the global min */
        min[0] = atoi(min_distance_parcel->data[0])
           ;
        min[1] = min_distance;
```

```
                MPI_Allreduce(min, global_min, 2,
                    MPI_DOUBLE, min_distance_op,
                    parcels_comm);

                /* Add the min to the output dataset using
                    front insertion */
                sprintf(output_csv, ”\”%s\”,\”%d\”\n”,
                    employer->data[0], (int) global_min[0]);
                start = 0;
                output_record =
                    clusterGIS_Create_record_from_csv(
                    output_csv, &start);
                output_record->next = output->data;
                output->data = output_record;

                employer = employer->next;
        }

        /* Write one copy of the result dataset out */
        if(world_rank % BLOCK_SIZE == 0) {
                clusterGIS_Write_csv_distributed(
                    employers_comm, output_filename, output)
                    ;
        }

        MPI_Op_free(&min_distance_op);
        clusterGIS_Finalize();
        return 0;
}
```