MRGIS: A MapReduce-Enabled High Performance Workflow System for GIS

Qichang Chen, Liqiang Wang
Department of Computer Science
University of Wyoming
{qchen2, wang}@cs.uwyo.edu

Zongbo Shang
WyGISC and Department of Geography
University of Wyoming
zshang1@uwyo.edu

Abstract

The growth of data used by data-intensive computations, e.g. Geographical Information Systems (GIS), has far outpaced the growth of the power of a single processor. The increasing demand of data-intensive applications calls for distributed computing. In this paper, we propose a high performance workflow system MRGIS, a parallel and distributed computing platform based on MapReduce clusters, to execute GIS applications efficiently. MRGIS consists of a design interface, a task scheduler, and a runtime support system. The design interface has two options: a GUI-based workflow designer and an API-based library for programming in Python. Given a GIS workflow, the scheduler analyzes data dependencies among tasks, then dispatches them to MapReduce clusters based on the current status of the system. Our experiment demonstrates that MRGIS can significantly improve the performance of GIS workflow execution.

1 Introduction

As sensor and storage technologies continue to improve, they make it possible to collect and store previously inconceivable amounts of data. Scientific research increasingly relies on computing over massive datasets. However, data collection rates exceed our ability to process it, as CPU frequency has been staggering. This challenge calls for massive parallelism for scientific computing. Cluster and grid are traditional parallel computing platforms. The innovations of multi-core architectures make parallel programming more pervasive. The emerging cloud computing is delivering even larger-scale parallel and distributed data processing. Although for the past decade we have witnessed incrementally more programmers writing parallel programs, the vast majority of applications today are still single-threaded because of the difficulty of designing parallel programs.

To alleviate the complexity of parallel programming, many programming models have been proposed. One of the most successful frameworks is MapReduce [5], which processes massive datasets in parallel manner with supporting load balancing and fault tolerance, etc. MapReduce programs can be automatically parallelized and executed on a large cluster of computers. The MapReduce runtime system takes care of data partitioning, task scheduling, failure handling, and communication managing. This allows programmers with no parallel programming experience to easily utilize cluster for data-intensive computing.

In this paper, we propose a MapReduce-enabled high performance workflow system for applications of Geographical Information System (GIS). The system is called *MRGIS*, which stands for **M**ap**R**educe-enabled **GIS**. As GIS data grow significantly and computations become much more complex, current GIS products (e.g. ESRI ArcGIS [1] and GRASS [2]) are very inefficient in executing such computing jobs, largely because GIS systems are designed for executing sequentially on a single workstation. Based on MapReduce cluster, MRGIS provides a massive parallel computing platform to execute and manage data-intensive GIS applications more efficiently and effectively.

Although a few workflow management systems have been developed over the past several years, they either do not support parallel computing, or are inefficient and ineffective for GIS applications. First, GIS applications often operate on large-scale datasets. In support of better performance, the datasets should be dynamically partitioned for parallel executions. However, how to partition and how many chunks to split are difficult problems, which depend on many factors, such as the current GIS operation, data type, the workflow execution status, and even the current parallel system status. The workflow scheduling and data partitioning are closely related and should be designed specifically for GIS workflows. Second, GIS workflows involve many specific GIS operations not supported by general workflow systems. MRGIS provides a specialized GUI-based user interface in support of these GIS operations. Finally, MRGIS supports data reliability and fault tolerance using MapReduce architecture, which most exist-

978-0-7695-3535-7/08 $25.00 © 2008 IEEE
DOI 10.1109/eScience.2008.169

ing workflow systems often do not support or just have a limited support.

We evaluated MRGIS on two real-world GIS computation workflows. Our experiment demonstrates that MRGIS can significantly speed up the execution time of GIS workflows, as compared to the current popular GRASS [2] on a single machine.

This paper is organized as follows. Section 2 presents the details of the design and implementation of MRGIS. The experiments in Section 3 demonstrate the efficiency of MRGIS. Section 4 reviews the related work. In Section 5, we summarize the contributions of our work and describe the future work.

2 The Design and Implementation of MRGIS

Geographical Information System (GIS) is an information system for capturing, storing, analyzing, managing, and presenting spatial data. GIS has been increasingly used in resource management, scientific research, and even our daily life. As current GIS tools(*e.g.* ESRI ArcGIS [1]) are mainly designed for executing sequentially on a single workstation, a GIS becomes much less efficient when dealing with tremendous data and complex computations.

We designed and implemented a distributed and parallel GIS computing platform, MRGIS, based on an open source implementation of MapReduce - Hadoop [3] and an open source GIS tool - GRASS [2]. Figure 1 shows the architecture of MRGIS. MRGIS consists of a design interface, a scheduler, and a runtime support system. There are two options for the user design interface: a GUI-based GIS workflow designer and a workflow scripting language implemented as a Python library, since Python is a dominating scripting language used for GIS applications. The interface enables programmers to easily design complex workflows using specialized toolbox for GIS operations without knowing details of the underlying parallel computing platform. Given a GIS workflow, the scheduler analyzes data dependencies among tasks, then dispatches tasks to MapReduce cluster based on the current status of the system to minimize execution time. Built over MapReduce architecture, the runtime system supports and manages GIS operations on each computer node.

2.1 MapReduce and Hadoop

MapReduce [5] is a programming model for data-intensive computing, which consists of two functions: a *map* function applies a specific operation to a set of items, and produces a set of intermediate key/value pairs on a group of distributed computer nodes, and a *reduce* function merges all intermediate values associated with the same key on distributed computer nodes. The runtime system takes

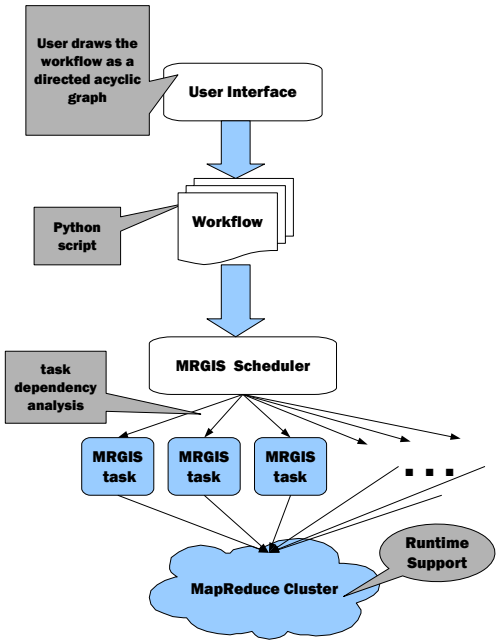

Figure 1. The architecture of MRGIS.

care of data partitioning, scheduling, load balancing, fault tolerance, and network communications. The simple interface of MapReduce allows programmers to easily design parallel and distributed applications.

Hadoop [3] is an open-source implementation of MapReduce in Java. It is composed of MapReduce runtime system and a distributed file system (HDFS), which provides data redundancy support and makes the data diffusion transparent among each node in the cluster. Both MapReduce and the distributed file system are designed to automatically handle node failures. One of Hadoop nodes works as a master, which dispatches tasks and controls the executions of the other Hadoop nodes, *i.e.*, slaves.

2.2 A GIS Workflow Example

A typical GIS application needs to process various raster and vector images which are usually in the scale of 100 MiB - 1 GiB (of each file). For example, it takes more than 10 minutes to perform a *plus* GIS operation on two typical GIS raster dataset files in the size of 1 GiB on a typical desktop computer.

To process and analyze massive spatial datasets, the operations of GIS are usually organized as a workflow. A GIS workflow is a directed acyclic graph (DAG), which consists of nodes and edges. A node denotes a task or an input/output/intermediate data file; and an edge denotes a relationship (*i.e.*, input/output) between a task and a data

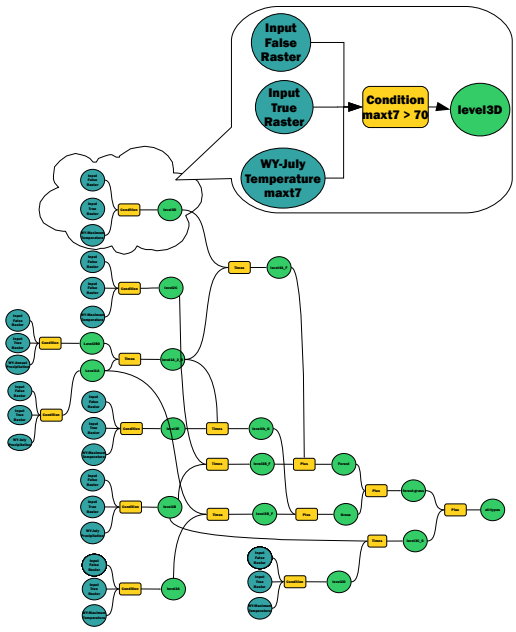

Figure 2. A GIS workflow to classify land into four different vegetation groups, where circles denote datasets and rectangles denote GIS operations. A zoom-in view is shown on the top.

```
import RasterMapAlgebra, MRGIS
input[] = [input1, input2]
output[] = [output]
MRGIS.task(RasterMapAlgebra.times, input, output)
```

Figure 3. A code block showing the definition of a task.

file. Figure 2 shows a real-world GIS workflow on the raster data. With the input of environmental dataset collected by Wyoming Geographic Information Science Center (WyGISC), the workflow is to classify land into four different vegetation groups: forest, grass, shrub, and desert.

Figure 3 shows an example of a task defined using MRGIS python library. Task operations are implemented as wrappers with calling for GIS operations in GRASS [2].

2.3 Workflow Model and Scheduling

Formally, a *workflow* $w = \langle T, D, E_{in}, E_{out} \rangle$ consists of a set T of tasks, a set D of datasets, a set E_{in} of input edges, and a set E_{out} of output edges between tasks and data nodes. Let $t_1 \rightarrow_{out} d \rightarrow_{in} t_2$ denote that task t_1 produces a data set d, and task t_2 consumes it as an input. For such a relationship between t_1 and t_2, we say that t_2 *depends* on t_1, denoted $t_1 \Rightarrow t_2$. Task dependencies are transitive, *i.e.*, if $t_1 \Rightarrow t_2$ and $t_2 \Rightarrow t_3$, then we have $t_1 \Rightarrow t_3$. There is no cyclic transitive dependency on tasks, since all MRGIS workflows are DAGs.

Given a work $w = \langle T, D, E_{in}, E_{out} \rangle$, for a task $t \in T$, let $\textit{depd-ancestors}_w(t) = \{t' | t' \Rightarrow t \wedge t' \in T\}$ (*i.e.*, all tasks that task t depends on) and $\textit{depd-descendents}_w(t) = \{t' | t \Rightarrow t' \wedge t' \in T\}$ (*i.e.*, all tasks that depend on task t).

Let $c(t, D_{t_{in}}, D_{t_{out}})$ denote the *execution cost* (in this paper, only execution time is considered) of a task t, where $D_{t_{in}}$ and $D_{t_{out}}$ denote the input datasets and output datasets of t, respectively. In other words, the execution cost of a task is related to what operation it performs and the sizes of input and output datasets. For a task t, its *finishing cost* is defined as $fc(t) = \sum_{t^i \in \{\textit{depd-descendents}_{w}(t)\}} c(t^i, D_{t^i_{in}}, D_{t^i_{out}})$.

Given a workflow in Python script designed by a GIS tool (*e.g.*ArcGIS) or our GUI-based designer , MRGIS can parse it into a DAG in the format described above. The DAG contains all the necessary information to guide scheduling and execute the tasks. Our parser is implemented using an Eclipse plug-in for Python [4].

MRGIS scheduler exploits two kinds of parallelism. The first parallelism is on the task level based on the fact that many operations in the workflow can be executed in parallel if they do not have data dependency relationships. When the number of ready-to-run tasks is more than the available compute nodes, the scheduler chooses the tasks with higher priority to execute. The priority of a task is computed based on a *simulation*: for each task, we estimate its execution time for a given size of input data based on previous statistical results of executions; the finishing cost of a task is the total execution time of all dependent tasks; thus, a task has a higher priority if it has a higher finishing cost.

The other kind of parallelism is on the data level, *i.e.*, by data partitioning. Most GIS operations work block by block. Thus, a large dataset can be split into multiple independent partitions on which a task can work in parallel. Given a workflow, a difficult problem is to estimate how many chunks to split. Our approach is based on a *simulation*: based on our estimation for the execution time of each task, we simulate the execution of the workflow and compare the execution times for different numbers of chunks. Usually, the overall execution time of a workflow initially drops down along the number of chunks increasing; after reaching a turning point, it will go up even the number of chunks continuously increases because the data communication cost goes up and there are not enough compute nodes to process the chunks. The optimal data partitions will minimize the overall execution time of the workflow.

Even there is not parallelism on workflow level, the data partition approach can still improve performance against the

old sequential computing.

Our scheduling algorithm is shown in Algorithm 1, which is also used for the simulations to find task with the highest priority and compute the optimal data partition.

```
Input: ( w: a GIS workflow)

T := the set of all tasks in the workflow w;
F := ∅; /* the set of finished tasks */

FetchTaskToRun(){
maxFC := 0; /* the max finishing cost */
taskToRun := null; /* the next task selected to run
      which has the max finishing cost in the set
      of non-scheduled tasks */
while F ⊉ T do
    for each t ∈ T − F do
        if depd-ancestors_w(t) ≠ ∅ ∧
        depd-ancestors_w(t) ⊈ F then
            continue;
        end
        if fc(t) > maxFC then
            maxFC := fc(t);
            taskToRun := t;
        end
    end
    if maxFC == 0 then
        return wait;
    end
    else
        return taskToRun;
    end
end
return done;
}

/* Exec() is called by a set of threads on a Hadoop
master node, each of which services a Hadoop slave
node */
Exec() {
while there is an idle Hadoop node do
    switch FetchTaskToRun() do
        case done: return;
        case wait: sleep(some time) ;
        case task t to run:
                submit t to Hadoop;
                wait for finishing;
                F := F ∪ {t};
    end
end
}
```

Algorithm 1: The scheduling algorithm.

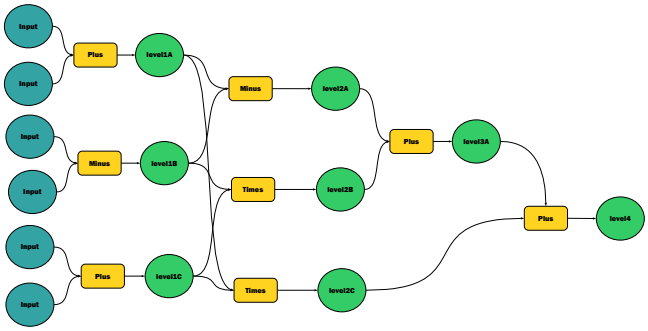

Figure 4. The workflow for benchmark 1.

2.4 Runtime Support System

Our runtime support is responsible to bridge task operations in MRGIS and the corresponding functions in GRASS GIS [2]. We implement our library by wrapping GRASS GIS operations that can be easily called by tasks dispatched on MapReduce platform. Currently our library supports basic GIS raster map operations. The library will be extended to support more GIS operations in our future work.

3 Experiments

We evaluate our system against two typical GIS applications that are representative of the data-intensive nature of GIS applications.

The computing environment is a Hadoop MapReduce cluster consisting of 32 desktop computers running Linux CentOS 5. Each machine has a Intel dual-core 2GHz CPU and 2 GiB RAM. We compare the performance of running the two GIS workflows on a single machine (*i.e.*, any node in the cluster) to the performance of running them on the cluster.

The first benchmark is shown in Figure 4, which is a typical GIS workflow that involves basic GIS algebraic operations. Its input involves only a single data set whose size is around 700 MiB. The data is in Arc/ASCII format[1].

The second benchmark is already shown in Figure 2. The input datasets are obtained from remote sensing on a 30m resolution scale. The details of the input datasets are shown in Table 1. The workflow implements a decision tree classification algorithm consisting of 18 tasks which are all basic operations to GIS.

Figure 5 shows the normalized comparison of the above two benchmarks running on a MapReduce cluster and a single machine.

[1]The Arc/ASCII is a popular GIS data exchange format. We chose it because of its simple structure to split the data.

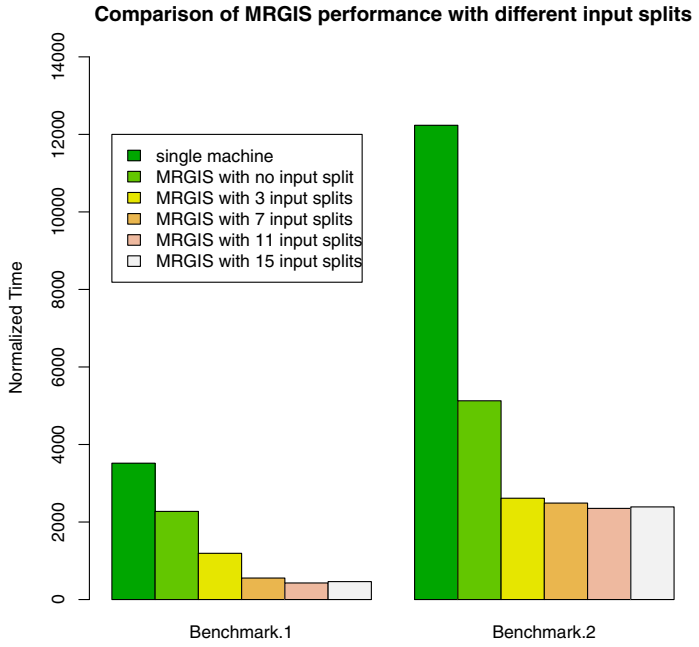

Figure 5. The performance comparison.

Type of data	format	Size
The precipitations in July	Arc/ASCII	2 GiB
The annual precipitations	Arc/ASCII	1.2 GiB
The maximum temperature in a year.	Arc/ASCII	1 GiB

Table 1. Details of the input data for benchmark 2

For benchmark 1, the executions on our MapReduce-enabled platform are much faster than the execution on a single machine. As the number of data partition increases from 3 to 11, the performance continuously improves. However, when the number of partitions reaches 15, the performance drops. The reason is that the number of ready tasks exceeds the number of idle machines in the cluster which causes some ready tasks to wait, and deploying more tasks induces more overhead.

For benchmark 2, the executions on our MapReduce-enabled platform are also much faster than the sequential execution. However, as the number of data partition increases, the performance does not improve significantly as in benchmark 1. It is because the number of ready tasks quickly exceeds the number of available machines, since benchmark 2 has more tasks than benchmark 1.

4 Related Work

There are some extensions for MapReduce. Map-Reduce-Merge [11] adds an additional merge phase that can efficiently merge data already partitioned and sorted by map and reduce operations. However, it is only applicable to a specific type of programs that demand such operations. GridBatch [8] breaks down MapReduce into elementary operators and introduces additional operators, which include map, distribute, recurse, join, cartesian, and neighbor.

A bunch of workflow tools for parallel programming have been developed. Dryad [7] is a general-purpose execution engine for distributed and parallel applications. All jobs and their dataflows are expressed as a directed acyclic graph (DAG). Unlike MapReduce requires a sequence of map/distribute/sort/reduce operations, Dryad supports an arbitrary sequence of operations defined as a DAG. Sawzall [10] is a scripting language over MapReduce to provide an easier interface for data processing. Like MapReduce, a Sawzall program consists of a filtering phase (the map step) and an aggregation phase (the reduce step). Besides these, Sawzall provides an efficient way to format data using protocol buffers and supports scheduling jobs on a cluster of

machines. Built on Hadoop, Pig Latin [9] is a data processing language like Sawzall. While Pig Latin has the similar higher level primitives like filtering and aggregation provided by Sawzall, it supports additional primitives such as cogrouping, which can be used for join operation. The primitives are allowed to be chained in Pig Latin. In addition, Sawzall is a script language and more like procedural languages such as Java, whereas Pig is more like an extension of SQL. Swift [12] is another parallel programming tool that supports defining a workflow in a scripting language and dispatches the workflow onto multiple Grid sites. The Pegasus system [6] provides a framework which maps complex scientific workflows onto distributed grid resources. Artificial intelligence planning techniques are used in Pegaus for workflow composition.

Our MRGIS is also a scripting-based parallel programming tool, similar to the above systems. However, our system is specialized for GIS applications, and optimized to MapReduce computing infrastructure. Without requiring users' definition, data partitioning and merging are computed and performed automatically based on simulation. Tasks are scheduled dynamically by analyzing the status of the current workflow and cluster.

5 Conclusions and Future Work

Data-intensive GIS applications are beyond the capacity of what a single processor can process in a reasonable amount of time. The challenge requires us to resort to some form of parallel computing. Unfortunately, writing parallel programs is inherently difficult especially for those scientific programmers most of who have not been well trained. MapReduce architecture provides a promising way to parallelize the existing applications. We presented an platform for running distributed GIS applications over MapReduce clusters. The experiments show that MRGIS can improve the performance significantly.

In future, we plan to extend our current system to support more GIS operations and applications. We will improve the scheduling algorithm using better data locality. The user-interface will also be improved to be more user-friendly.

References

[1] ESRI. http://www.esri.com/.

[2] GRASS. http://grass.ibiblio.org/.

[3] Hadoop. http://lucene.apache.org/hadoop/.

[4] pyDev. http://pydev.sourceforge.net/.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th symposium on Operating systems design and implementation (OSDI)*, pages 137–150. USENIX Association, 2004.

[6] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[7] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*. ACM, 2007.

[8] H. Liu and D. Orban. Gridbatch: Cloud computing for large-scale data-intensive batch applications. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 295–305. IEEE Computer Society, 2008.

[9] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 1099–1110. ACM, 2008.

[10] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[11] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 1029–1040. ACM, 2007.

[12] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Proceedings of the IEEE International Workshop on Scientific Workflows (SWF)*, pages 199–206. IEEE, 2007.