AUTO-TUNED OPTIMIZED PARALLEL I/O FOR GISCIENCE AND SPATIAL APPLICATIONS

BY

BABAK BEHZAD

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Advisers:

Professor Marc Snir
Associate Professor Shaowen Wang

ABSTRACT

Reading and writing big data is increasingly becoming a major bottleneck of using high-performance computing systems as we are heading towards the Exascale era. An unprecedented amount of data is being produced everyday by different sources. On the other hand, the computation power of HPC systems is getting scaled to hundreds of thousands cores. However, for an application to be able to utilize this much data and computation power, using I/O effectively is a must. One of the fields dealing with huge amount of data is geographic information science. In this thesis, we have implemented a parallel I/O library specialized for spatial data analysis in GIScience, capable of treating different I/O patterns such as Row-Wise, Column-Wise and Block-Wise I/O. We then establish an auto-tuning framework for finding optimal parallel I/O configurations. This auto-tuning framework is based on genetic algorithm and works on a range of configurations from the parallel file system all the way up to spatial data-analysis applications. The results and findings of a set of I/O intensive experiments executed on large HPC systems are also presented to demonstrate the effectiveness of the framework.

To my parents and my beloved sister, for their love and support.

ACKNOWLEDGMENTS

I would like to thank my advisor professor Marc Snir for his help and support and my co-advisor professor Shaowen Wang whose help and insight in geospatial context was vital to get this work done.

I also should thank The HDF Group and Lawrence Berkeley National Laboratory for supporting me as a summer intern in The HDF Group in developing a framework for Auto-tuning of parallel I/O parameters for HDF5 applications.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

One of the biggest challenges scientists are facing is dealing with the huge volume of data, often called the data deluge [1]. In addition to theory and the experiments, computation and computer simulation has been accepted as the third pillar of science and as the data needed for these simulations are increasing with unprecedented rates, the fourth paradigm is being shaped: data-intensive sciences [2]. Geographic Information Science (GIScience) is one of the interdisciplinary fields facing big data challenges in some unique ways.

Improvements in technology make instrumentes such as remote sensing satellites, GPS devices, etc. generate spatial (i.e. geographically referenced) data with higher spatial and temporal resolutions. Storing, searching, managing, analyzing and sharing of big spatial data poses a significant challenge in GIScience. An important approach to data-intensive challenges such as those in GIScience is High Performance Computing (HPC), in particular parallel computing. As a general technological trend of computing architecture, single-core and sequential computing architecture has been replaced by multicore parallel computers. Furthermore, high performance computers based on parallel computing architecture have been increasing used for enabling data-intensive computation for spatial analysis and modeling.

Parallel computing is a well accepted approach for solving data-intensive scientific problems across numerous fields. Particular to parallel computing for spatial data analysis, typically, a problem is divided into a number of parts and each part is treated by an element within a computer will all of the involved elements coordinated to achieve a cohesive solution to the original problem. Spatial domain decomposition is an important strategy in parallel computing of geographic information analysis and has been widely employed. An important question in spatial domain decomposition is how to determine the sizes of data partitions for efficiently matching with specific

computing elements. This question is especially challenging when massive data require to be decomposed. Another challenge related to big spatial data lies at regular and intensive workload pertaining to input and output (I/O) of data partitions within individual computing elements and across the elements.

As the parallelism used in the applications grows, there is no way to do the I/O operations sequentially. As parallelism is exploited in the computation aspects of the parallel computer, there is a need for a mechanism to support parallelism in I/O. Nowadays, parallel I/O is an unavoidable part of modern high-performance computing (HPC), however, unfortunately currently there is a large gap between I/O technology and other components of a supercomputer both from hardware and software perspective. This gap is of orders of magnitude and is still growing with a rapid pace of the advancement in processor technology, which is way ahead of the advancement in the storage and I/O technologies. The rate of producing data highlights this gap as we can see a big mismatch between the ability to produce and store/analyze data [3].

A unique characteristic of parallel I/O is that the software has a layered structure often regarded as the HPC I/O stack, each components with their own settings; This system-wide dependencies has eluded optimization across platforms and applications and makes parallel I/O the bottleneck of the HPC applications, especially as they become increasingly data-driven. The good news is that various studies [4] [5] [6] have shown that dramatic improvements are possible when each of these I/O stacks parameters are set appropriately. However, having multiple layers in the HPC I/O stack, each with its own optimization parameters, makes finding the optimal parameter values a very complex problem. Additionally, optimal sets do not necessarily translate between use cases, since tuning I/O performance can be highly dependent on the individual application, the problem size, and the compute platform being used. In particular, various typically encountered I/O patterns can have very different demands on the system to achieve adequate throughput [7].

Tunable parameters are exposed primarily at three levels in the I/O stack: the file system, middleware, and high-level data-organization layers. HPC systems need a parallel file system, such as Lustre [8], to store data in a parallelized fashion. Middleware communication layers, such as MPI-IO, support this kind of parallel I/O and offer a variety of optimizations, such as collec-

tive buffering. Scientists and application developers often use HDF5 [9] and NetCDF [10], high-level cross-platform I/O libraries that offer hierarchical object-database representation of scientific data.

There are two approaches to effectively tune I/O parameters. One is model-driven optimization: develop an analytical model for the process and optimize this model for the tunable parameters. Unfortunately, this model is a specific to each platform and the complex I/O stack makes extremely nontrivial. The second approach is an empirical optimization technique, also called auto-tuning. Since auto-tuning can potentially be an autonomous and portable approach, it has been investigated as a solution to similar problems such as compiler optimization [11]. In order to auto-tune an application, it is necessary to match the I/O demands with high-performing configuration sets for similar use cases. Traversing the search space of possible configurations is intractably costly; the number of permutations is huge, and a test run takes a nontrivial amount of time (sometimes on the order of minutes). Additionally, the timing results have serious nondeterministic contributions (e.g. bandwidth contention from other jobs), making the application of traditional optimization techniques very difficult. As such, heuristic approaches offer an attractively simple approach, albeit one without formal guarantees on convergence or result quality.

In order to solve the challenge of finding the optimal parameters for parallel I/O, this research uses auto-tuning in parallel I/O context. Additionally, another novel contribution of this research deals with taming the data-intensive nature of GIScience applications, by conducting this auto-tuning not only on the I/O stack but at the application level, given the knowledge about the domain of the application. We have developed a simple parallel I/O library specific to data-intensive raster-based GIScience applications, abstracting the frequent I/O access patterns of these applications, related to the decomposition of these applications, namely Row-based parallel I/O, Column-based parallel I/O and Block-based parallel I/O. Our auto-tuning approach framework tunes the parameters of the parallel I/O stack as well as determining the sizes of data partitions of each processing element conducting I/O, providing a solution for both of the challenges introduced above.

To our knowledge, there has not been any kind of auto-tuning work that covers this much variety of space ranging from parallel file-system configurations all the way up to the I/O pattern of the application. In this general

auto-tuning framework, based on the platform that the application is being run on and the number of processors it is using, a genetic algorithm tries to optimize file system properties, middle-ware settings, high-level I/O library settings and the parameters of the I/O pattern of the application. The authors believe this framework will be very useful as we make progress in data-intensive scientific world.

This thesis is organized as the following: Chapter 2 reviews the literature and introduces the definitions and backgrounds needed for the rest of the thesis. After describing the background, in chapter 4 we describe the parallel auto-tuning framework that we have developed. 3 describes the parallel I/O library that we have developed specifically for Geospatial applications and the results of using the auto-tuning framework are shown in chapter 5. Finally, we conclude our work in Chapter 6.

CHAPTER 2

OVERVIEW OF PARALLEL I/O

There are several ways to address I/O needs of an application. Sequential applications use POSIX I/O, a common way of performing I/O in applications, in which files are treated as sequences of bytes [12]. POSIX I/O works well in sequential applications in which there is only one processor reading and writing files.

Parallel applications in which multiple processors are used, however, need a different way of performing I/O operations as they are not able to efficiently conduct concurrent POSIX I/O operations on a single file. Researchers have come up with different ways of conducting parallel I/O. The most straightforward way is to have one processor to do the I/O after collecting (or distributing data in case of a read operation) to a single file sequentially as shown in Figure 2.1. A problem with this approach is that a single processor easily becomes the I/O performance bottleneck.

In order to address the scalability issue of the single-processor I/O approach, all the processors can perform their I/O independently using POSIX I/O as shown in Figure 2.2. As every processor conducts its I/O in parallel this approach seems to be working better than the previous approach. However, a drawback to this approach is that when we use it with a large number of processors, handling many files becomes difficult and time-consuming. The third approach is to use parallel I/O libraries, which means that all processors perform I/O operations on a single shared file as shown in Figure 2.3.

As shown in Figure 2.4, parallel I/O capabilities are often organized into multiple layers of system software and libraries. At the bottom of this I/O stack, a parallel file system handles all the low-level activities and managing storage devices. As opposed to common serial file systems such as NTFS and ext2, parallel file systems are optimized to be used within a parallel application performing parallel I/O. One of the widely used, open-source parallel file systems is Lustre [8]. The internals of Lustre are discussed in

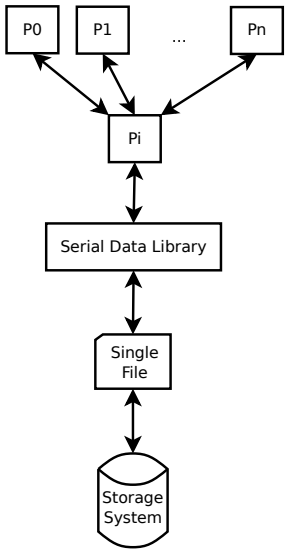

Figure 2.1: Having a single processor responsible for I/O operations in a parallel application, adopted from [12].

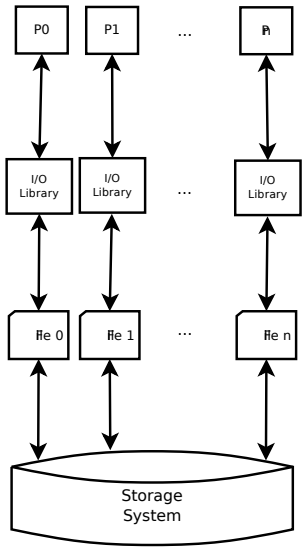

Figure 2.2: Having each processor handle I/O of its own file in a parallel application, adopted from [12].

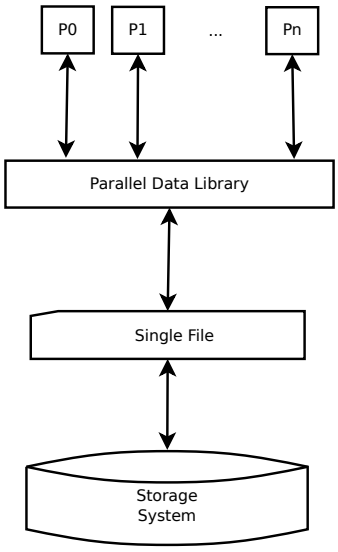

Figure 2.3: Having each processor perform its I/O to a shared single file in a parallel application, adopted from [12].

section 2.1. In the middle layer, I/O middleware is in charge of the interface between groups of processes and the file system. MPI-IO is widely used as I/O middleware that is provided in the MPI programming model. A brief description of MPI-IO is in section 2.2. Using MPI-IO is not trivial and getting high I/O performance out of it requires expertise. Therefore, it is common to use higher-level I/O libraries on top of the middleware layer. The data model that these high-level I/O libraries provide often match well with the application-level data models. As examples, HDF and NetCDF high-level libraries are discussed in section 2.3.

2.1 Lustre File System

Parallel file systems provide high-speed concurrent file access to more than one node of a computer cluster. The state-of-the art parallel file systems including Lustre [8] and GPFS [13] are scalable to thousands of nodes connecting to peta-byte storages with very high throughput. In order to come

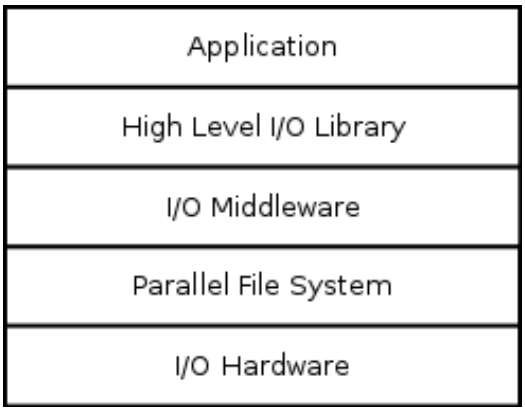

Figure 2.4: Current I/O Stack of an application, adopted from [12].

up with an efficient parallel I/O library, one needs to understand the details of the underlying file system, its components, the concepts and how to set their settings. Since Lustre is an open source file system and is being widely used, we are going to concentrate on its design and parameters in this thesis.

2.1.1 Lustre File System Components

The Lustre file system has two important components. It divides data operations into the Object and Metdata operations. Object operations deal with the data itself and the Metadata operations deal with the metadata (i.e filenames, directories, permissions, etc.). For the Object operations, there are two components: Object Storage Servers (OSSs) which are the I/O servers and the Object Storage Targets (OSTs) which are the storage devices. File metadata is controlled by a Metadata Server (MDS) and stored on Metadata Target (MDT). This separation of operations is one of the key feature of Lustre's scalability.

Being able to stripe the files is another important feature of the Lustre file system which basically means dividing a single file across multiple disks by separating its sequence of bytes into chunks. The reason to do this is to have the read and write operations done in parallel by accessing multiple OSTs increasing the available I/O bandwidth. Also, file striping makes available space for very large files, which can not be fit into a single OST. In order to do file striping, Lustre defines two parameters as follows:

- **Stripe count:** The number of OSTs that the file is striped on.

- **Stripe size:** The number of bytes written in each stripe

2.2 MPI-IO

The MPI-2 standard provides a parallel I/O interface with the portability goal. ROMIO [14] is the major implementation of MPI-IO and is distributed through different MPI libraries. However, since the MPI-IO interface is hard to use, higher level libraries are developed on top of it such as NetCDF and HDF5.

Similar to sequential I/O, before the I/O operations, programmers have to either open(an existing file) or create a file in MPI-IO using `MPI_File_Open()` or `MPI_File_create()` function calls. These function calls have different arguments that one has to pass to them, but we are not going to get into those details here. Gropp et al. [15] has the full description of how to use MPI-IO interface. At the end of the I/O operations, programmers should close the file by using `MPI_File_close()` function call of the MPI interface.

MPI-IO has an important concept named file *view*, which is used to determine the section of the file accessible processors [15]. File views should be set by the programmer using `MPI_file_set_vew()` function. If the file view is not set by the programmer, the whole file will be accessed by the processors [12]. `MPI_file_set_vew()` has several arguments including one to specify the unit of data in the file called *etype*. It also includes *filetype* as an argument, a MPI derived datatype or a basic datatype used by the programmer to assign each part of the file to each processors doing the I/O.

Once the file view is set, programmer may use different read and write functions such as `MPI_File_read()` and its variants and `MPI_File_write()` and its variants.

ROMIO has implemented a couple of optimizations which have made parallel I/O much more efficient. Parallel I/O can be of two general types: (1) Independent I/O operations in which MPI-IO user specifies what each of the processors should do. (2) Collective I/O which is like a more coordinated way of doing I/O by a subset of processors. [16]. One of the algorithms that we are going to tune its parameters later in this thesis is the collective buffering algorithm implemented in the MPI-IO library. Collective buffering is an optimization for I/O, in which instead of having all the processors to

do the I/O, a subset of them perform the I/O operations. This is an effort to combine I/O requests of all the processors into a subset of I/O requests. The processors in this subset are called *aggregators* [12]. The reason of doing this mainly is to reduce the pressure on the I/O subsystem by having smaller number of nodes doing the I/O operations. Note that the data exchange between the nodes that are conducting I/O and the ones that are requesting them are done by MPI communication.

2.3 HDF and NetCDF

NetCDF(Network Common Data Form), developed by the Unidata program at the University Corporation for Atmospheric Research (UCAR). It provides a set of libraries for creating, reading and writing a machine-independent data format for scientific data having structured datasets, which are basically set of multidimensional arrays. It is used by a variety of sciences and since 2009, the Open Geospatial Consortium(OGC) has standardized the NetCDF format as an open standard. The goal of this standardization effort is to have the NetCDF with CF (Climate and Forecast) conventions recognized as an international standard for encoding georeferenced data in binary form.

Before getting into the details of the georeferenced data in NetCDF, an introduction to NetCDF is required. NetCDF format has three main components namely, *dimensions*, *variables* and *attributes*. It is made up of two parts: file header and the data. File header contains the metadata information about the dimension, attributes and variables, whereas the data section contains the value of each variable. A dimension has a name and a length and variables are based on them. For instance, latitude and longitude can be two dimensions of a georeferenced data, containing the dimensions of the grid. Variables are for storing data and look like multidimensional arrays. As stated earlier, a variable is defined by a list of dimensions. Therefore, a variable has a name, the type of the data stored in that variable and a list of dimensions. Last but not least, attributes are used to store information about the variables.

At around 2005, Unidata released NetCDF-4 which uses HDF5 in order to have more optimized I/O by supporting parallelism and several optimizations. HDF(Hierarchical Data Format) is also a portable file format, de-

veloped at National Center for Supercomputing Applications (NCSA). It is currently being maintaned by The HDF Group. HDF5, the latest version of HDF, has a hierarchical structure for storing the data and supports reading and writing the data in parallel making use of MPI-IO.

The primary reason that we have chosen NetCDF-4 over all the other GIS data formats for raster-based data formats(such as GeoTiff, etc.) is that NetCDF-4 supports parallel I/O and therefore can be used in data-intensive GIScience using parallel I/O capabilities. To our knowledge, NetCDF-4 is the only one supporting parallel I/O and also accepted as a standard in OGC(Open Geospatial Consortium). GDAL [17] has recently anounced its support for NetCDF-4.

CHAPTER 3

A PARALLEL I/O LIBRARY FOR SPATIAL DATA ANALYSIS

As parallel processing has been increasing used in data-intensive spatial analysis, various spatial domain decomposition methods have been developed. A large number of these methods are based on 2-dimensional spatial representations that are often classified as field-based and object-based [18]. In the case of field-based representation, basic spatial units such as row, column, and block(either regular or irregular) are often used to determine the granularity and size of decomposed spatial sub-domains for parallel processing [19]. This section describes a parallel I/O library tailored to spatial data analysis based on NetCDF.

3.1 A Parallel I/O Library for Data-Intensive Spatial Analysis

I/O is often regarded as a bottleneck of exploiting high-performance and parallel computing resources in data-intensive spatial analysis and modeling. In order to be able to analyze large datasets by efficiently taking advantage of high performance computing, it is imperative to establish effective I/O strategies tailored to spatial characteristics. However, using low-level libraries requires in-depth understanding of MPI and parallel I/O and, thus, is not suitable for GIScientists and programmers. This research aims to establish a simple to use library of parallel processing functions for reading and writing spatial data based on NetCDF-4 library.

This library is named pGIOLibrary and is written in C programming language and aims to abstract the features of NetCDF-4 that are used in spatial data analysis. These features include *dimensions*, *variables* and *attributes*. For each of these features, the library provides simple functions to read and write, and uses the NetCDF API functions internally.

In addition to interfacing these elementary types, pGIOLibrary, uses NetCDF API functions for conducting I/O with an additional option to configure a small set of parameters for spatial domain decomposition. The patterns currently supported in the library are: `ROW_WISE`, `COLUMN_WISE`, and `BLOCK_WISE`. Corresponding methods and their implementations of the library are discussed as follows.

3.2 Different Methods of Conducting Parallel I/O

A straightforward way to conduct I/O for a raster spatial data is row-wise. In case of sequential I/O, one processor may start from the first row of a raster and issue reads/writes with buffers equal to the size of one row or multiples of rows. Buffers are arrays in memory containing the data. This scheme can also be applied as a parallel I/O strategy by dividing the rows of the raster into different chunks for each processor of the parallel system and have each of the processors do the I/O for its own section.

NetCDF-4 does parallel read/write to/from variables by making use of two arrays called `count` and `start`. The NetCDF-4 function declaration of a typical variable write looks like the following:

```
int nc_put_vara_ type (int ncid, int varid, const size_t start[],
const size_t count[], const type *valuesp);
```

Based on NetCDF-4 documentation, `start` is an array of integers specifying the index in the variable where the first of the data values will be written. `count` is an array of integers specifying the edge lengths along each dimension of the block of data values to be written. The length of count is the number of dimensions of the specified variable.

Given the rank of each processors in a parallel computing system and using appropriate start and count variables, we have implemented three popular I/O schemes as explained in this section.

3.2.1 Row-Wise Parallel I/O

Figure 3.1 shows how pGIOLibrary conducts Row-wise parallel I/O. Each processor in the parallel computer is in charge of reading/writing one chunk

of rows of the 2D raster. These methods are not restricted to just 2D data and can be used for any number of dimensions.

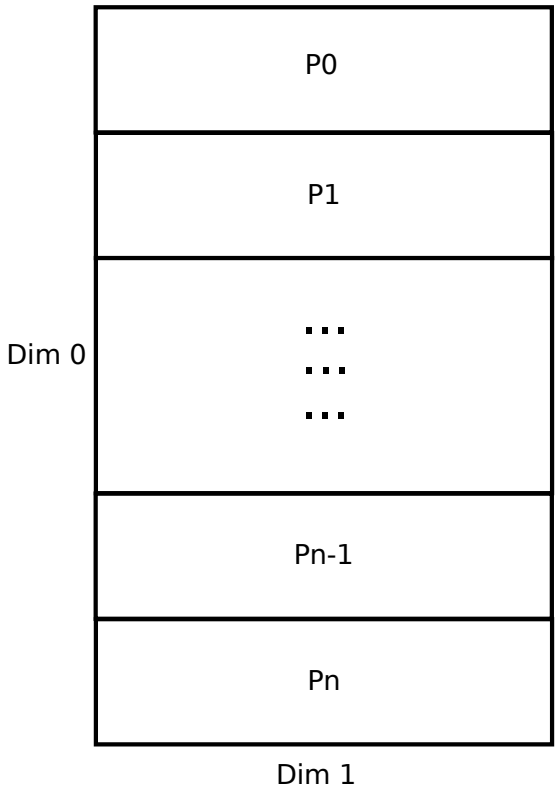

Figure 3.1: Row-wise Parallel I/O on a 2D raster dataset by n cores

The way pGIOLibrary implements Row-wise parallel I/O is simply using general formulas for `count` and `start` arrays based on the `mpi_size` and `mpi_rank` variables of a parallel program. The values of these arrays for 2D rasters are provided in Table 3.1.

Array Name	Element 0	Element 1
count	$\frac{dim\_0\_length}{mpi\_size} * mpi\_rank$	dim_1_length
start	$\frac{dim\_0\_length}{mpi\_size}$	0

Table 3.1: Values of `count` and `start` for conducting Row-Wise parallel I/O

3.2.2 Column-Wise Parallel I/O

As it can be seen in Figure 3.2, Column-wise parallel I/O is the transpose of Row-wise and therefore its `start` and `count` arrays are going to be the transpose of Row-wise as shown in Table 3.2.

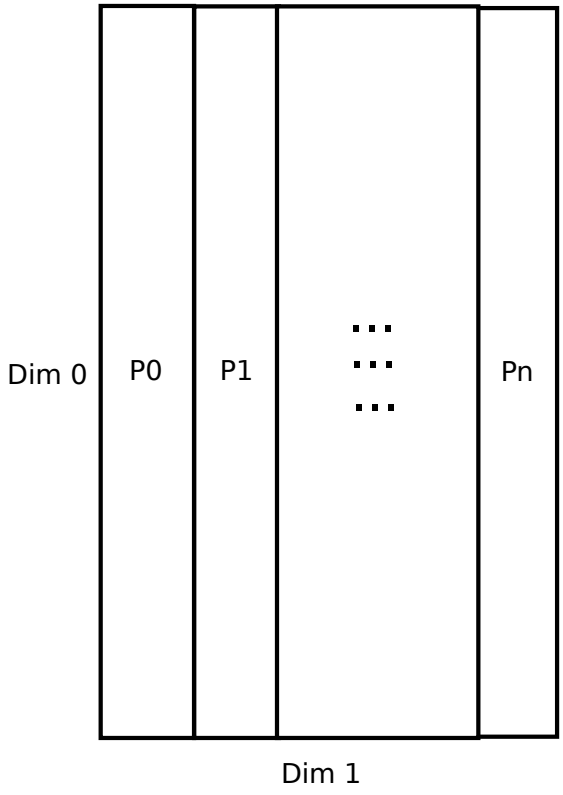

Figure 3.2: Column-wise parallel I/O on a 2D raster dataset by n cores

Array Name	Element 0	Element 1
count	dim_0_length	$\frac{dim\_1\_length}{mpi\_size} * mpi\_rank$
start	0	$\frac{dim\_0\_length}{mpi\_size}$

Table 3.2: Values of `count` and `start` for conducting Column-Wise Parallel I/O

3.2.3 Block-Wise Parallel I/O

Implementing Block-wise parallel I/O using `start` and `count` variables, however, is more complicated. One restriction is on the dimension of a raster.

This implementation just supports 2D rasters. Also, another restriction of this implementation is that the number of processors should be a complete square. This way, we can divide each of the two raster dimensions into $\sqrt{mpi\_size}$. If the values of `count` and `start` variables are set to the values shown in Table 3.3, each processor with id in range $0, ..., \sqrt{mpi\_size} - 1$ is going to write the first row of blocks, processors with id $\sqrt{mpi\_size}, ..., 2 \times \sqrt{mpi\_size} - 1$ will write the second row of blocks and so on. This concept is illustrated in Figure 3.3.

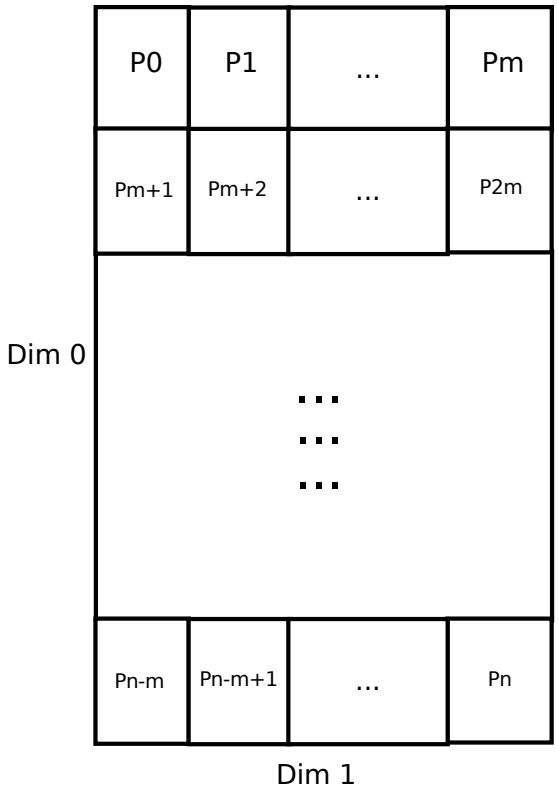

Figure 3.3: Block-wise parallel I/O on a 2D raster dataset by n cores

Array Name	Element 0	Element 1
count	$\frac{dim\_0\_length}{\sqrt{mpi\_size}}$	$\frac{dim\_1\_length}{\sqrt{mpi\_size}}$
start	$(\frac{mpi\_rank}{\sqrt{mpi\_size}}) * count[0]$	$(mpi\_rank \% \sqrt{mpi\_size}) * count[1]$

Table 3.3: Values of `count` and `start` for conducting Block-Wise Parallel I/O

3.3 Parameterized Methods of Conducting Parallel I/O

As it is stated in the Introduction, the goal of this work is to optimize parallel I/O settings from system settings such as the file-system all the way up to application-specific characteristics. To this end, the common I/O patterns seen in spatial data analysis (row-wise, column-wise and block-wise) have been implemented as a library. However, in order to optimize these application-specific patterns, they should be parameterized. Methods discussed in last section are useful when a user does not want to get her hands dirty with the parameters capturing these characteristics, i.e. she does not want to specify how many rows should a processor core read/write. In this case, the library calculates the number of rows that each processor should deal with. But this may not be the most optimal configuration. In order to give users the opportunity of changing these settings, some new capabilities have been added to the pGIOLibrary by which users can specify the settings of each of the patterns. The following sections review the implementation of each of these parameterized methods for doing I/O.

3.3.1 Parameterized Row-Wise Parallel I/O

In this case, the number of rows is equal to the length of dimension 0 (`dim_len_0`) and the number of rows that each of the processors should read/write is given by the user as a parameter we call `num_0_per_PE`. The first thing to calculate is the number of rounds that all the processors should do in order to finish all the rows. This number can be calculated as the floor of the result of dividing `dim_len_0` by `num_0_per_PE`. The remainder of this division is what is known as *padding* and there are several strategies to write those paddings. After finding the value of number of rounds, a loop of size this value is necessary to write each of the stripes that are written by the processors. The `count` and `start` of these stripes (which is equal to the number of rows) given to the NetCDF-4 write function is provided in table 3.4. k is the index of the loop.

Array Name	Element 0	Element 1
count	$num\_0\_per\_PE$	dim_1_length
start	$num\_0\_per\_PE \times (k * mpi\_size + mpi\_rank)$	0

Table 3.4: Values of `count` and `start` for conducting parameterized Row-Wise Parallel I/O

3.3.2 Parameterized Column-Wise Parallel I/O

The case of Parameterized Column-Wise I/O is again similar to Row-wise except for the fact that the elements are transposed. `dim_len_n` is used instead of `dim_len_0` since the n^{th} dimension is the dimension that is going to be divided between the processors, Table 3.5 shows the values for this kind of I/O. k is the index of the loop that is conducting these I/O operations and is equal to the number of rounds.

Array Name	Element 0	Element 1
count	dim_0_length	$num\_0\_per\_PE$
start	0	$num\_0\_per\_PE \times (k * mpi\_size + mpi\_rank)$

Table 3.5: Values of `count` and `start` for conducting parameterized Column-Wise Parallel I/O

3.3.3 Parameterized Block-Wise Parallel I/O

In order to implement parameterized Block-Wise I/O operations, pGIOLibrary uses a slightly different approach. At first, we introduce a notion of a block as a structure which has three elements: `start_0`, `start_1` and `id`. Since the width and height of each block is known, the end of each block is not stored. Same as in parameterized row-wise and column-wise the number of blocks in each row and column are determined. An array of blocks of size $\frac{dim\_len\_0 \times dim\_len\_1}{num\_0\_per\_PE \times num\_1\_per\_PE}$ is then created and the starts of each of the blocks along with their id are set. In order to write, this array should be divided among processors and each processor can write each of its blocks. There are several ways to partition this array between processors and some of them are implemented in pGIOlibrary. The most notable one is a circular division in which processor i get block number (i % mpi_size).

CHAPTER 4

AUTO-TUNING OF PARALLEL I/O PARAMETERS FOR HDF5 APPLICATIONS

Numerous tunable parameters exist at each level of the current state-of-the-art I/O stack and they greatly affect the I/O performance of HPC applications. The lack of application-specific tools available to determine optimal values for these parameters has caused scientists either to use the default values(sometimes with a very bad I/O performance!) or guess the values of these parameters based on their experience and hardcode them into the application. Hardcoding the values of these parameters into an application means that adjusting these values and also porting the application to a new platform requires recompiling the application.

In this research we have impelemented an auto-tuning framework that can be used to identify appropriate HDF5/NetCDF-4 parameters, MPI-IO parameters and Lustre settings on a given platform. This framework eliminates the need of recompiling the application by using dynamic libraries and a configuration file from which the application reads the values of the parameters and set them before the I/O operation of the application. This way, there is no need to hard-code the parameters in the application. We have developed the H5Tuner library, a shared library that can be preloaded before the HDF5 library and executes before HDF5 is invoked, read the configuration file and set the values before calling the HDF5 function.

A diagram of the application functional schematic is shown in Figure 4.1; the contributions of this work are primarily the middle section: collecting performance statistics and refining the parameter search process using a genetic algorithm approach. As [20] shows, this work as part of the HDF5 AutoTuning project is a joint research between the HDF group and Lawrence Berkeley Lab to develop an auto-tuning framework for parallel I/O of scientific applications based on the HDF5 library.

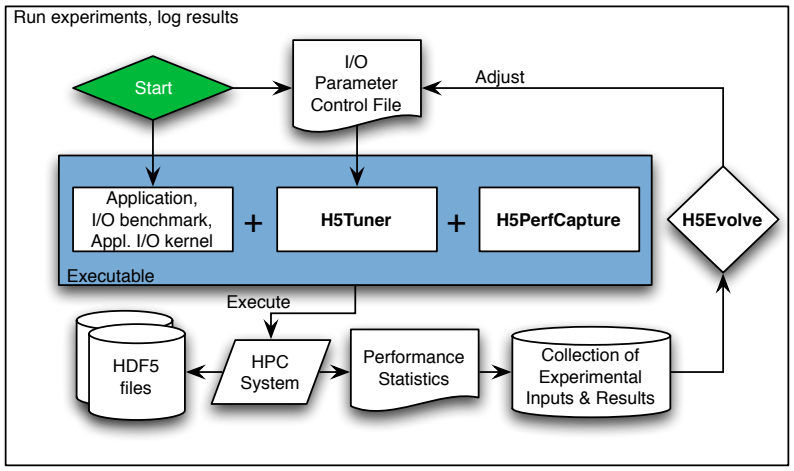

Figure 4.1: A functional schematic of the HDF5 AutoTuning project

4.1 H5Tuner

The H5Tuner dynamic library is developed as a tool to set the parameters of different levels of the I/O stack (i.e. HDF5 level, MPI-IO level, File system level). In order to accomplish this task, it makes use of the dynamic linking concept. The H5Tuner gets preloaded before the HDF5 library, once a HDF5 function is called in the application, H5Tuner's function will get executed first. In the execution of this function, all the parameters to be tuned are set. After setting these variables, the original HDF5 function is called with the arguments that the user has specified. The values for the parameters are specified by the user or as we will see in the next section can be specified by an optimization framework. The values for each of the parameters are written in an XML file with different sections for different I/O stack levels. This XML configuration file resides in the user's home directory, so that H5Tuner can open it and read each of the sections of it and calls the appropriate functions for setting the values. An example of this configuration files is shown in figure 4.2. MiniXML [21] library, a small XML library to read and write XML data files, is used in the H5Tuner library.

Another capability of H5Tuner is to set these parameters on a per file basis in order to have a finer granularity in setting these parameters. In order to achieve this goal, each of the elements in the configuration file can have an attribute named *FileName* as shown in figure 4.3. H5Tuner library checks this FileName value with the name of the file being used and sets the

```
<?xml version="1.0" ?>
<Parameters>
  <High_Level_IO_Library>
    <alignment>
      2097120,1048560
    </alignment>
    <sieve_buf_size>
      536870912
    </sieve_buf_size>
  </High_Level_IO_Library>
  <Middleware_Layer>
    <cb_nodes>
      4
    </cb_nodes>
    <cb_buffer_size>
      4194304
    </cb_buffer_size>
  </Middleware_Layer>
  <Parallel_File_System>
    <striping_factor>
      16
    </striping_factor>
    <striping_unit>
      4194304
    </striping_unit>
  </Parallel_File_System>
</Parameters>
```

Figure 4.2: An example of an H5Tuner XML configuration file

corresponding configuration only if these names match. In case no FileName is provided, the configuration will be set for all the files.

```
<Parameters>
        <High_Level_IO_Library>
                <H5Pset_alignment> 0 </H5Pset_alignment>
        </High_Level_IO_Library>

        <Middleware_Layer>
                <cb_buffer_size> 4194304 </cb_buffer_size>
                <cb_nodes> 32 </cb_nodes>
        </Middleware_Layer>

        <Parallel_File_System>
                <striping_factor FileName="sample_dataset.h5part"> 4 </striping_factor>
                <striping_factor FileName="sample_dataset_1.h5part"> 16 </striping_factor>
                <striping_unit> 65536 </striping_unit>
        </Parallel_File_System>
</Parameters>
```

Figure 4.3: An example of an H5Tuner XML configuration file with different settings for different file names

4.2 Parameters Space

The benchmarks will be run with differing configurations of 5 parameter values (although the framework developed can be readily expanded to include different parameters):

Parameter	Min	Max	# Values
strp_fac	4	156	10
strp_unt	1 MB	128 MB	8
cb_nds	1	256	12
cb_buf_siz	1 MB	128 MB	8
*num_rows	1	1024	9
*num_columns	1	1024	9
*num_blks	(1,4096)	(8192,32768)	48

Table 4.1: A list of the tuned parameters in the search space, the range of values, and the number of discrete values allowed. One of the parameters with * is being auto-tuned at a time.

- Lustre stripe factor (`strp_fac`)
  - number of OSTs over which a file is distributed
- Lustre stripe unit (`strp_unt`)
  - number of bytes written to an OST before cycling to the next
- MPI-IO number of collective buffering nodes (`cb_nds`)
  - max number of aggregators for collective buffering
- MPI-IO collective buffer size (`cb_buf_siz`)
  - size of the intermediate buffer for collective I/O
- Geo-Spatial parameters
  - Row-wise: Number of rows assigned to each processor
  - Column-wise: Number of columns assigned to each processor
  - Block-wise: Size of the blocks, containing the size in `x` axis and `y` axis

Table 4.1 shows the range and number of possible values for each of the parameters. Based on these numbers the entire configuration space ranges from 69120 to 368640 configurations. Considering that each of the runs take a long time to finish, it is clear that a brute-force approach is not feasible.

4.3 Optimization Framework

4.3.1 Genetic Algorithms

Genetic Algorithms are heuristics for approximating optimal solutions in complex search spaces, particularly those that are ill-suited for traditional exact or approximation methods [22].

Genetic algorithms work with a population of solutions in each generation as they evolve. The evolution process is by crossing over two members of the population or mutating one. The goal is that this process converges to an optimal or near-optimal solution. In the first step of the genetic algorithm an initial population is generated randomly and each configuration of this initial population is evaluated. A set of intermediate configurations is then chosen based on their evaluation fitness value and in order to create the next population this set goes through mutation and recombination. This process continues iteratively until convergence to a near-optimal solution [23]. Unfortunately, there is no guarantee on this convergence by the GA, however, it has been shown to converge to a near-optimal after some time. There are different methods for doing each of the selection, mutation and recombination operations. [24] describes fifteen different crossover operators.

4.3.2 H5Evolve

Our use of Genetic Algorithms is a component in our framework called H5Evolve; It is shown in Figure 4.4 and is built off the Pyevolve [25] library, which provides a framework for performing genetic algorithm experiments in Python. We use swap mutator as our mutation operator and one-point crossover as the crossover operator. As we have discussed previously, each of the members of the populations represents a member of the parameter space containing Lustre stripe settings, Collective Buffering settings and I/O patterns settings. The way that the fitness function is defined in H5Evolve is that it first creates an XML file with the values of the member that GA wants to evaluate. It then starts a timer and defines the H5Tuner library as the library to be preloaded in the application and then runs the application. This way, H5Tuner library will make sure that the parameters are set in the application. Once the application is finished, H5Evolve ends the timer and

returns the difference of the timers as the fitness value. Genetic algorithm tries to minimize this fitness value as it makes progress.

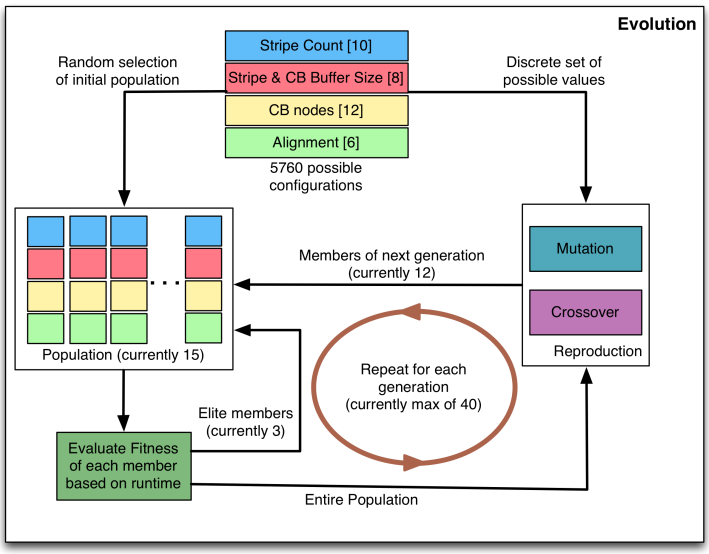

Figure 4.4: The design of the genetic algorithms used for this work.

In our experiments, we have used population size of 15 and a generation number of 40 to sample the space. Mutation rate of 15% and crossover rate of 90%. The three elite members (i.e. configurations with the best evaluation result) were directly taken to the next generation. Based on our initial experiments, these numbers have shown the best performance. However, we leave the study of other approaches to this optimization problem as a future work.

Note that since our runs is structured in a way that a large batch job is submitted for the duration of the experiment on the HPC platform, this batch job calls H5Evolve which is our main Python script making use of PyEvolve. The fitness function as explained earlier runs the code containing parallel I/O operations. Since each of the runs of the application takes a considerable amount of time, and we do not want to run the GA for a really long time, we had to use relatively small numbers for our GA parameters. In practice, we found that a smaller population (15) with a greater number of generations (40) provided a sufficient span of the sample space and a reasonable convergance time (less than 10 hours).

CHAPTER 5

EXPERIMENT RESULTS

5.1 A Simple Write Benchmark

In order to evaluate this framework we have developed a simple write benchmark which writes different-size 2D rasters using pGIOLibrary. An extension to H5Evolve is also developed so that in addition to setting the I/O stack parameters such as Lustre stripe size and strip count, it can set the I/O access pattern parameters such as the size of each blocks to be assigned to processors.

This write benchmark creates two NetCDF dimensions `X` and `Y` of different sizes based on the size of the file requested. For example for an output file size of `16GB`, we have selected a 2D raster of dimension `x=65536` by `y=32768` is written. After defining these dimensions and the corresponding variables, the write benchmark calculates the size of the array each processor ought to write. For example for this 16 GB 2D raster data, each processor should allocate an array of size $\frac{65536 \times 32768}{\#of processors}$ of a specific type such as `double`. Each processor then fills in its array with random numbers and calls a pGIOLibrary `write` function with the I/O pattern and I/O pattern options read from the configuration file (which is provided by the user or in case of auto-tuning by the Genetic Algorithm).

5.2 Experimental Platform

All of the following experiments were run on Ranger supercomputer that is a Sun Constellation cluster at the Texas Advanced Computing Center(TACC). It contains 3,936 16-way SMP compute nodes providing 15,744 AMD Opteron processors. Ranger contains 62,976 compute cores, 123 TB of memory and

1.7 PB of raw global disk space with theoretical peak performance of 579 TFLOPS. It uses a Lustre [8] parallel distributed file system for storing data on disc and relies on the MPI-IO communication standard for I/O communication among the system components.

5.3 Results of Running GA Framework on the Write Benchmark

In this section, different results of running the Genetic Algorithm framework for the write benchmarks described in the previous section are presented:

5.3.1 Row-Wise Pattern of a 16GB Raster Dataset Write using 128 processors

The set of experiments in this section were run on 128 cores of Ranger.

Figure 5.1 shows the evolution of running time of pGIOLibrary write benchmark. As GA makes progress, the trend of running time of the write application decreases. Please note that since GA is a search algorithm and has to explore different parts of the space, it does not always decrease the objective (running time in our case). However, after several generations it keeps the parameters leading to the minimum running time and the trend of running time decreases as the algorithm makes progress. The total number of 144 experiments were done by the GA in this case in a course of 4 hours. In spite of a little number of spikes in the evolution of our genetic algorithm (due to the searching nature of GA), the decrease in running time is clear. Table 5.1 shows the minimum, maximum and the converged set of these experiments in order to the influence of each of these tunable parameters on the performance of the write application.

	Exp. ID	Gen. #	Strp_Fac	Strp_buf_size	CB_Nodes	CB_buf_size	Num_0_per_PE	Time(s)
MIN	110	7	32	32 MB	256	8 MB	512	62.33
MAX	18	1	128	64 MB	96	2 MB	256	181.81
LAST_EXP	144	9	48	32 MB	8	128 MB	512	66.74

Table 5.1: Minimum, maximum and the converged value of the running time of our Genetic Algorithms framework for the Write Benchmark

Figure 5.2 shows how better performance is gained as the number of rows

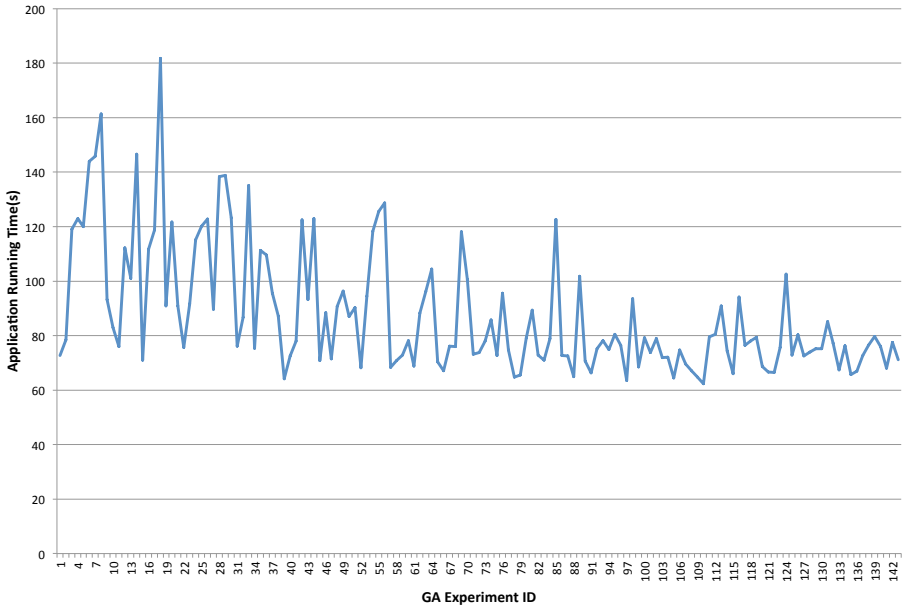

Figure 5.1: Evolution of Running Time of pGIOLibrary write benchmark for a 16GB dataset on 128 processor cores

written by each processor is changed corresponding to the sorted running time. It can be concluded from this plot that having each processor core write 512 rows (and 256 rows after that) is among the best options for this write benchmark. The plot also shows that having a very small number of rows per processors such as 1 or 2 is not a good candidate.

Figure 5.3 shows the evolution of Lustre stripe count metric with respect to the sorted running time. Again, it can be concluded that having large values for the stripe count such as 128 is not a perfect candidate neither having very small stripe counts such as 2 and 4 (the default Lustre stripe count is 4, which is not the optimum value for many parallel I/O tasks!). The good candidates for stripe count in this case are 16, 32, and 48 with 32 leading to the minimum running time. Figure 5.4 shows the evolution of Lustre stripe size in MB with respect to the decreased running time. It is very clear in the plot that GA has converged to the value of 32 MB for Lustre stripe size for this I/O pattern.

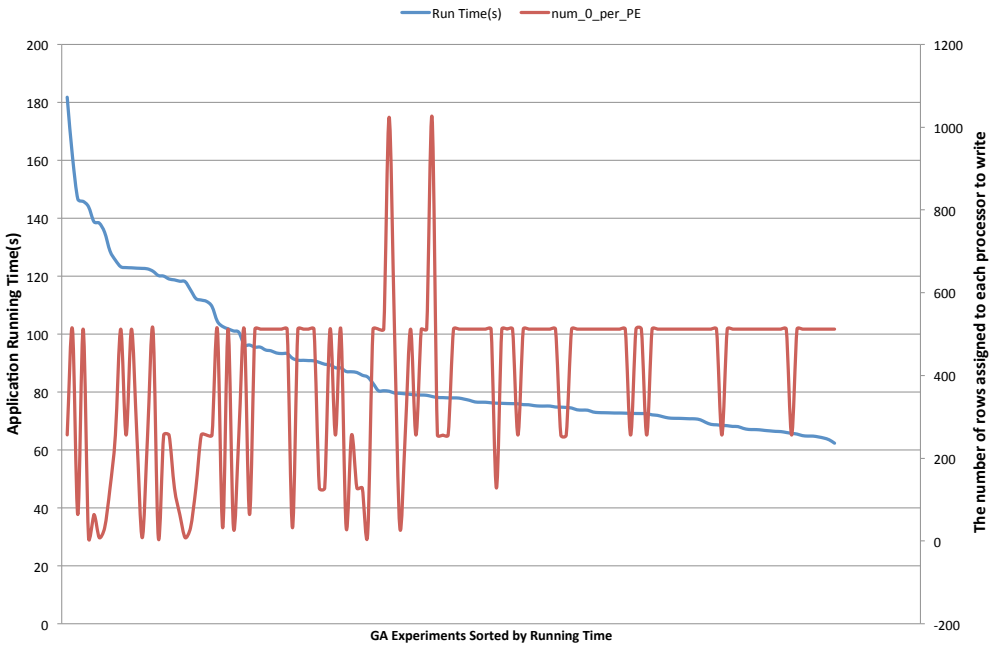

Figure 5.2: Sorted running time of the experiments with respect to the evolution of number of rows written by each processor.

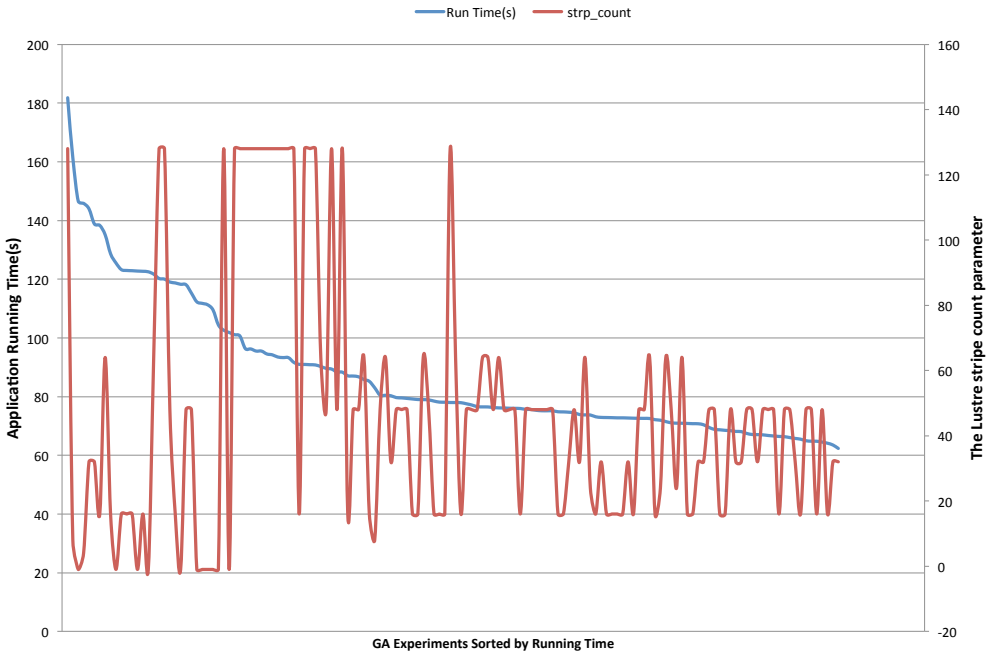

Figure 5.3: Sorted running time of the experiments with respect to the evolution of Lustre stripe count.

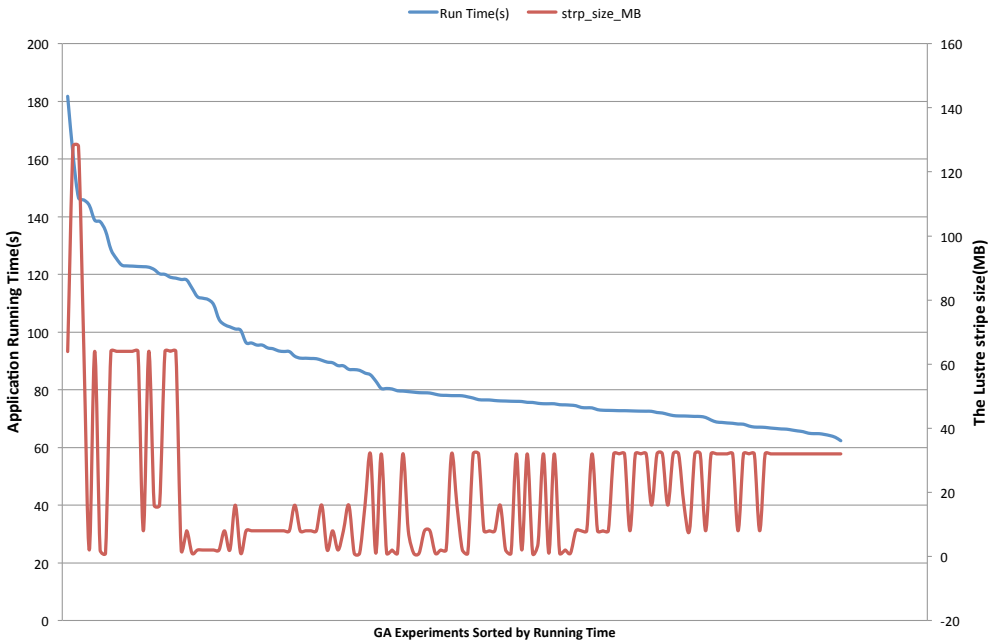

Figure 5.4: Sorted running time of the experiments with respect to the evolution of Lustre stripe size.

5.3.2 Block-Wise pattern of a 16GB Raster Data Write using 128 processors

Row-wise pattern showed a very good speedup as the genetic algorithms evolves. However, a subsequent set of experiments on block-wise I/O showed an interesting behavior: The first experiments done, showed not only good speedup, but also a huge gap between the running time of the experiments with blocks which have smaller size in their second dimension. This means that as we go forward to Column-wise I/O, the performance decreases. Figure 5.5 shows the running time of a set of experiments on the Y-axis in seconds. The X-axis shows the size of the second dimension of the blocks (`Blk_size_y`) of each experiment. As it can be seen in this figure, 9 experiments out of 27 experiments used 4096 columns for each block and all of these experiments took more than 1000 seconds to be done! This time, compared to the time that the Row-wise I/O was taking, is very high. However, the figure shows that in several experiments for those blocks with 32768 as the size of their second dimension, 70 seconds time could be achieved. These are the experiments where each block contained all of the columns of the whole raster data, i.e. Row-wise pattern (with different numbers of rows such as 256, 512, and 1024). Figure 5.6 shows the same data with sorted time. Note that the Y-axis in Figure 5.6 is logarithmic; Therefore, the performance of column-wise I/O is much worse than row-wise I/O. In the following discussion, we will try to find the reason of this bad performance.

Prior to focusing on the genetic algorithms and how H5Evolve can optimize the performance of parallel I/O, an investigation is required for this bad performance. Exploring HDF library documentation brought us to the concept of file layout in the HDF library. According to the documentations, datasets are stored as a 1-dimensional array in the file and *file layout* is defined as the mapping of these higher-dimension datasets to a sequential 1-dimensional file. The default layout of HDF files is called *contiguous* which is a row-major layout. HDF5 supports other kinds of layouts as well. One of them, related to the Block-wise pattern of this work, is called the *chunked* layout. In the chunked layout, the datasets are split into different chunks and these chunks can be read and written individually and therefore the operations can be done in parallel. In order to enable chunk layout a HDF5 function call such as `H5Pset_chunk((hid_t plist, int ndims,`

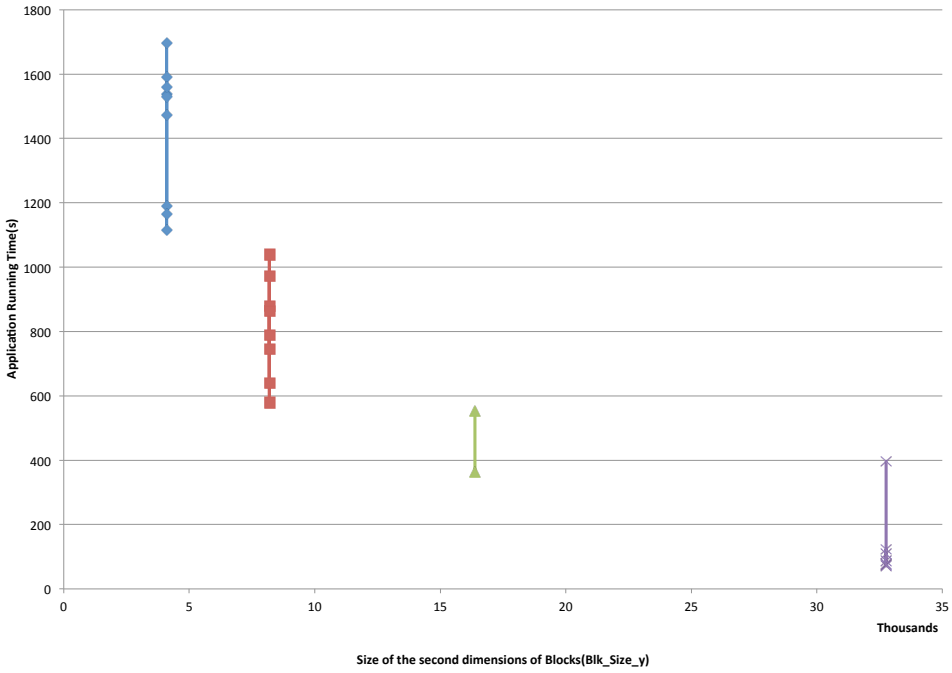

Figure 5.5: Application running time for the experiments sorted by the value of the second dimension of blocks. Block-Wise write benchmark for a 16GB dataset on 128 processor cores

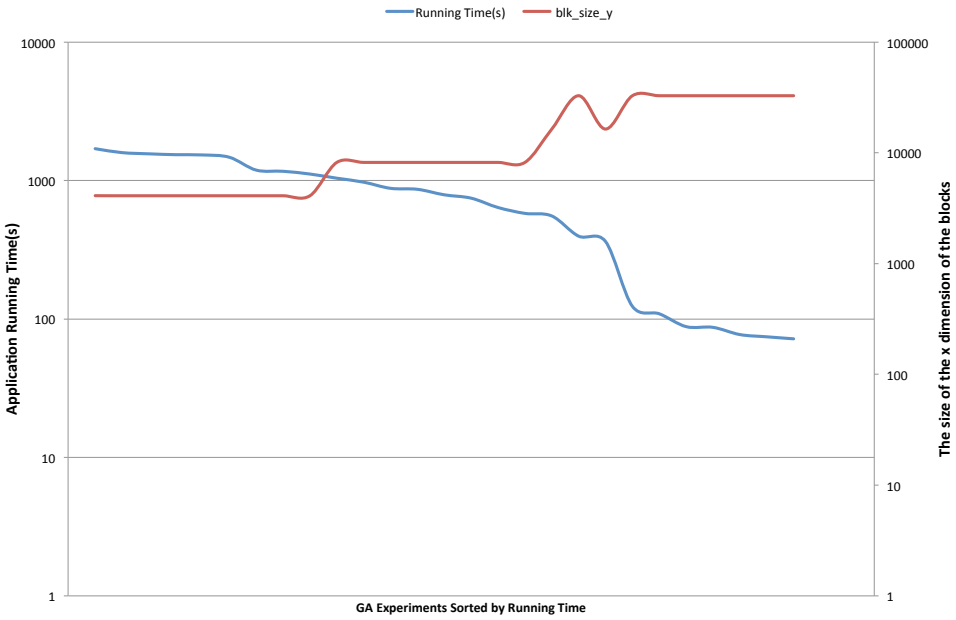

Figure 5.6: Evolution of the value of the second dimension of blocks with respect to the sorted running time. Block-Wise write benchmark for a 16GB dataset on 128 processor cores

`const hsize_t *dim )` can be used, in which `plist` is the id of the dataset, `ndims` is the number of the dimensions of each chunk and `dim` is an array defining the size of each chunk. For testing this kind of layout, a new option has been added to the XML configuration file named `chunk` with the variable name of the dataset to be used for as an attribute. Figure 5.7 shows an example of this new option.

```
<?xml version="1.0" ?>
<Parameters>
  <High_Level_IO_Library>
    <chunk VariableName="variable_2">
      16,8192
    </chunk>
  </High_Level_IO_Library>
  <Middleware_Layer>
    <cb_nodes>
      2
    </cb_nodes>
    <cb_buffer_size>
      33554432
    </cb_buffer_size>
  </Middleware_Layer>
  <Parallel_File_System>
    <striping_factor>
      8
    </striping_factor>
    <striping_unit>
      16777216
    </striping_unit>
  </Parallel_File_System>
</Parameters>
```

Figure 5.7: A configuration file for H5Tuner library in which for a variable named `variable_2` chunks of size 16×8192 has been set

After adding two new parameters to H5Evolve named `blocks_dim_0_var_2` and `blocks_dim_1_var_2`, a set of new experiments were run on Ranger and Figure 5.8 shows the same kind of plot as in Figure 5.5 but with HDF5 chunked layout. As it can be seen, a small amount of running times have been achieved for block sizes with different values for their second dimensions and not just for those blocks of shape of a row-wise pattern. Based on these experiments, Table 5.2 shows the top three I/O performance achieved for a 2D raster dataset of size 65536 × 32768 in a block-based fashion achieved.

Strp_Fac	Strp_size	CB_Nodes	CB_buf_size	Blk_size_x	Blk_size_y	Time(s)
24	4 MB	1	64 MB	256	16384	75.36
48	32 MB	24	32 MB	32	32768	80.95
64	16 MB	24	16 MB	256	4096	83.82

Table 5.2: Configuration of the top 3 I/O performance of block-based spatial pattern

Figures 5.10 to 5.11 show the evolution of the Lustre parameters for this Block-Wise I/O operations using HDF5 chunked layout.

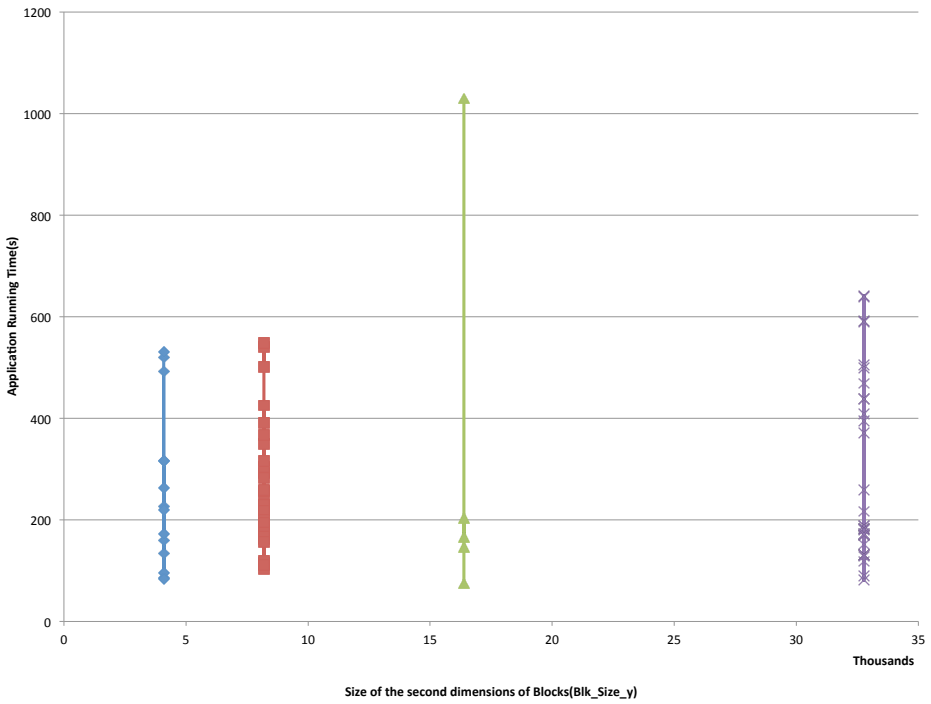

Figure 5.8: Application running time for the experiments sorted by the value of the second dimension of blocks. Block-Wise write benchmark with HDF5 chunked layout for a 16GB dataset on 128 processor cores

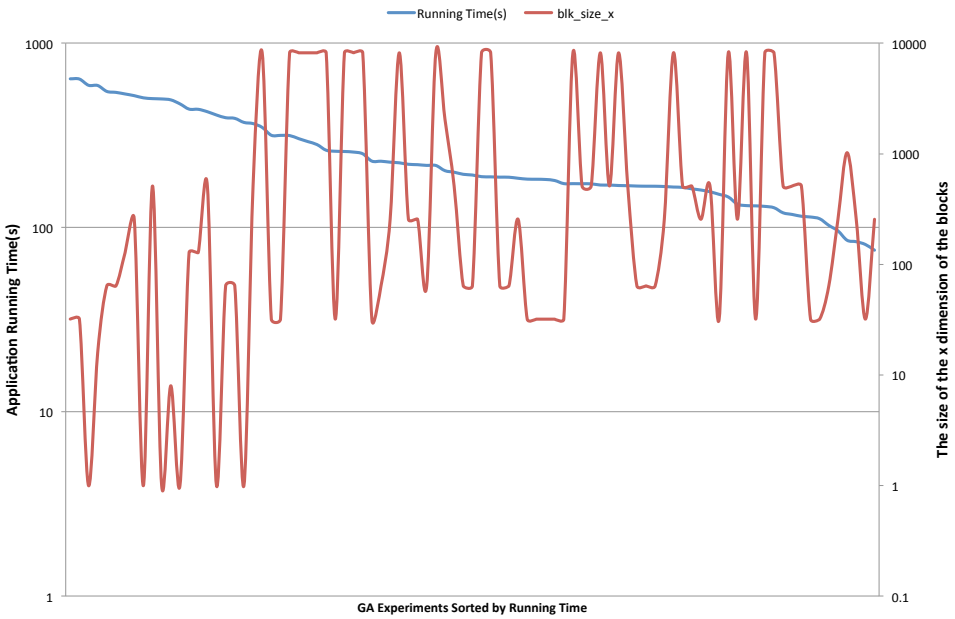

Figure 5.9: Evolution of the value of the first dimension of blocks with respect to the sorted running time. Block-Wise write benchmark with HDF5 chunked layout for a 16GB dataset on 128 processor cores

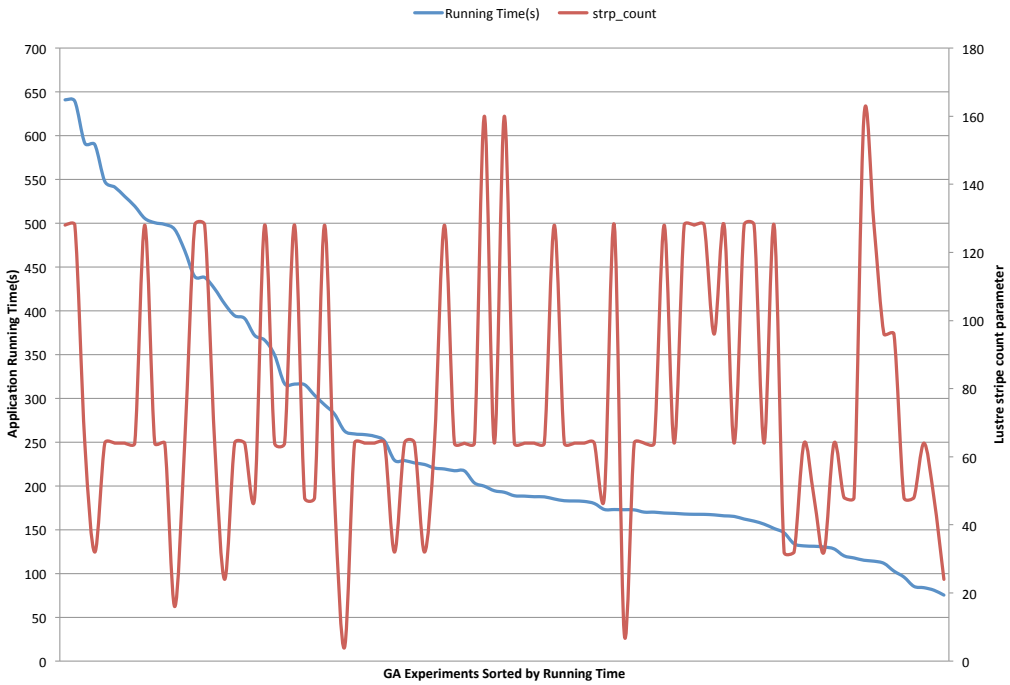

Figure 5.10: Sorted running time of the experiments with respect to the evolution of Lustre stripe count. Block-Wise write benchmark for a 16GB dataset on 128 processor cores

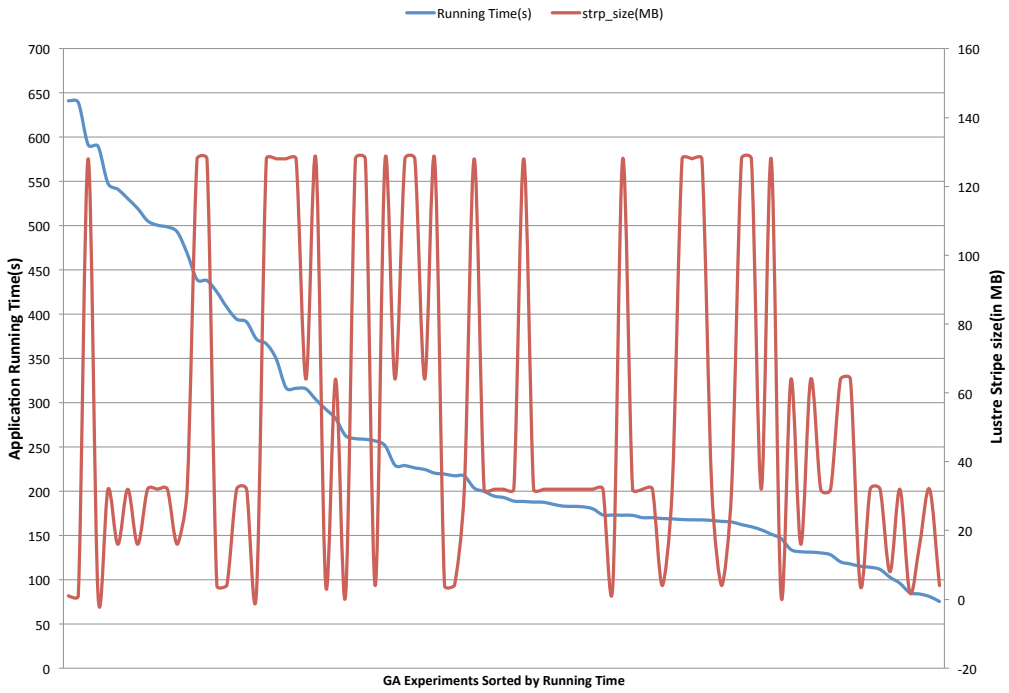

Figure 5.11: Sorted running time of the experiments with respect to the evolution of Lustre stripe size. Block-Wise write benchmark for a 16GB dataset on 128 processor cores

As it can be seen in these sets of plots, we could achieve the same performance as row-wise performance using HDF5 chunked layout in our library. This means that based on the application's characteristics and the way it needs to conduct I/O operations, pGIOLibrary can achieve near-optimal I/O performance. Proving the optimality of this I/O performance requires developing a model for I/O subsystem of an HPC platform which due to complex structures of current platform is a tedious task and we leave it for future work.

CHAPTER 6

CONCLUSION AND SUMMARY

As scientists are dealing with a data deluge, parallel computing has been playing a major role in taming this amount of data. One of the fields facing this problem is Geographic Information Science (GIScience). Two important challenges in dealing with large amount of data in GIScience has been addressed in this thesis: Determining the optimal configurations of parallel I/O stack and, Determining the size of data partitions in spatial domain decomposition.

Current I/O stack of a HPC supercomptuer has three levels: the file system (e.g. Lustre), middleware (e.g. MPI-I/O), and high-level data-organization layers (HDF5 and NetCDF), each of which has different parameters and configurations with a big impact on the performance of the I/O operations. An auto-tuning framework, using Genetic Algorithms has been developed to find the best performing configurations in this stack for each HPC platform and applications. This framework is applicable to any data-intensive application using parallel I/O on any HPC platform.

Conducting this auto-tuning not only on the I/O stack but at the application level addresses the second challenge which is more specific to data-intensive spatial analysis and GIScience. Having developed a simple parallel I/O library specific to data-intensive GIScience applications which abstracts the frequent I/O access patterns of these applications, related to the decomposition of these applications and using auto-tuning approach to tune the parameters of data partitions of each processing element conducting I/O, this research encourages the use of domain knowledge to increase the performance of different tasks in parallel computing.

The auto-tuning framework that we have developed contains two XML configuration files. One containing parallel I/O configurations including Lustre stripe factor, Lustre stripe unit, MPI-IO number of collective buffering nodes and MPI-IO collective buffering buffer size(and HDF5 chunking pa-

rameters, in case of parallel Block-wise I/O). The other configuration file is specific to the parallel I/O library and the write benchmark developed in which for each of the variables an I/O pattern and its configurations are specified. A dynamic library named H5Tuner reads in the first XML configuration at runtime and sets the parameters accordingly on the fly without a need of recompilation of the application. Summing up all of these parameters (Lustre settings, MPI-IO setting, HDF5 settings and the size of Row-Wise, Column-Wise or Block-Wise I/O) leads to a large space of configurations one need to tune for. Genetic Algorithms is used to find the optimal values for each of these parameters in a certain amount of time.

Several experiments in Chapter 5 done on a leading-edge HPC computer show the effectiveness of this approach in finding optimal parameters in different layers of parallel I/O and even at the application layer. The spatial parallel I/O library that has been developed in this research can now be used by GIScientists to conduct the I/O operations of their parallel application based on the way they have decomposed their problem and as the results show get the same optimal performance by using the I/O parameters that our auto-tuning framework specifies.

REFERENCES

[1] G. Bell, T. Hey, and A. Szalay, "Beyond the data deluge," *Science*, vol. 323, no. 5919, pp. 1297–1298, 2009. [Online]. Available: http://www.sciencemag.org/content/323/5919/1297.short

[2] T. Hey, S. Tansley, and K. Tolle, Eds., *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Redmond, Washington: Microsoft Research, 2009. [Online]. Available: http://research.microsoft.com/en-us/collaboration/fourthparadigm/

[3] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/o performance challenges at leadership scale," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654100 pp. 40:1–40:12.

[4] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp, "Parallel I/O prefetching using MPI file caching and I/O signatures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413415 pp. 44:1–44:12.

[5] A. Uselton, M. Howison, N. Wright, D. Skinner, N. Keen, J. Shalf, K. Karavanic, and L. Oliker, "Parallel i/o performance: From events to ensembles," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, april 2010, pp. 1 –11.

[6] Y. Yin, S. Byna, H. Song, X.-H. Sun, and R. Thakur, "Boosting Application-Specific Parallel I/O Optimization Using IOSIG," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, may 2012, pp. 196–203.

[7] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu, "Six degrees of scientific data: reading patterns for extreme scale science IO," in *Proceedings of the 20th international symposium on High performance distributed computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011.

[8] T. Zhao, V. March, S. Dong, and S. See, "Evaluation of a Performance Model of Lustre File System," in *ChinaGrid Conference (ChinaGrid), 2010 Fifth Annual*, july 2010, pp. 191 –196.

[9] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the hdf5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, ser. AD '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1966895.1966900 pp. 36–47.

[10] R. Rew and G. Davis, "Netcdf: an interface for scientific data access," *Computer Graphics and Applications, IEEE*, vol. 10, no. 4, pp. 76 –82, july 1990.

[11] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2009.5161054 pp. 1–12.

[12] A. Ching, K. Coloma, J. Li, and A. Choudhary, "High-performance techniques for parallel I/O," in *Handbook of Parallel Computing: Models, Algorithms, and Applications*. CRC Press, December 2007.

[13] F. Schmuck and R. Haskin, "Gpfs: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, ser. FAST '02. Berkeley, CA, USA: USENIX Association, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=1083323.1083349

[14] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in romio," in *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*. Washington, DC, USA: IEEE Computer Society, 1999.

[15] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface*. Cambridge, MA, USA: MIT Press, 1999.

[16] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in romio," in *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, ser. FRONTIERS '99. Washington, DC, USA: IEEE Computer Society, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=795668.796733 pp. 182–.

[17] GDAL Development Team, *GDAL - Geospatial Data Abstraction Library, Version x.x.x*, Open Source Geospatial Foundation, 201x. [Online]. Available: http://www.gdal.org

[18] S. Wang and M. P. Armstrong, "A theoretical approach to the use of cyberinfrastructure in geographical analysis," *International Journal of Geographical Information Science*, vol. 23, no. 2, pp. 169–193, 2009.

[19] Q. Guan and K. C. Clarke, "A general-purpose parallel raster processing programming library test application using a geographic cellular automata model," *Int. J. Geogr. Inf. Sci.*, vol. 24, no. 5, pp. 695–722, May 2010. [Online]. Available: http://dx.doi.org/10.1080/13658810902984228

[20] B. Behzad, J. Huchette, H. Luu, R. Aydt, S. Byna, M. Chaarawi, Prabhat, Q. Koziol, and Y. Yao, "Poster: Auto-tuning of parallel i/o parameters for hdf5 applications," SC'12, 2012.

[21] M. Sweet, "Mini-XML, a small XML parsing library," 2003-2011. [Online]. Available: http://www.easysw.com/ mike/mxml

[22] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[23] D. Whitley, "A genetic algorithm tutorial," *Statistics and Computing*, vol. 4, pp. 65–85, 1993.

[24] P. Pongcharoen, D. J. Stewardson, C. Hicks, and P. M. Braiden, "Applying designed experiments to optimize the performance of genetic algorithms used for scheduling complex products in the capital goods industry," *Journal of Applied Statistics*, vol. 28, no. 3-4, pp. 441–455, 2001. [Online]. Available: http://EconPapers.repec.org/RePEc:taf:japsta:v:28:y:2001:i:3-4:p:441-455

[25] C. S. Perone, "Pyevolve: a Python open-source framework for genetic algorithms," *SIGEVOlution*, vol. 4, no. 1, pp. 12–20, 2009. [Online]. Available: http://dx.doi.org/10.1145/1656395.1656397