5-2013

Improving the performance of the Parallel Ice Sheet Model on a large-scale, distributed supercomputer

Timothy J. Morey

Recommended Citation

Morey, Timothy J., "Improving the performance of the Parallel Ice Sheet Model on a large-scale, distributed supercomputer" (2013). *Electronic Theses and Dissertations*. 1895.
http://digitalcommons.library.umaine.edu/etd/1895

IMPROVING THE PERFORMANCE OF THE PARALLEL ICE SHEET MODEL ON A LARGE-SCALE, DISTRIBUTED SUPERCOMPUTER

By

Timothy J. Morey

B.A. Hendrix College, 2003

A THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science
(in Computer Science)

The Graduate School
The University of Maine
May 2013

Advisory Committee:

Philip Dickens, Associate Professor of Computer Science, Advisor

James Fastook, Professor of Computer Science

Bruce Segee, Associate Professor of Electrical and Computer Engineering

THESIS ACCEPTANCE STATEMENT

On behalf of the Graduate Committee for Timothy J. Morey, I affirm that this manuscript is the final and accepted thesis. Signatures of all committee members are on file with the Graduate School at the University of Maine, 42 Stodder Hall, Orono, Maine.

Philip Dickens, Associate Professor of Computer Science (Date)

LIBRARY RIGHTS STATEMENT

Signature:
Date:

IMPROVING THE PERFORMANCE OF THE PARALLEL ICE SHEET MODEL ON A LARGE-SCALE, DISTRIBUTED SUPERCOMPUTER

By Timothy J. Morey

Thesis Advisor: Dr. Phillip Dickens

An Abstract of the Thesis Presented
in Partial Fulfillment of the Requirements for the
Degree of Master of Science
(in Computer Science)
May 2013

In this thesis, we describe our work to understand and improve the performance and scalability of the Parallel Ice Sheet Model (PISM) on the Ranger supercomputer. PISM enables the simulation of large-scale ice sheets, such as those found in Greenland and Antarctica, which are of particular interest to climate scientists due to their potential to contribute to sea-level rise.

PISM has a unique parallel architecture that is designed to take advantage of the computational resources available on state-of-the-art supercomputers. The problem, however, is that even though PISM can run without modification on a supercomputer, it is generally unable to do so efficiently. We observed that PISM exhibits rapidly diminishing performance gains as the number of processors is increased, even experiencing an increase in execution time with large processor counts. PISM's inability to make efficient use of the resources available on today's supercomputers presents a challenge to researchers, particularly as larger and higher resolution data sets become available. In this work, we analyzed the reasons for PISM's poor performance and developed techniques to address these issues, resulting in an increase in performance by as much as a factor of 20.

IMPROVING THE PERFORMANCE OF THE PARALLEL ICE SHEET MODEL ON A LARGE-SCALE, DISTRIBUTED SUPERCOMPUTER

By Timothy J. Morey

Thesis Advisor: Dr. Phillip Dickens

A Lay Abstract of the Thesis Presented
in Partial Fulfillment of the Requirements for the
Degree of Master of Science
(in Computer Science)
May 2013

Keywords: high performance computing, parallel i/o, ice sheet model, scalability

In this thesis, we describe our work to understand and improve the performance and scalability of the Parallel Ice Sheet Model (PISM) on the Ranger supercomputer. PISM is a software system and research tool for the study of large-scale ice sheets, such as those found in Greenland and Antarctica, which are of particular interest to climate scientists due to their potential to contribute to sea-level rise.

We have found that PISM, in its default configuration, is not able to make efficient use of supercomputer resources. We evaluate the performance of various components of the software, and both find and explain the reasons for the poor performance. We then introduce several changes to the software that result in significant performance improvements. In particular, we modify the components that read input data and write output data to make better use of the unique hardware and software available on Ranger, and we achieve up to a 20-fold performance improvement.

ACKNOWLEDGEMENTS

This material is based on the work supported by the National Science Foundation under Grant No. FY2013-030.

We acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. URL: http://www.tacc.utexas.edu.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

The Parallel Ice Sheet Model (PISM) [29, 6, 39], developed by researchers at the University of Alaska Fairbanks and the Potsdam Institute for Climate Impact, enables the simulation of large-scale ice sheets, such as those found in Greenland and Antarctica. It is one of the few ice sheet models designed to take advantage of the computational resources available on state-of-the-art supercomputing systems, by allowing the simulation to be distributed over many processors on many machines. The distributed approach can improve the computational performance of the model by making use of multiple processors to perform computations in parallel, and it also allows for the execution of larger and more detailed simulations that would not fit within the memory of a single machine.

The problem, however, is that even though PISM can run without modification on a supercomputer, it is generally unable to do so efficiently. The current implementation of PISM exhibits rapidly diminishing performance gains as the number of processors is increased. In fact, we observed that in some situations, the overall execution time actually increases as we add more computational resources. PISM's inability to make efficient use of the resources available on today's supercomputing systems is problematic, particularly as larger and higher resolution data sets become available.

The ability of a simulation system to efficiently scale to large problem sizes is an important issue for a number of reasons. First, if a system is unable to scale to large problem sizes, it will be unable to handle new high-resolution data sets, such as those being created by the Center for Remote Sensing of Ice Sheets (CReSIS) [13]. Such datasets are of interest to researchers, since they have the potential to reveal new information about the evolution of ice sheets, and they may lead to new insights. Second, a system that scales poorly will waste the researcher's valuable time. Large simulations may result in very long wait times, which slows the pace of experimentation. In addition to wasting the time of researchers, an inefficient simulation will also waste the researcher's resources.

Users of supercomputers are typically charged according to the computational resources they consume, and a long running simulation can incur a significant cost.

The efficiency is also of interest to computer scientists, as PISM provides a unique platform in which current high performance computing (HPC) techniques can be evaluated and new approaches can be implemented and tested. PISM currently uses many of the latest HPC technologies, including the PETSc [3, 2, 4], which provides a computational and mathematical framework, as well as NetCDF [38] and Parallel-NetCDF [28, 21], which provide high performance file access. These libraries are designed to abstract away the details of HPC and produce software that is easy to use and will run consistently across a wide range of systems. Indeed, the goal is to encapsulate the best known practices and latest research in HPC, and make them available to the larger research community. However, the performance of the technologies used in PISM is, in many cases, quite poor, indicating the need for more research before the power of the model can be fully realized.

This thesis describes our work in determining the root causes of PISM's poor scaling characteristics and developing solutions to address them. We investigated the scalability of both the computational and the input and output (I/O) performance of the simulation, varying the computational resources and model sizes. In this, we found that while the computation scales remarkably well, the I/O operations impose a significant performance penalty, which may dominate the overall system performance.

Many of the experiments and exercises we describe could have alternatively been performed with the use of synthetic benchmark applications. Despite the comparative complexity, we decided to work with a large-scale and widely-used simulation for two reasons. First, benchmarks may not capture some of the subtle performance details exhibited by more complex applications, and second, benchmarks may exaggerate the benefit of certain techniques. However, we still use benchmarks in some cases to complement and help explain results that we observed in PISM.

This thesis is organized as follows. In Chapter 2, we will provide some background information about the design of PISM, with particular emphasis on its parallel architecture and its approach to file I/O. Chapter 3 provides a survey of related work. In Chapter 4, we will discuss the software test cases, hardware systems, and measurement techniques used in this work. We then gather baseline data to analyze the performance and scalability issues in Chapter 5. Chapters 6 and 7 describe our work in improving and understanding the scalability characteristics of PISM. Chapter 8 provides a summary of our findings and a discussion of future work.

CHAPTER 2
BACKGROUND

In this chapter, we provide an overview of ice sheet modeling in general and PISM's approach to the problem in particular. We also describe the parallel architecture of the system and the I/O system that it uses to read and write file data.

2.1 Ice Sheet Modeling

Ice sheets have been shown to have a dramatic influence on the world's climate, and they contribute to the rise of sea-level when they melt. This makes it critical to understand the dynamics of ice sheets. However, the latest report from the Intergovernmental Panel on Climate Change cited a lack of understanding about ice sheet dynamics as a major obstacle to predicting the magnitude of future sea-level rise [18].

Developing insight into the behavior of ice sheets presents a number of challenges to domain scientists. Collecting empirical data about ice sheets is very difficult since they exist in remote locations and inhospitable climates. Furthermore, many portions of the ice sheet, such as the interior and the base where the ice meets the bedrock, are only accessible through remote sensing technologies. Another obstacle is that ice sheets move very slowly, and it may be on the order of years or centuries before an ice sheet exhibits an observable response to a given stimulus. For these reasons, the primary tool available to researchers is the computer simulation, in which millennia of ice sheet evolution can be simulated in a matter of hours.

Ice sheet models combine physical flow laws, such as the Shallow Ice Approximation (SIA), the Shallow Shelf Approximation (SSA), and the Navier-Stokes equations [39], with observations about the current and historical state of the ice sheet. These observations include measurements of the bedrock topography, ice surface geometry, and ice thickness, which are gathered with remote sensing techniques. The Sea-level Response to Ice Sheet Evolution (SeaRISE) project, which is a community organized project that aims to estimate the contributions of ice sheets to sea-level, has aggregated many of these observations into standardized datasets for the Greenland and Antarctic ice sheets [5, 33].

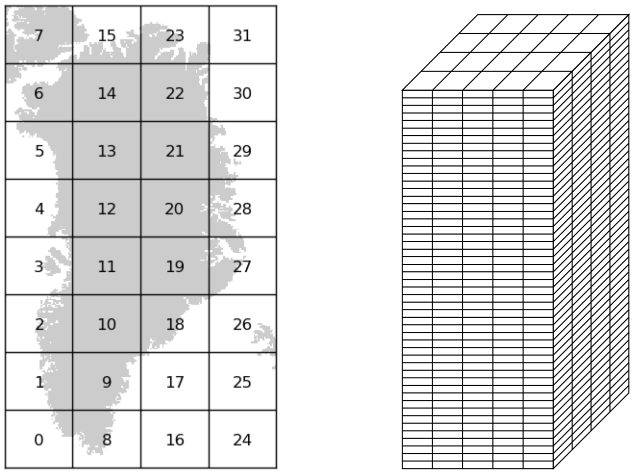
2.2 PISM

The Parallel Ice Sheet Model (PISM) [29, 6, 39] is one of several open source ice sheet models available today [12, 19, 34], but it is one of a few that has a parallel architecture designed to scale to large problem sizes [11, 32]. This, in addition to offering multiple modeling techniques (e.g. SIA, SSA, and a hybrid thereof [6]) makes PISM a valuable tool for researchers.

PISM gains much of its parallel structure from the Portable Extensible Toolkit for Scientific Computation (PETSc), which is a popular library of data structures and routines for numerically solving partial differential equations. [3, 2, 4] PISM makes use of many of PETSc's distributed data structures, and allows PETSc to manage much of the communication between the processes participating in a simulation. PISM also relies upon PETSc's algebraic solvers to assist in the finite difference calculations that drive the simulation.

Within PISM, the model takes place in a rectangular computational box that consists of a collection data points in three dimensions. This box represents the space that encloses a glacier, and in the case that the model is based on a real-world glacier, each of these data points may be mapped to a unique geographical location in or near that glacier. Since most glaciers exist in polar regions, it is typical to express the coordinates of these points in a projected coordinate reference system based on the polar stereographic projection. While the points may not describe a regular rectangular space in the real world, they do define a regular rectangular space in their native coordinate reference system.

In each of the x and y dimensions, the grid points are equally spaced, and have a relatively coarse resolution. The z dimension is given a relatively finer resolution than x or y, and the spacing of grid points along this dimension may vary within a given model, which allows for more detail near the base of the ice where driving forces are greatest. Each x, y pair represents a single column of ice that is parallel with the force of gravity. The structure of the grid is shown in Figure 2.1



(a) (b)

Figure 2.1. The Distribution of Data Points among Processes in PISM. In (a), we see how regions of the computational box are mapped to process, identified by rank. Each process owns a rectangular subgrid of the computational box that is structured like figure (b), with the top face of the subgrid in (b) corresponding to one of the rectangles in (a). Each space within the grid in (b) has many associated data values, such as temperature, which describe the current state of a block of ice, ocean, atmosphere, or ground that occupies that space.

The form of parallelism used in the Parallel Ice Sheet Model is process parallelism, where independent processes, possibly on separate machines, communicate to solve problems. PISM uses the single-program, multiple-data (SPMD) paradigm, where each process stores a region of the model's computational box its local memory, and all processes perform the same computations on their local data. While a given process may need to read data values that are stored at another process, each process only modifies the data values within its local data. An example of how processes may be assigned to regions of the model is given in Figure 2.1.

When a simulation is started using n processes, PISM uses an algorithm that attempts to make the n rectangular subgrids as square as possible in the x and y dimensions, with respect to the number of data points. However, for many computational box sizes and values of n, there is no way to evenly distribute the grid points to n non-intersecting squares. In such cases, the subgrids will be arranged into r rows and c columns with $n = rc$, with no row being more than one grid point taller than any other row, and no column being more than one grid point wider than any other column. Squareness is desired,

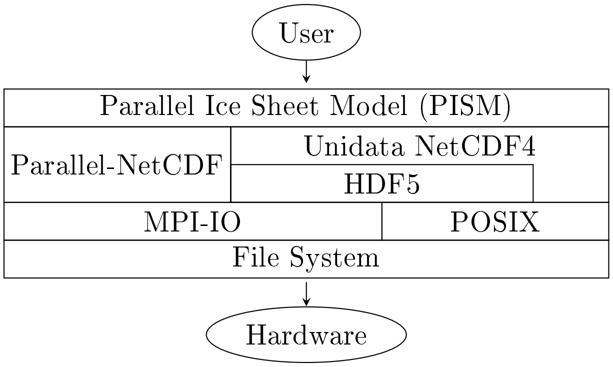


Figure 2.2. The Layers of Software in PISM's File I/O System. Each component communicates with those components directly above and below, and each vertical line that can be drawn through the libraries represents a possible I/O path for PISM. Not all of these I/O paths are actively supported, as it is not currently possible to use the HDF5 library's POSIX driver without code modifications.

because when calculating the new value of a variable at a single grid point, it is often only necessary to know the current values of variables at adjacent grid points. Therefore, the degree to which one process must communicate with other processes is proportional to the perimeter of local grid, and the square has the minimum perimeter for any rectangle of area n.

2.3 PISM I/O

In an abstract sense, PISM is a simple data processing program, which reads input data from a file, performs calculations with the data, and writes the results to a file. Most of the data that PISM reads and writes consists of multidimensional arrays of numerical data that we refer to as *variables*. These variables store quantities for the grid points in PISM's computational box, so the size and shape of the arrays correspond to the size and shape of the computational box. Some variables are three-dimensional, giving a value for each data point in the computational box, while many others are two-dimensional, giving a value for each x, y point. Some variables evolve over the course of the simulation, and therefore extend along the *time* dimension as well as the spatial dimensions. We refer to these as *record variables*, and they may be written to file at each time step to produce a record of the evolution of the variable.

Mode	Library	File Format(s)	Parallel?	Large Data?
NetCDF4/HDF	NetCDF4	HDF	yes	yes
NetCDF4/CDF	NetCDF4	CDF{1,2}	no	no
PNetCDF/Default	PNetCDF	CDF{1,2}	yes	no
PNetCDF/CDF5	PNetCDF	CDF5	yes	yes

Table 2.1. File I/O Modes. The name given in the 'Mode' column is the term we will use to identify each I/O mode.

There are a number of variables used in PISM. For example, the *enthalpy* variable is a three-dimensional variable that extends along the x, y, and z dimensions, and it stores a measurement of the enthalpy field for each grid point in the computational domain. In contrast, the *thk* variable is a two-dimensional variable that extends along the x and y dimensions, and it stores a measurement of the ice thickness at each x, y point in the computational grid. Both *enthalpy* and *thk*, are record variables and therefore evolve over time as the simulation progresses. In contrast, the *bheatflx* variable, which gives the geothermal flux at the bedrock, is not a record variable and therefore does not evolve over the course of a particular simulation.

Given that PISM computes a number of variables, some of which can be quite large, and that such variables may be written to disk at each time step, it is critical to have a highly efficient I/O system. As of version 0.5, PISM provided three techniques for file access, each of which has its own performance characteristics. These techniques combine two I/O libraries, Unidata's NetCDF (NetCDF4) [38] and Argonne's Parallel-NetCDF (PNetCDF) [28, 21]. These libraries can be used in two I/O modes: serial and parallel. The first mode, *serial* access, is a widely used strategy where only a single process is allowed to modify a file at any given time. The POSIX standard defines serial file access operations, and these may be used through NetCDF4. The second strategy is *parallel* access, which allows multiple processes to cooperatively access a file and modify it in parallel.

The ability to perform parallel I/O operations is based on MPI-IO [25], which defines a rich set of operations with well-defined semantics. Both NetCDF4 and PNetCDF implement their parallel I/O support through the use of MPI-IO. Figure 2.2 shows the software stack used for I/O in PISM, and Figure 2.1 summarizes the three I/O techniques supported

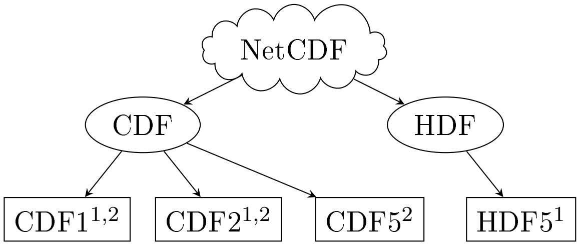


Figure 2.3. The NetCDF Family of File Formats. The NetCDF family contains two sub families corresponding to two internal file structures: the Common Data Format (CDF) and the Hierarchical Data Format (HDF). Within these two subfamililes, there are four specific file formats, which are shown in the rectangles at the bottom of the tree. The superscripts on the file formats indicate if the format is supported in the NetCDF4 library (1) and the PNetCDF library (2).

by PISM (NetCDF4/HDF, NetCDF4/CDF, and PNetCDF/Default), as well as a fourth technique that was implemented for this thesis (PNetCDF/CDF5). It is important to note that in some cases the various I/O libraries use different file formats.

2.3.1 File Formats

Ostensibly, PISM is said to read and write "NetCDF files" [30], and its input and output files are typically labeled with the `.nc` file extension. However, there are four specific formats that use two significantly different structures within the family. Figure 2.3 shows the relationship between the file formats in the NetCDF family.

The Common Data Format (CDF) is the traditional file structure associated with NetCDF files, and it has evolved over time. The Hierarchical Data Format (HDF) is a recent addition to the NetCDF family, which was introduced with version 4 of Unidata's NetCDF library. All of these file formats are said to be *self-describing*, which means that they contain metadata that describes the structure and meaning of the file data.

The structure imposed by the Common Data Format is shown in Figure 2.4. A single header block, stored at the beginning of the file, contains all of the metadata for the file. This metadata includes definitions for each variable, as well as attribute data that provides additional context for the data. The header is followed by a section that contains all non-record variables, also known as *fixed-size variables*. These are variables for which the required storage space is known when the variable is defined. The third and final section

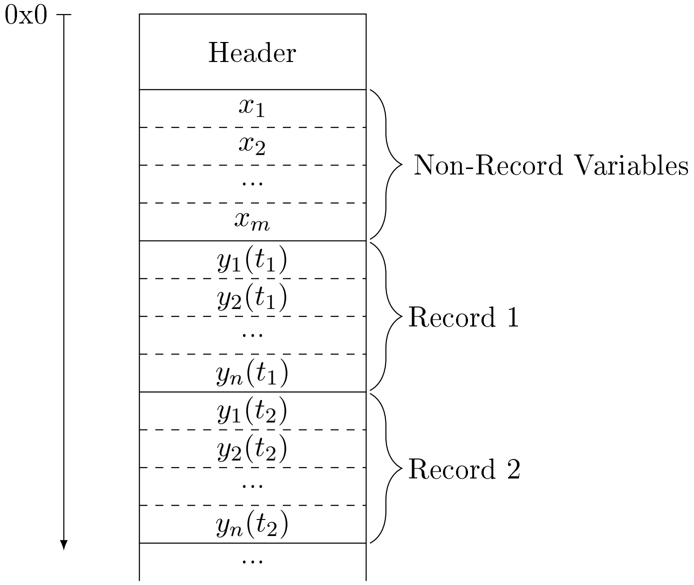


Figure 2.4. The CDF File Structure. The CDF file structure is shared by the CDF1, CDF2, and CDF5 file formats. The structure is characterized by three disjoint regions: the header section, the non-record section, and the record section. Above, $x_1, x_2, ..., x_m$ are fixed-size variables, and $y_1, y_2, ..., y_n$ are record variables, with m and n as positive integers.

in the file is the record section, which is used to store record variables. These variables are allowed to grow along one dimension, which is designated the *unlimited dimension*, and which differs from the other dimensions in that its size does not need to be specified when the dimension is defined.

In PISM, and in many other scientific applications, the unlimited dimension is the *time* dimension, while the spatial dimensions, such as x, y, and z, are fixed-size dimensions. This configuration makes it easy to append new values of the variables as they are updated in the simulation.

The first version of the Common Data Format, CDF1, is also called the *classic format.* Its primary limitation is that it uses 32-bit offsets to express locations in the file, thus limiting the practical file size to 4GB. The CDF2 revision partially corrected this shortfall by using 64-bit offsets for records, thus allowing for significantly larger files. However, CDF2 still limited the size of a single record for a single variable to 4GB. The CDF5 format relaxes this restriction by using 64-bit offsets for variables too, which allows PISM to work with large datasets.

The Hierarchical Data Format (HDF) is a significantly more complex format, offering features that are a superset of those offered by CDF. In particular, HDF allows for the use of multiple unlimited dimensions, and it allows datasets to be hierarchically organized [15]. It is worth noting that PISM does not benefit from either of these extensions, as its data fits very conveniently into the CDF structure. While there have been revisions to HDF, much like there have been revisions to CDF, we are only considering the latest revision, which is the version supported by the HDF5 library [14].

The diversity in the NetCDF family of formats presents a challenges to PISM's users, since the formats are not universally supported by the tools utilized for pre and post-processing of datasets. The CDF1 and CDF2 file formats are widely supported, due to their maturity. The HDF format also has wide support, due to the popularity of the HDF5 library and the flexibility of the format. The CDF5 format, however, has very limited support, primarily because Argonne's Parallel-NetCDF is currently the only library that supports the format. Support is planned for future versions of Unidata's NetCDF library, but at the time of writing, this functionality has not been implemented.

2.3.2 Serial File Access with NetCDF4 (NetCDF4/CDF)

Until recently, PISM only supported serial file access using Unidata's NetCDF library [38]. We refer to this I/O technique as NetCDF4/CDF throughout this document. With this technique (illustrated in Figure 2.5), a single process, P_1, performs all of the file access operations for the simulation. Each PISM process, P_i, needs to read and write data for its local subgrid and must communicate with P_1 to do this. When reading data, P_1 first reads the data for each P_i and then sends the data to P_i. Similarly, when writing data, each P_i sends its data to P_1, which then writes the data to file. The primary advantage of this technique is its ubiquity, as it is a technique that is supported on many systems, and allows the same code to run on a laptop and on a supercomputer. It is also a technique

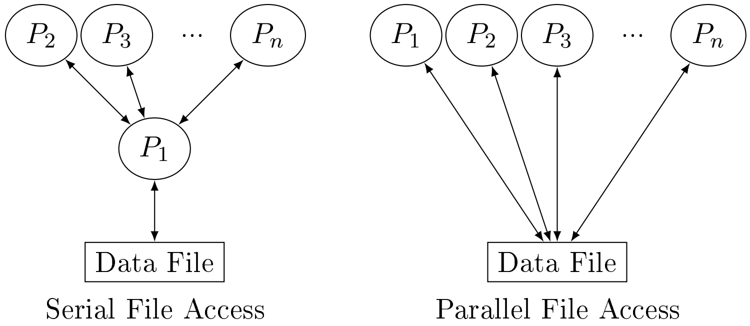


Figure 2.5. The Serial and Parallel File Access Patterns. In each example, n PISM processes, labeled P_1, P_2, P_3, ..., P_n, access a single shared data file. In the serial access mode, each process must access the file through P_1, which can complicate the I/O code and create a performance bottleneck at P_1. In the parallel access mode, each process can directly read from and write to the shared data file.

that is familiar to many experienced application developers, and as we shall see, it can offer reasonable performance in some limited cases.

Since the NetCDF4 library is used for this I/O technique, CDF1, CDF2, and HDF files can be read and written. However, when creating a new file, PISM will always default to CDF2.

2.3.3 Parallel File Access with NetCDF4 (NetCDF4/HDF)

Reading and writing datasets serially is very expensive. For this reason, PISM recently introduced support for parallel I/O, while maintaining the legacy serial access described above. The two file access patterns are illustrated in Figure 2.5. The latest versions of Unidata's NetCDF library may be used for parallel file access, starting at version 4.0 [38]. By this method, each process reads and writes the data values that are within that process' local subgrid, with the goal of improving performance and evenly distributing the I/O workload between the processes. This also has the advantage of simplifying the file access code, as there need not be separate cases for P_1 and all of the other processes as needed when using serial I/O. We will abbreviate this technique, which uses the NetCDF4 library to perform parallel I/O, as NetCDF4/HDF. It is important to note that NetCDF4 inherits its parallel functionality from the HDF5 library [14], and therefore when performing a parallel write, the data must be written to an HDF file.

2.3.4 Parallel File Access with PNetCDF (PNetCDF/Default)

The Parallel-NetCDF library (PNetCDF) [21, 28] was developed by researchers at Argonne National Laboratory, and was the first library to offer parallel I/O support for NetCDF files. However, PNetCDF provides parallel support for CDF files rather than HDF files.

There are a number of reasons that PNetCDF has not been widely adopted for scientific applications. One problem is that when PNetCDF was developed, Unidata's NetCDF library was already established as the standard I/O library for NetCDF data files. Since, PNetCDF exposes a significantly different programming interface than NetCDF, switching from NetCDF to PNetCDF is a nontrivial task for domain scientists since it requires significant development and testing. Another obstacle is that, until recently, PNetCDF provided limited support for large data sets. The addition of CDF5 in recent versions of PNetCDF provides the large-data support, but the limited application support for CDF5 remains an obstacle.

2.3.5 Parallel Large-File Access with PNetCDF (PNetCDF/CDF5)

The fourth and final I/O mode allows for parallel access to CDF5 files with the PNetCDF library, which we will refer to as PNetCDF/CDF5. We implemented this technique for this work, and it is not available in PISM 0.5. We will discuss this mode in detail in Chapter 6.

CHAPTER 3
RELATED WORK

In this thesis, we are attempting to explain and improve the scalability of the Parallel Ice Sheet Model (PISM). As will be seen, much of this work involves parallel I/O techniques, so we will focus on related work in this area. A great deal of work has been done to understand and address the I/O requirements for large-scale, parallel, scientific applications, and we will describe some of it in this chapter. We will describe the work to discover the I/O pattern that is common to PISM and many other scientific applications, as well as the parallel I/O paradigm that was developed to efficiently support this pattern. We will then describe some of the I/O libraries that were developed to support parallel I/O, and we will look at some evaluations of their performance. We will conclude by reviewing some work to understand why these parallel I/O techniques perform poorly with the Lustre filesystem and some approaches that have been taken to address these problems.

It has been well established that the I/O requirements of large-scale scientific applications, such as PISM, present a significant performance bottleneck [8, 10, 27]. The primary reason is that such applications often perform a large number of small I/O operations, and I/O operations can be quite expensive. PISM exhibits this I/O pattern, due to the parallel nature of the simulation. The problem stems from a mismatch between the way data is stored in memory and the way it is stored on disk. In PISM, processes generally access their data in terms of multidimensional arrays, while it is stored on disk as a linear sequence of bytes. An example may help clarify this.

Consider Figure 3.1, which shows how a two-dimensional variable is stored on disk, and how it is stored in memory when distributed among four processes named P_0, P_1, P_2, and P_3. Suppose that P_0 needs to write its data to disk. It must first write blocks 0 and 1, and then it must seek to a new location and write blocks 4 and 5. The other processes each have to perform similar operations to write their data, requiring a total of 8 write operations and and additional 4 seek operations.

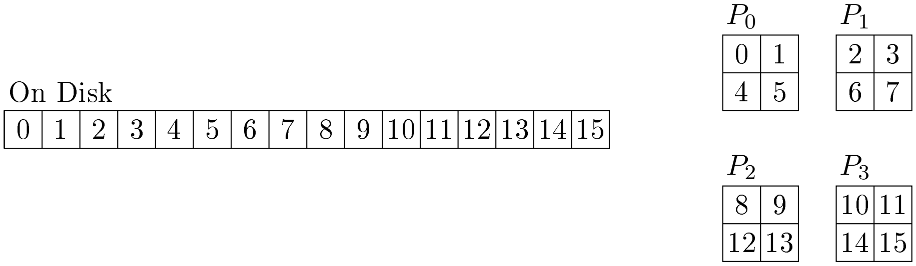


Figure 3.1. The Different Representations of Array Data On Disk and In Memory. In this figure, we see how a single two dimensional variable is stored on disk and in memory when distributed among four processes named P_0, P_1, P_2, and P_3. On the left, we see that the variable is represented as a one-dimensional array on disk. On the right, we see that the variable is divided between the four processes and represented as a two-dimensional array at each process.

The MPI-IO specification [25], which is an extension of the MPI standard [23], was developed to address the performance issues caused by operations described above. It defines *collective* I/O operations, which must be performed by all processes at the same time, as well as *independent* operations, which may be performed by an individual process. The collective operations allow a group processes to open a file collaboratively and cooperate in read and write operations. Having knowledge of the aggregate I/O operations of all processes allows the the MPI-IO implementation to optimize the accesses to the underlying filesystem.

It is important to note that MPI-IO is only a specification of how parallel I/O operations will behave, and it does not provide an implementation for any of the operations. ROMIO [36] is a widely used implementation of the MPI-IO standard, and it was developed at Argonne National Laboratory. In addition to implementing the MPI-IO operations, it implements a number of optimizations for these operations, including *data-sieving* and *two-phase I/O* [37]. Both of these optimizations attempt to coalesce many small I/O operations into fewer large contiguous I/O operations, and we will discuss each in turn.

Data-sieving is the process of combining multiple I/O operations into a single operation that accesses a contiguous region of memory. For example, consider a data-sieving independent read operation for P_0 in Figure 3.1. Rather than performing one read operation for blocks 0 and 1, a seek, and a separate read operation for 4 and 5, a data-sieving read would read blocks 0, 1, 2, 3, 4, and 5 into an intermediate buffer with a single I/O operation. Blocks 0, 1, 4, and 5 would then be copied into P_0's memory, and blocks 2 and 3 would be discarded. This provides improved performance because the cost of performing three separate I/O operations is greater than the cost of a single I/O operation plus an in-memory copy.

Two-phase I/O is a collective I/O technique that takes advantage of the global knowledge of the aggregate I/O operations for a number of processes to improve performance. In the case of a read operation, each process reads a contiguous block from the file in a single I/O operation and then redistributes the data to other processes using inter-process communication. In the case of a write operation, the processes first exchange the data such that each process can perform a single contiguous I/O operation, and then the processes write the data to file. For example, consider a two-phase collective write involving the processes in Figure 3.1. In the first phase of the collective write, the processes would exchange data such that P_0 has blocks 0, 1, 2, and 3, process P_1 has blocks 4, 5, 6, and 7, and so forth. Each process would then perform a single I/O operation to write the contiguous blocks of data to file. This improves performance because it is orders of magnitude less expensive to perform inter-process communication than it is to perform multiple I/O operations.

PISM, and other scientific applications, generally use MPI-IO operations indirectly, through the use of high-level I/O libraries like PNetCDF [21], NetCDF4 [38], and HDF5 [14]. When using MPI-IO directly, the application developer needs to know the structure of the file, but the high-level libraries hide the file structure from the developer and allow him to focus on the application's data model. The PNetCDF, NetCDF4, and HDF5 libraries each translate between the application data model and the file data model, and also provide a portable standardized file format.

The Adaptable Input/Output System (ADIOS) [1] is another abstract I/O library that is built upon MPI-IO, and seeks to simplify and optimize I/O for parallel applications. As with PNetCDF, NetCDF4, and HDF5, ADIOS helps translate between the application data model and the file data model. Additionally, it allows the I/O behavior of applications to be changed without having to modify the application code. This feature is implemented through the use of an XML configuration file that controls the application's I/O behavior at runtime. ADIOS also introduced a new self-describing file format that is optimized for parallel writes.

While we do not develop a new file format for this research, we do augment PISM's I/O capabilities through the use of the CDF5 file format, which is supported through PNetCDF for large data sets. As with ADIOS' format, CDF5 offers exceptional write performance. In contrast, the ADIOS file format is considerably more complex than the CDF5 format we use.

During the course of our research, we have done a great deal of experimental evaluation of the I/O libraries. Such comparisons have also been performed by other researchers. In [21], the developers of PNetCDF evaluated the performance of the new library using a pair of benchmark applications. In one test, they established that parallel write performance through PNetCDF consistently performed better than serial write performance with NetCDF, which at that time only offered serial I/O. They also showed that PNetCDF performs better than parallel HDF5 using the FLASH I/O benchmark [40], and they attribute this performance advantage to the relative simplicity of PNetCDF compared to HDF5. There is a trade-off, however, because with the added complexity, HDF5 provides a more powerful data model.

The results we present in Chapters 5 and 6 show a similar performance advantage for PNetCDF compared to HDF5. There are two primary differences between these experimental studies. Most importantly, their results were based on benchmarks rather than a widely-used scientific application. Also, their results were obtained using early versions of the libraries, at versions 0.8.4 and 1.4.5 for PNetCDF and HDF5, respectively. In contrast, we are using versions 1.3.1 and 1.8.9, respectively, in our tests.

The parallel I/O approaches described above perform well across many different filesystems. However, it has been observed that many of the optimization techniques used in these libraries result in poor performance on the Lustre filesystem. In [9], Dickens and Logan found that the assumption that better performance can be achieved by performing few large I/O operations, rather than many smaller operations, does not hold true for Lustre filesystems. Rather, they found that it was the number of Object Storage Target (OST) accessed by each process that dominates the I/O performance. Similar results have also been reported by Liao and Choudhary [20] and Coloma et al. [7]. We discuss the issues associated with the Lustre filesystem in Chapter 7.

In [16], Howison et al. applied the findings of Dickens and Logan to optimize HDF5 parallel collective write operations for the Lustre filesystem. They used tuning parameters available in the HDF5 library, including chunking and block alignment [15], to ensure that write operations were assigned with Lustre stripes. They also used a Lustre-optimized collective buffering algorithm provided with Cray's Message Passing Toolkit. They found that they were able to achieve significantly better performance with this approach, as opposed to default performance of HDF5 on Lustre, for collective write operations involving as many as 40,960 processes.

In contrast to the work by Howison et al., the work we describe in Chapter 7 applies the findings of Dickens and Logan to the PNetCDF library, rather than HDF5. Furthermore, instead of using Cray's implementation of MPI and MPI-IO, we use the MVAPICH2 implementation. Since MVAPICH2 does not include Cray's Lustre-optimized collective buffering algorithm, we implemented our own approach.

CHAPTER 4
EXPERIMENTAL DESIGN

In this chapter, we describe the software test cases we are using and how these tests were designed to focus on different aspects of PISM's computational performance. We also describe the two hardware systems we used to run these tests, and we describe how we measure performance and present our results.

4.1 Test Cases

We have designed our test cases to measure both the computational and the I/O performance of PISM. The execution of PISM can be broken into three stages: initialization, computation, and output. During the initialization stage, PISM reads its input data from file and builds up its internal data structures. The computational phase consists of discrete time steps, and it is in this stage that the majority of the numerical calculations are performed. During the output stage, PISM writes the final model state to file. By measuring the time it takes to perform the initialization and output stages, we can measure the I/O performance, and by measuring the time it takes to perform a single time step in the computation stage, we can quantify the computational performance.

We are focusing on four PISM test cases that simulate the two largest ice sheets on the planet at two different resolutions each. These tests are all based upon examples that are distributed with PISM [30], which are based on the SeaRISE datasets [33]. Figure 4.1 summarizes the statistics for the various test cases.

4.1.1 Greenland (G1km and G5km)

The first model that we will be using is a model of the Greenland ice sheet. While this data can be used to model the ice sheet at a variety of resolutions, we are focusing on the one and five kilometer resolutions, which we will refer to as the G1km and G5km models, respectively.

Model	x	y	z	Grid Points	File Size	Large File?
G1km	1,501	2,801	401	1,685,924,701	28 GB	yes
A5km	1,200	1,200	301	433,440,000	7.2 GB	yes
A10km	600	600	301	108,360,000	1.8 GB	no
G5km	301	561	201	33,941,061	1.1 GB	no

Table 4.1. Model Statistics. The table shows some statistics about each model we are using, including the number of data points in each spatial dimension (x, y, and z), the total number of data points in the computational grid, an approximation of the size of the output files produced by the models, and whether the model requires large file support, via the HDF or CDF5 formats.

The G5km model consists of 301 grid points in the x dimension, 561 in the y dimension, and 201 in the z dimension, for a total of just over 33 million total grid points. In contrast, the G1km model consists of 1,501 grid points in the x dimension, 2,801 in the y dimension, and 401 in the z dimension, for a total of more than 1.6 billion grid points. A single three-dimensional variable, which consists of a double-precision floating point value at each grid point, requires about 271 MB of file storage space for G5km and about 13 GB of storage space for G1km. When loaded into memory, these variables require slightly more storage space, since some data points must be duplicated across processes. A typical output file, which contains multiple one, two, and three-dimensional variables and all of the data necessary to restart the model, will be about 1.1 GB for G5km, and about 28 GB for G1km.

Due to the large number of data points in the G1km model, the CDF1 and CDF2 formats cannot be used to store the input and output for the simulation. As discussed in Section 2.3.1, these formats use 32-bit offsets for variables, and thus limit the size of a single variable to 4GB. Only the CDF5 and HDF formats, which use 64-bit offsets, may be used at this resolution. The size of G1km also presents problems at run-time, as the model requires more than a terabyte of memory when the simulation is running. Few workstations provide this amount of memory, so it is necessary to use a distributed supercomputer at this resolution.

As mentioned previously, the source data for the models was provided by the SeaRISE project, but this data was modified significantly to prepare it for our test runs. The data had to be preprocessed to create a PISM input file representing the present state of the ice sheet. The stable-0.5 distribution of PISM provides example scripts to perform the necessary preprocessing computations, which is described in [30].

The test runs we performed use the preprocessed data as input for a predictive simulation. There is a very large parameter space available to control different aspects of the simulation, and we used those provided in PISM's example scripts [30]. The only modification that we made was to shorten the simulation so that it only performs a single time step.

4.1.2 Antarctica (A5km and A10km)

We also used data sets representing the Antarctic ice-sheet at 5 kilometer and 10 kilometer resolutions, which we will refer to as A10km and A5km, respectively. As with the G1km and G5km models, the Antarctica models were based upon SeaRISE data [33], and was produced through the example preprocessing scripts distributed with PISM. The parameters of our test runs were again based upon the example scripts provided by PISM. As shown in Figure 4.1, the size of the A10km and A5km models fall between the G1km and G5km models.

The A10km model provides a nice complement to the other models in several respects. First, it represents nearly the largest model that one can fit into the CDF1 and CDF2 formats, which allows us to compare all I/O techniques. Furthermore, the computational boxes for A10km and A5km are square and have evenly divisible dimension sizes. This allows the data points to be more evenly distributed among processes than either G1km or G5km, both of which result in load imbalance across the processes.

4.2 Hardware

PISM can run with little or no modification on a wide range of hardware platforms, ranging from a modest laptop to a state of the art supercomputer. This has the advantage of accelerating the pace of research and development, since a user can develop and test code on a local workstation expect that the same code will run on a supercomputer.

We have focused on two such systems in our analysis: a local desktop workstation named Tessy, and the Ranger supercomputer at the University of Texas at Austin [35]. Both systems use 64-bit builds of the Linux operating system, and both use the same version of high-level libraries, including Unidata NetCDF-4.2 [38] and Parallel-NetCDF-1.3 [28]. We will discuss both systems below.

4.2.1 Tessy

Tessy is a desktop workstation with two Intel® Xeon® processors providing four physical cores each. These 8 physical cores are presented to the operating system as 16 logical cores through the use of Hyper-Threading. The system has 48 gigabytes of main memory in the form of 12 four gigabyte modules. Persistent storage is provided with a single mechanical hard drive. The system is running Debian Linux with the 2.6.32-5-amd64 kernel and is using an Ext3 filesystem. The software stack is built with the Gnu Compiler (GCC), and it uses the MPICH2-1.4 [24] implementation of MPI-IO [25].

4.2.2 Ranger

The Ranger supercomputer at the Texas Advanced Computing Center at the University of Texas at Austin is a Sun Constellation Linux cluster, which first went on-line in 2008 [35]. It consists of 3,936 compute nodes with 4 quad core AMD Operton™ processors and 32GB of memory each. These compute nodes do not have any persistent local storage, but do provide a 300MB temporary file system. Persistent storage is available in a Lustre filesystem [22], which we describe below. The system is connected through an InfiniBand [17] interconnect, which provides a theoretical point-to-point throughput of 1GB per second. Ranger uses the version 2.6.18.8 Linux kernel, and the software stack is built with the Intel compiler. Its MPI-IO implementation is MVAPICH2-1.8 [26].

The Lustre ® [22] filesystem is a high-performance, parallel, distributed filesystem. It consists of four components that collaboratively provide persistent storage for many client machines. These components are Metadata Server (MDS), Metadata Target (MDT), Object Storage Server (OSS), and Object Storage Target (OST). The MDS manages the metadata and catalog services for the filesystem, and an MDT is used to store the metadata. This metadata maps file paths to storage locations, and it manages ownership and access permissions for objects in the filesystem. The OSTs are the actual hardware storage devices that store the file data. A single OST may correspond to a single hardware device, or it may correspond to a RAID array of hardware devices. The OSSs manage the OSTs, handling I/O requests and dispatching them to the appropriate OSTs. The Lustre filesystem used in this research consisted of 50 OSSs and 300 OSTs, thus each OSS hosts 6 OSTs.

A single file may be distributed across many OSTs using a technique called *striping*. Using this technique, the file is divided into fixed-size blocks, and these blocks are mapped to OSTs with round robin assignment. Striping on Lustre filesystems can be customized on a per-file and per-directory basis by setting the *stripe-size* and *stripe-count* parameters. The stripe-size controls the size, in bytes, of each stripe, and the stripe-count controls the number of OSTs over which stripes are distributed.

The Lustre filesystem derives it parallelism and scalability from two sources. First, the metadata operations are handled by a separate component than the file I/O operations. The MDS handles all metadata operations, such as directory listing and mapping of filesystem paths to hardware devices. Once a client retrieves the metadata for a file, it can communicate directly with the OSSs to perform file I/O operations. The second source of parallelism comes from the striping of files across many OSTs, which provides parallel access to shared files for multiple application processes.

4.3 Measuring Performance

In order to measure the performance, we are using several techniques to quantify aspects of the run time of the simulation. We focus on the run time because this is the primary indicator of the expense to the user. On a shared supercomputing system like Ranger, users are charged based on the wall-clock run time of the application and the number of processing elements used. When the simulation runs faster, more experiments can be done within a fixed budget and time span.

At the coarsest level, we use the the POSIX `time` command to measure the wall-clock run time of the whole simulation. This provides the statistic that is most relevant to the end user, but it does not provide much insight into the performance characteristics of various components within the simulation. In order to gain more insight, we measured the performance of individual components with PETSc's profiling features.

4.4 Calculation and Presentation of Results

Throughout this thesis, we will present results in a graphical form, primarily using clustered column charts. In all cases, the height of the column gives the mean of the results we collected across multiple test iterations. Generally, we consider the average of five or more test iterations, yet in a few cases, time and resources prevented us from collecting even this many results. In such cases, we will make note of the limited number of test iterations. Error bars are provided for each column, which indicate the variation between test iterations.

An attempt was made to run our experiments at different times of day and on different days of the week. We do this because Ranger is a shared system, and it experiences varying workloads that can affect performance and measurements. Programs are submitted to a shared batch scheduling queue over which we had not control.

CHAPTER 5

INITIAL PERFORMANCE AND SCALABILITY

At the beginning of our investigations, PISM offered sluggish performance and presented a clear need for optimization. In this chapter, we quantify PISM's initial performance and inability to scale to large problem sizes. We will separately discuss the compute and I/O performance of the simulation and examine how well these scale with increasing computational resources. Finally, we present our modifications to the initial code and show that these changes have a drastic and positive impact on performance.

5.1 Initial Performance

The starting point for our investigations was the 'dev' branch of PISM's GitHub repository [31], shortly after the release of PISM version 0.5. We modified the code to implement some profiling and performance measuring features, and to ensure that the code would compile and run on Ranger. Otherwise, we made no attempt to optimize its performance.

We first ran the A10km model on Tessy, testing each I/O technique across several different process counts. The results are shown in Figure 5.1. As can be seen, there was a significant increase in performance for increasing the number of processes, up to 8 processes. Larger process counts seem to hurt performance, but not significantly. One reason for this is that Tessy only has 8 physical cores, and we are relying upon the logical cores presented via Hyper-Threading for the tests with 12 and 16 processes.

The next thing we see is that there is not a significant difference in performance between the three I/O techniques. This is not unexpected, since Tessy does not provide a parallel file system.

In our next set of experiments, we ran the same A10km model on Ranger, again testing each I/O technique and a variety of process counts. In addition to specifying the number of nodes, Ranger allows the user to specify how many processes should be executed at each node. In particular, one may specify the *wayness* of an application when it is launched,

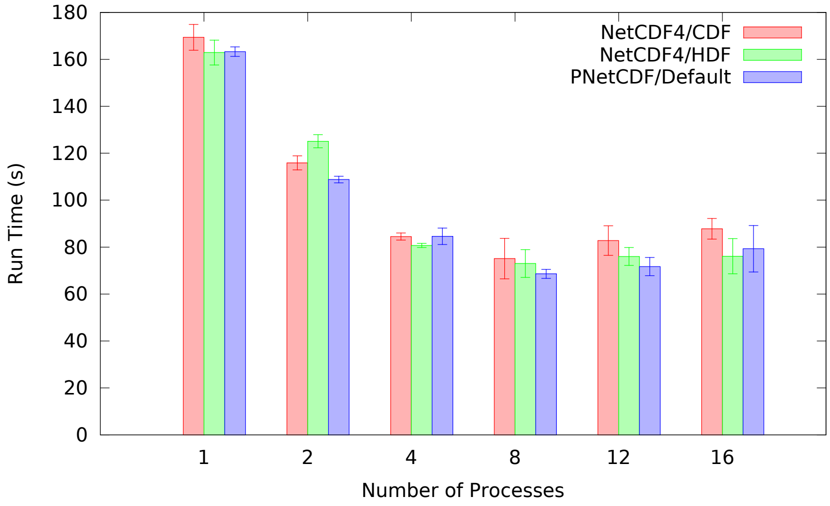


Figure 5.1. Initial A10km Performance on Tessy. The total run time, including both computation and I/O, of the A10km simulation on Tessy for each I/O technique. This chart shows the run time in seconds as a function of the I/O technique and process count.

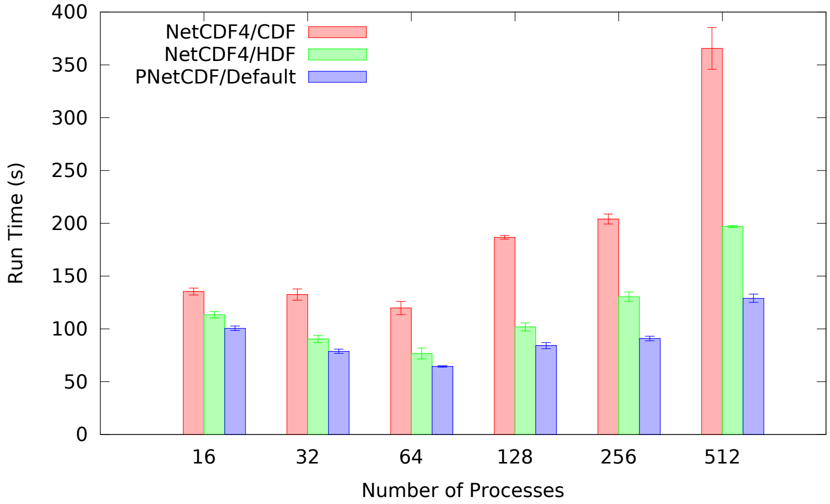


Figure 5.2. Initial A10km Performance on Ranger. The total run time, including both computation and I/O, of the A10km simulation on Ranger for each I/O technique. This chart shows the run time in seconds as a function of the I/O technique and number of processes.

which tells the system how many processes to run on each compute node. The minimum wayness is 1 (1-way), which assigns one process per node, and the maximum is 16 (16-way), which assigns 16 processes per node. Our first set of experiments fixed the wayness at four, which results in one process per physical processor on each node.

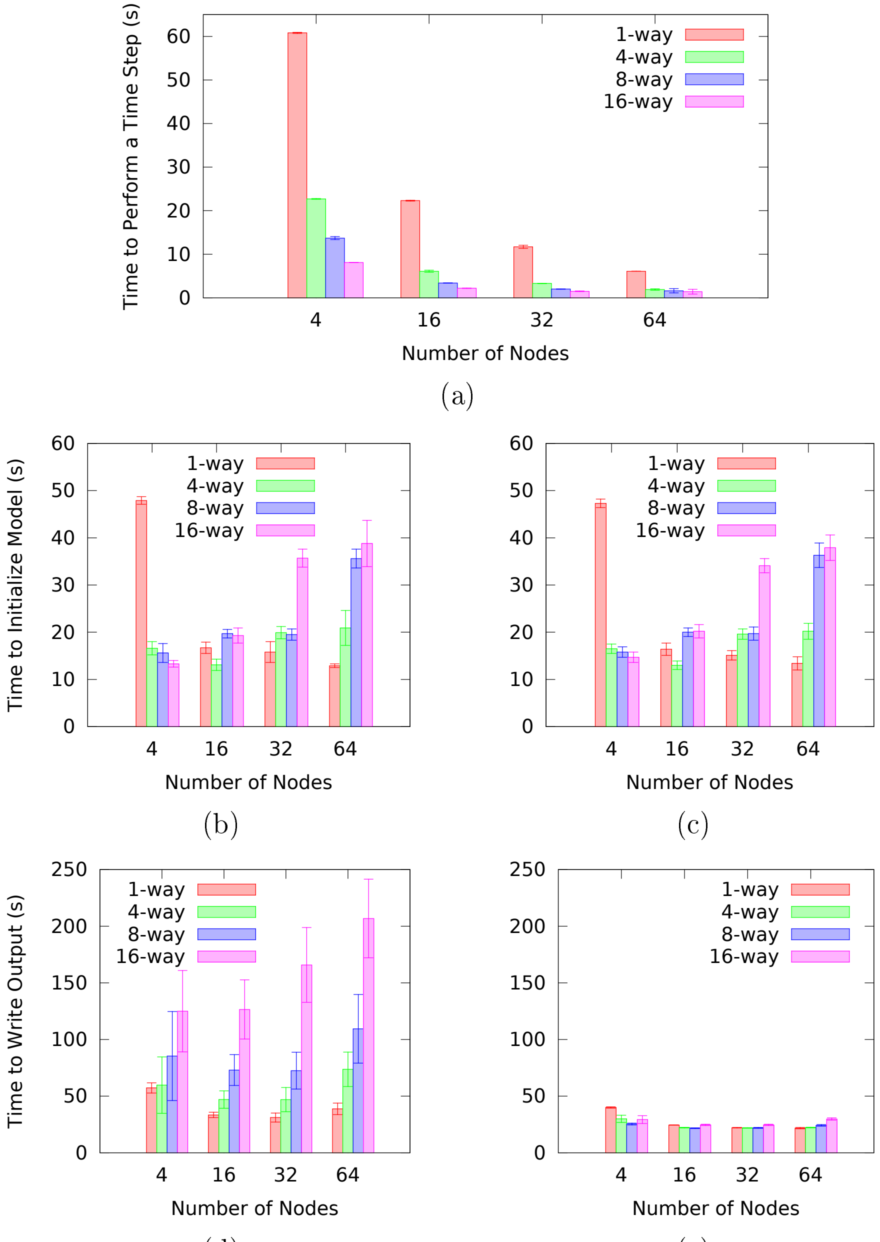
The resulting simulation run times are presented in Figure 5.2. In these results, we see that Ranger provides very different performance characteristics than Tessy. While we saw significant improvement in performance by increasing the number of processes on Tessy (up to the number of cores), we see a general trend toward diminished performance as the number of processes increases on Ranger. We also see a significant difference in performance based on the I/O technique that is used, where both of the parallel I/O techniques (NetCDF4/HDF and PNetCDF/Default) perform better than the serial I/O technique. This is not unexpected, and we note that the difference in performance increases with increasing process counts.

5.2 Compute, Initialization, and Write Performance

We next investigated the contribution of each stage of the simulation to the total run time on Ranger. We used the profiling features available in PETSc to time the initialization, compute, and write stages of the simulation. In the initialization stage, PISM reads input data from disk and constructs its internal data structures. In the compute stage, PISM models the evolution of the ice sheet over time. The write stage consists of writing the final model state to disk.

The results are shown in Figure 5.3. In these charts, the number of nodes is shown on the x-axis, and the number of processes can be calculated by multiplying the number of nodes by the wayness. In these experiments, the largest experiment we performed used 16-way processing with 64 nodes for a total of 1,024 processes.

First, we consider the performance of compute stage, which is shown in Figure 5.3 (a). Since the compute stage does not require significant file I/O, we do not show separate graphs for each I/O technique. As can be seen, the compute performance of the simulation scales exceptionally well with increasing process counts. improves as the number of nodes



(a)

(b)

(c)

(d)

(e)

Figure 5.3. Initial A10km Performance on Ranger, by Stage. (a) shows the performance for the compute stage; (b) shows the initialization performance for NetCDF4/HDF, and (c) shows the initialization performance for PNetCDF/Default; (d) shows the write performance for NetCDF4/HDF, and (e) shows the write performance for PNetCDF/Default.

increases and as the wayness (and hence the number of processes) increases. Thus, it is necessary to look to the initialization and output stages to explain the poor scalability characteristics we saw in Figure 5.2.

We next looked at the performance of the initialization stage with each of the parallel I/O techniques, which is shown in Figure 5.3 (b) and (c). We ignored the serial NetCDF4/CDF technique at this point, since, as shown in Figure 5.2, it performs significantly worse than either of the parallel techniques. Despite having to read data from disk during the initialization phase, we see that the choice of I/O technique has little influence on the performance of the initialization stage.

We next looked at the write stage to explain the results seen in Figure 5.2. Figure 5.3 (d) and (e) show the write performance for NetCDF4/HDF and PNetCDF/Default, respectively. As can be seen, neither mode offers a significant performance improvement as the number of processes increases. In fact, NetCDF4/HDF actually exhibits a significant performance degradation as we add more processes. Here, we also note the consistency provided by PNetCDF/Default, showing little change in performance across all experiments. Comparatively, NetCDF4/HDF shows significant variability across the experiments, as is evident in the large error bars.

5.3 Scaling Considerations

Next, we consider how PISM performs as the number of grid points increases (i.e. as we scale to a higher resolution model) For this, we turned to the G1km model, which uses more than 15 times as many data points as the A10km model. This increased the memory requirement to more than a terabyte, which excluded Tessy from the tests and required more than 32 nodes on Ranger. Furthermore, due to the size of the model, only the NetCDF4/HDF output technique could be used. This is because the other techniques (NetCDF4/CDF and PNetCDF/Default) use 32-bit offsets for variables, as discussed in Section 2.3.1.

Our first set of experiments with the G1km model consisted of using 256 and 512 processes spread across 64 and 128 nodes (4-way processing), respectively. We were only

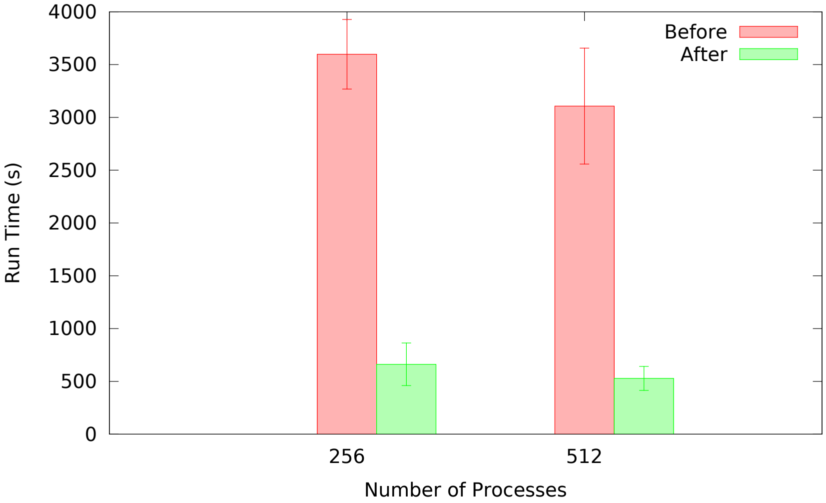


Figure 5.4. Improved G1km Performance on Ranger. This figure shows the total run time of the G1km simulation before and after the modifications described in Section 5.4

able to gather two data points for each process count, due to the excessively long run times and our limited supercomputing resources. Each run took more than 45 minutes to complete, with one test taking more than an hour. Further inspection revealed that more than half of the total run time was spent in the initialization stage, prompting a closer look at this phase of execution.

5.4 Identification of Performance Problems

Internally, PISM consists of many components that operate on both shared and private data structures. Each of these components is responsible for reading the data it needs during initialization, and writing the results it produces during the output stage. Due to this architecture, there are many entry points into PISM's I/O system, which presents the opportunity for components to use the I/O system inconsistently.

As we investigated the initialization code, we found that some of these components were using a serial I/O technique to read data stored in HDF files. With this read pattern, a single process reads all file data and then distributes the data to the other processes. Due to the magnitude of the data set and the number of processes involved, reading the file serially resulted in a significant performance bottleneck. It also results in unnecessary

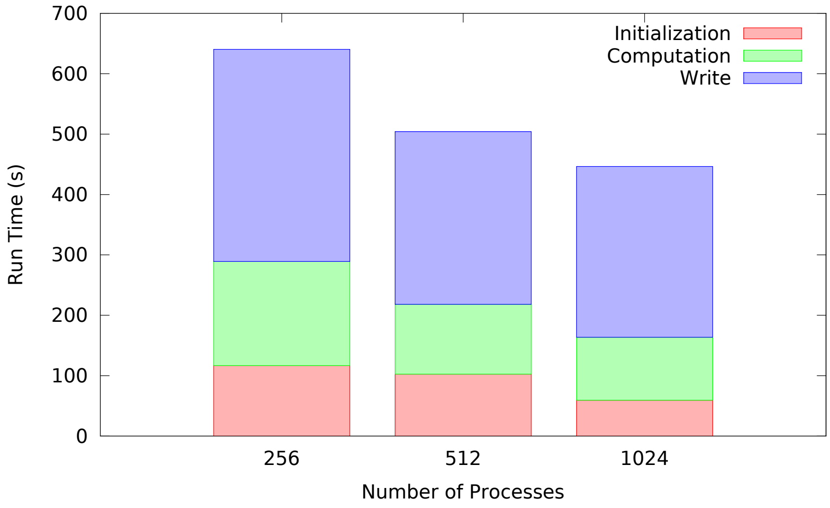


Figure 5.5. Improved G1km Performance on Ranger by Stage. This figure shows the run time of the G1km simulation, separated by stage, after the improvements described in Section 5.4

network traffic, since the input data travels across the network once for the initial read and again as the data is redistributed.

Thus, our first step was to ensure that the input data is read using the NetCDF4/-HDF technique. We then ran the test of the G1km model, again using 256 and 512 processes spread across 64 and 128 nodes, respectively. The results are shown in Figure 5.4. As can be seen, this simple modification had a tremendous impact on performance. The PISM developers have since independently corrected this issue.

With our improvement, we were able to run more tests using a wider range of process counts. The results of these experiments are shown in Figure 5.5. In these results, we see a steady improvement in performance as more processes are used, with each of the initialization, compute, and write stage times decreasing. We also see that it is no longer the input stage that dominates the run time. Instead, the output stage appears to be the bottleneck in performance.

5.5 Discussion

The experimental studies described in this chapter have established that the compute phase of the simulation scales remarkably well with increased processor count. It was the file I/O performance exhibited by the NetCDF4/HDF I/O technique that dominated the performance of the simulation. Indeed, we see that with the G1km model, it can take nearly twice as long to write the model state to file as it does to update the model state within a given time step.

In the case of the G1km model, we could not use the PNetCDF/Default technique due the limitations of the CDF1 and CDF2 formats. However, with the smaller A10km model, we saw that the PNetCDF library and the PNetCDF/Default technique offer significantly better performance than the NetCDF4/HDF technique. This brings up the question of whether PNetCDF would be able to scale to the much larger data set of the G1km model. We investigate this issue in the next chapter.

CHAPTER 6
LARGE FILE SUPPORT WITH PARALLEL-NETCDF

When simulating the Greenland ice sheet at one-kilometer resolution, PISM uses a 1,501 × 2,801 × 401 computational box, which consists of more than 1.6 billion data points. A three-dimensional, double-precision, floating-point variable requires approximately 13GB of file storage space at this resolution, which prohibits the use of the CDF1 and CDF2 file formats and the NetCDF4/CDF and PNetCDF/Default I/O techniques. As of PISM version 0.5, NetCDF4/HDF was the only I/O option that was supported for such large models.

Argonne's Parallel-NetCDF library [28, 21] also provides support for large variables, though it accomplishes this with the CDF5 format instead of the HDF5 format. We saw in the previous chapter that on Ranger, the PNetCDF/Default output technique offered consistently better performance than NetCDF4/HDF in the A10km model, so we were curious if it would continue to perform well at a larger scale. To do this, we implemented support for the CDF5 file format in PISM. We refer to our modified I/O technique as PNetCDF/CDF5. In this chapter, we discuss our implementation of this previously unavailable I/O technique, as well as its performance and scalability characteristics.

6.1 Initial Implementation

The first step was to enable large file creation with PNetCDF, which required that the `NC_64BIT_DATA` flag be passed to `ncmpi_create` calls. When this flag is used, new files created with the PNetCDF library will use the CDF5 format. Changing to this different file format revealed an interesting feature in PISM: when using the PNetCDF library to write output, it first used NetCDF4 to *stage* the file, and then used PNetCDF to fill the file with data. Staging is a technique that separates the creation of an output file from the population of that file. In the case of PISM, NetCDF4 was being used to create the output file, configure the metadata for the file, and the write a small amount of data to the file. After the file was staged by NetCDF4, PNetCDF performed the subsequent I/O operations. This staging approach does not work when writing CDF5 files, because NetCDF4 does not support that format. Therefore, we implemented a direct approach, whereby PNetCDF both creates and populates the output file.

Comments in the PISM source code noted that PNetCDF was not being used to write file metadata for performance reasons. The PISM developers had apparently performed some experimentation and found that they were able to achieve better performance with the staging approach. In the experiments described in the next section, we encounter these same performance issues. We then determine the root cause of the poor performance and implement a solution.

6.2 Initial Performance

After ensuring that PNetCDF/CDF5 was used to both create and populate the output file, we tested our implementation using the G1km model. As noted earlier, the G1km model requires CDF5's large variable support, because at this resolution, a three-dimensional variables requires more than 4GB of storage space. However, PISM became unresponsive while writing output, and we aborted the simulation after it became clear that it was not making progress.

We next executed a much smaller simulation, using the A10km model, in an attempt to learn more about our implementation. While there was a lengthy pause during the output phase, the simulation did run to completion and produce a valid CDF5 output file. We then performed a series of experiments to determine the effect that different process counts have on the performance.

In these experiments, we computed the time required to write the output file using 16, 32, and 64 processes. The results are shown in Figure 6.1. As can be seen, switching from the staging approach to the direct approach resulted in more than a 10-fold performance penalty. It is interesting to note that when we performed similar experiments on Tessy, we did not incur this performance penalty.

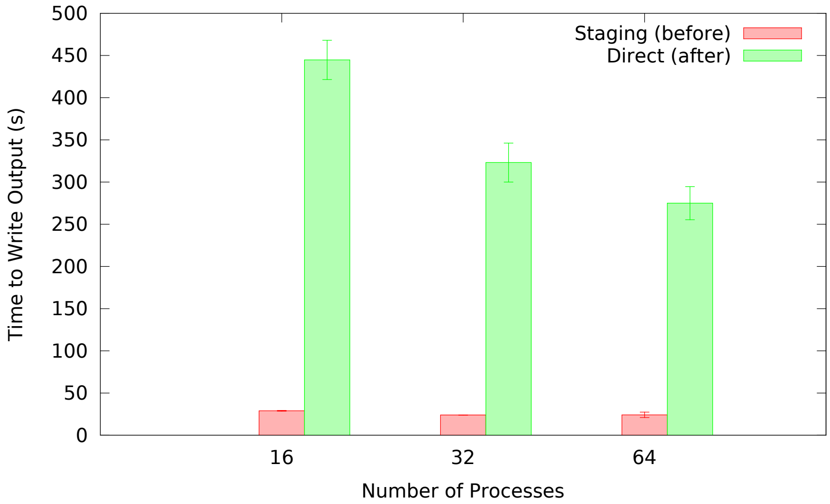


Figure 6.1. Initial PNetCDF/CDF5 Performance at A10km on Ranger. This figure shows the time it takes to write output using the PNetCDF library with the A10km model on Ranger, with the staging approach that PISM used before our modifications and the new direct approach described in Section 6.1.

6.3 Diagnosis of Performance Penalty

The switch from the staging approach to the direct approach was the cause of the slowdown. In order to understand why this caused such a dramatic performance degradation, it is first necessary to understand how a CDF file is created and populated using two modes of operation termed *define-mode* and *data-mode*.

When operating in define-mode, the metadata for a file may be created or modified, which includes the definition of dimensions, variables, and attributes. It is important to note that changing the metadata for a file may alter the file structure, including the location of variables within the file. In data-mode, one may read and write variable data, but the metadata for the dataset cannot be changed.

When the application switches from define-mode to data-mode, the I/O library allocates space for any new variables that were defined while in define-mode. This may require the file to be restructured such that it remains consistent with the CDF format. If variable data has already been written to the file, this restructuring may result in the need to copy the data to a different location in the file, and the cost of this copy operation is dependent on the size of the data that must be copied.

Given an understanding of the two modes of operation, we now re-visit our discussion of the format required for CDF files, which was shown in Figure 2.4 on page 10. In particular, all fixed-size variables (i.e. those whose size will not change during the simulation), must be stored immediately following the file header. The fixed-size variables are then followed immediately by the record variables. Note that in PISM, the variables representing the current state of the ice sheet, which are recomputed during each time step, are stored as record variables.

Now, consider the way in which PISM creates the output file, which is summarized in part (a) of Figure 6.2. In the first `for`-loop, PISM defines and initializes the data associated with each dimension (e.g. x, y, z, and *time*). This involves defining a dimension, defining a corresponding one dimensional variable to store coordinate values for that dimension, and writing data for that variable. In the following loop, the variables associated with the model state are defined. Most of these variables evolve over time and are therefore record variables. However, there are some model state variables that do not evolve over time, and are represented as fixed-size variables.

After all of the model state variables have been defined, PISM switches from define-mode to data-mode so that it can write data for the variables. As noted earlier, this switch triggers a restructuring of the file. Since data had already been written for the *time* variable, which is a record variable, PNetCDF recognized that there was a single record of data in the record section and moved it to a new location. This file restructuring is illustrated in Figure 6.3.

The problem we encountered when executing the G1km model was that the size of a single record is on the order of gigabytes of data, which cannot be copied in reasonable amount of time. It is worth noting that most of the record consisted of uninitialized data, as PISM had only written data for the *time* variable. We hypothesize that NetCDF4 has been optimized to avoid such an unnecessary copy, and this is what motivated the development of the staging technique in PISM.

Once we understood the nature of the problem, we examined the various mechanisms in PNetCDF that are designed to accommodate changes in the file structure. Unfortunately,

```
for each dimension d:
  DefineDimension(d)
  DefineVariable(d)
  EndDef()
  WriteDataForVariable(d)
  ReDef()                                 NetCDF4

for each model state variable v:
  DefineVariable(v)

EndDef()

for each model state variable v:
  WriteDataForVariable(v)                 PNetCDF
```

(a)

```
for each dimension d:
  DefineDimension(d)
  DefineVariable(d)

for each model state variable v:
  DefineVariable(v)

EndDef()

for each dimension d:
  WriteDataForVariable(d)

for each model state variable v:
  WriteDataForVariable(v)
```

(b)

Figure 6.2. Output Procedures. This figure gives a pseudo-code summary of (a) the original output procedure, and (b) the updated output procedures that uses the direct approach. In (a), the braces to the right of the code show how the operations were divided between the two I/O libraries when using the staging approach. The `DefineDimension` and `DefineVariable` functions configure the metadata for a dimension and a variable, respectively, and the `WriteDataForVariable` function writes an array of data to file for a particular variable. The `EndDef` function initiates a switch from define-mode to data-mode, and the `ReDef` function switches from data-mode to define-mode.

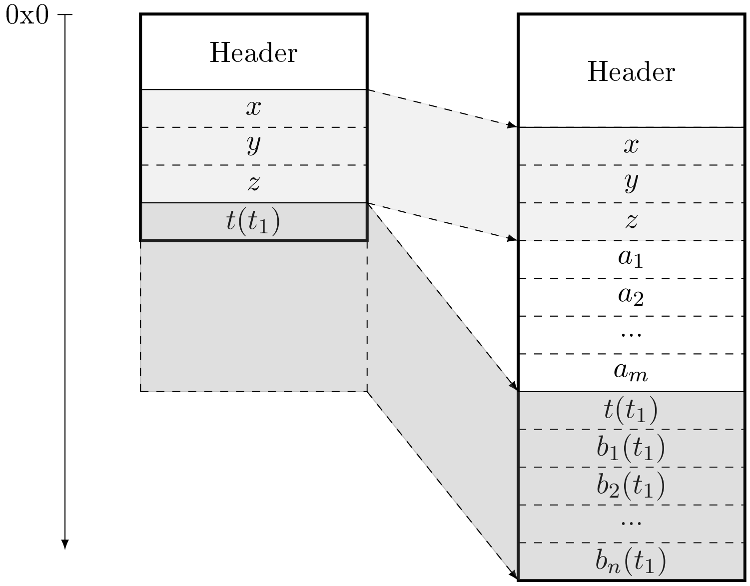


Figure 6.3. The Expense of Switching from Define-mode to Data-mode. In this example, the fixed-size variables $a_1, a_2, ..., a_m$ and the record variables $b_1, b_2, ..., b_n$ are defined after data has already been written for the x, y, z, and t variables. The header is expanded to store the metadata for the new variables, and the non-record section is expanded to make room for $a_1, a_2, ..., a_m$. As a result, the existing data in the non-record and record sections must be copied to a new location.

none of these mechanisms were applicable to this particular problem. Thus, we chose to address the problem at the application level by modifying the output procedure as shown in part (b) of Figure 6.2. Specifically, we modified the procedure such that all metadata is defined before writing any data. With this change, the output procedure only requires a single switch from define-mode to data-mode, and at the time of the switch, the metadata is fully defined but no data has yet been written. For this reason it is not necessary to relocate any of the variables, which makes the mode switch significantly more efficient.

6.4 Results

Once the application-level output procedure was revised, we ran a new set of experiments to determine its impact on performance. These experiments were conducted on both hardware platforms, and we measured the time it took to complete the output phase. In the following graphs, we show three series of data labeled "baseline", "v1", and "v2". These represent the output times for the original staging technique, PNetCDF/CDF5 without modification to the output procedure, and PNetCDF/CDF5 with the revised output procedure, respectively. The results are shown in Figure 6.4 (a) (for Ranger), and 6.4 (b) (for Tessy).

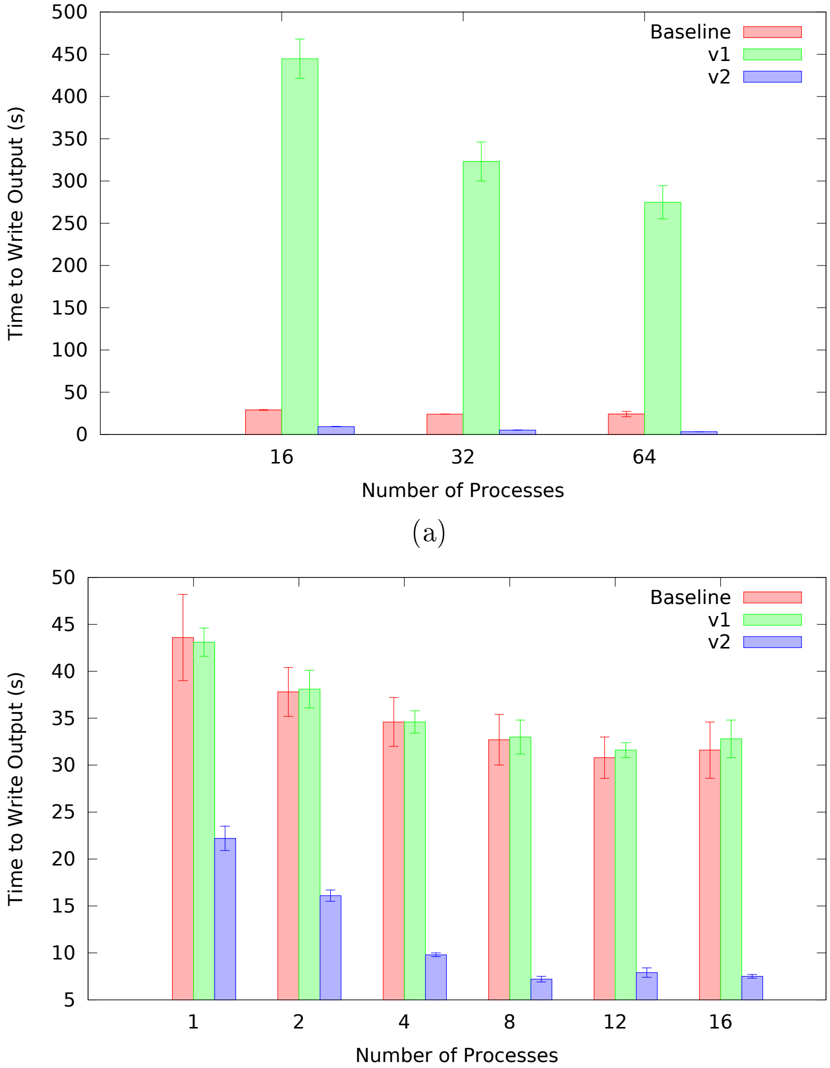
With respect to Ranger, the magnitude of the improvement in performance is quite striking. In particular, when comparing the "baseline" and "v2" implementations, the time to write the output file decreased by 80% when using 64 processes. We also note that the performance of our final implementation improves with an increase in the number of processes. This is encouraging, as it suggests that our implementation may be able to scale to larger problem sizes.

While we did not observe a performance penalty on Tessy when we moved from the staging approach to the direct approach, we still re-ran the tests to see if our solution impacted the performance on this platform as well. The results of these experiments are shown in Figure 6.4 (b). As can be seen, there is again a significant improvement in performance between the original and final implementations. As noted, moving from the staging approach tot he direct approach did not result in the tremendous degradation of performance that we observed on Ranger. While we can postulate some reasons for such significant differences in performance, we do not explore them further in this work.

Having observed significant performance improvements with the A10km model, we next returned to the much larger G1km model. Figure 6.5 compares the performance of the NetCDF4/HDF output technique with the PNetCDF/CDF5 output technique as the number of processes was increased from 256 to 2,048 processes. As can be seen, there is a dramatic improvement in performance for all process counts, with more than an 8-fold decrease in write time at 1,024 processes and more than a 7-fold decrease at 2,048 processes.

6.5 Discussion

We have observed that Argonne's Parallel-NetCDF library consistently performs better than Unidata's NetCDF4 library when writing output in parallel. This holds true for the relatively small A10km model as well as the much larger G1km model, it holds true on



(a)

(b)

Figure 6.4. PNetCDF/CDF5 Performance Improvements. This figure shows the time it takes to write output using the PNetCDF/CDF5 technique with the A10km model on Ranger (a) and Tessy (b), at each stage of the changes described in this chapter. The ‘baseline’ series gives the performance before any changes, the ‘v1’ series gives the performance after the changes described in Section 6.1, and the ‘v2’ series gives the performance after the changes described in Section 6.3.

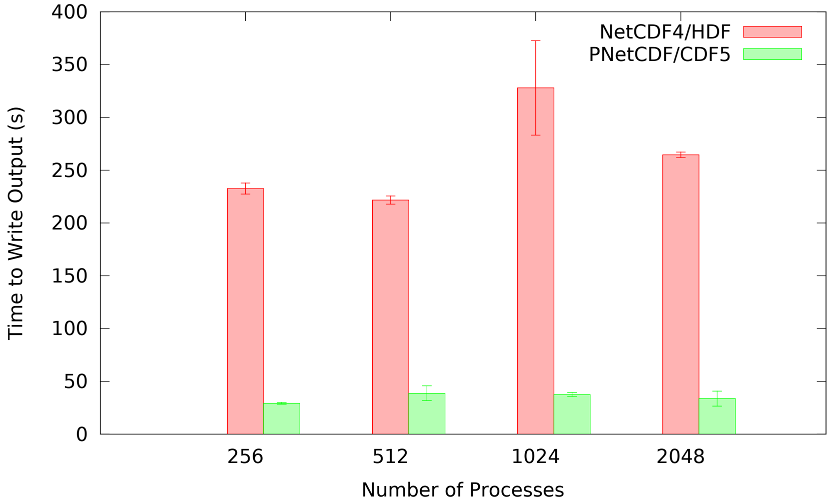


Figure 6.5. PNetCDF/CDF5 vs. NetCDF4/HDF on Ranger. This figure shows a comparison of the write performance that PISM achieves with the PNetCDF/CDF5 and NetCDF4/HDF output techniques when running the G1km model on Ranger.

Ranger's Lustre filesystem as well as Tessy's traditional filesystem, and it holds true for a wide variety of process counts on both systems. The problem, however, is that using CDF5 is not a feasible solution for many domain scientists because the CDF5 format is not widely supported, resulting in a lack of tools to assist in pre and post processing of the large datasets their applications require.

This points to the need for wider support for CDF5, which could be accomplished through modifications to Unidata's NetCDF4 library. There is currently an open support ticket with Unidata (NCF-163) that requests CDF5 support be added to the library, but comments on this ticket suggest that support is a low priority due to the lack of interest in the user community. We hope that the results we present here will lead to a greater interest.

CHAPTER 7
COLLECTIVE WRITE OPTIMIZATIONS FOR LUSTRE

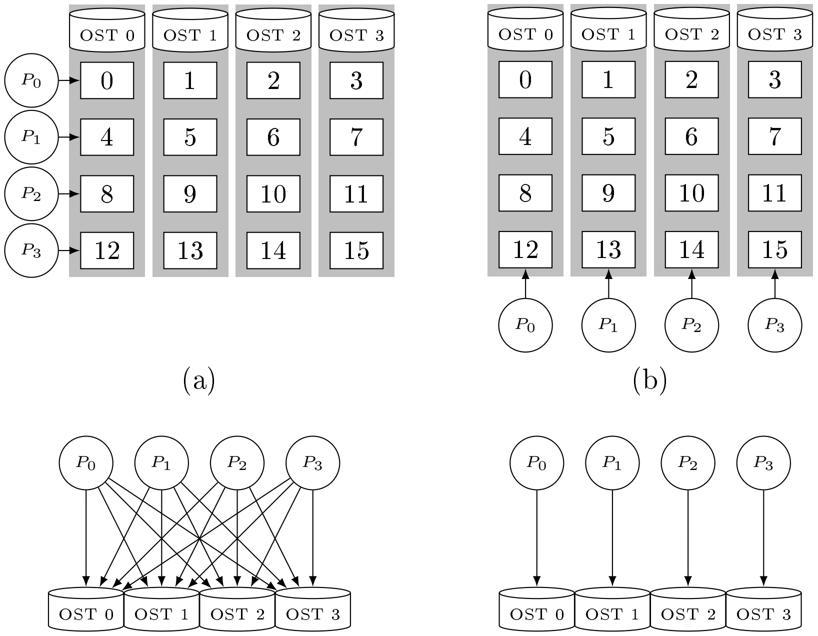
In this chapter, we describe our attempt to utilize a technique developed by Dickens and Logan [9] to improve PISM's write performance on the Lustre filesystem. In particular, they found that they were able to achieve up to 1000% improvement in write performance on Lustre using their YLib library instead of ROMIO. They explain that ROMIO operates under the assumption that the best performance is achieved by performing large contiguous write operations, which they contend can lead to poor performance on Lustre. Instead, they were able to achieve better performance by performing many smaller write operations that were both aligned on stripe boundaries and organized such that each process writes to only a small set of OSTs. Dickens and Logan used a simple benchmark application to test their new approach. In this work, we apply their techniques in PISM, which is orders of magnitude more complex than a simple benchmark.

However, we do first develop our own benchmark to test the performance of two different write patterns: large contiguous writes and smaller OST-aligned writes, both of which are described in detail in the next section. Given a significant speedup in performance using this benchmark, we decided to modify PISM's PNetCDF/CDF5 write technique to redistribute and write data using the OST-aligned write pattern. We begin with a discussion of the benchmark code.

7.1 Evaluation of Write Patterns

We first tested the performance of two different write patterns, which we refer to as *contiguous* and *OST-aligned*. Both of these patterns are illustrated in Figure 7.1.

In the contiguous write pattern, each process writes a large contiguous buffer to a shared file with a single independent write operation. This pattern is illustrated in Figure 7.1 (a) and (c). If the size of the write buffer is greater than the stripe count multiplied by the



(a) (b)

(c) (d)

Figure 7.1. The Contiguous and OST-Aligned Write Patterns. In this example, we use four processes, named P_0, P_1, P_2, and P_3, writing to four OSTs, and a total of 16 stripes of data evenly distributed among the processes. (a) shows the contiguous write pattern, with P_0 writing stripes 0, 1, 2, and 3, P_1 writing stripes 4, 5, 6, and 7, etc. This results in the all-to-all process-to-OST communication pattern shown in (c). (b) shows the stripe-aligned write pattern, with P_0 writing stripes 0, 4, 8, and 12, P_1 writing stripes 1, 5, 9, and 13, etc. This results in the one-to-one process-to-OST communication pattern shown in (d).

stripe size, then the result is an *all-to-all* communication pattern between processes and OSTs, where each process writes data to every OST.

In the OST-aligned write pattern, each process divides its data into stripe-sized blocks. Each process then writes its blocks to disk in a way that ensures that all of the data is written to the same OST. This pattern is illustrated in Figure 7.1 (b) and (d). We call the resulting process-to-OST communication pattern a *one-to-one* communication pattern, since each process writes data to exactly one OST.

Our benchmark consists of n processes, each of which writes a large amount of data into a shared file using the two write patterns described above. Each process writes 1GB of data to disk, and we measure the time that it takes to write the entire file with each write pattern.

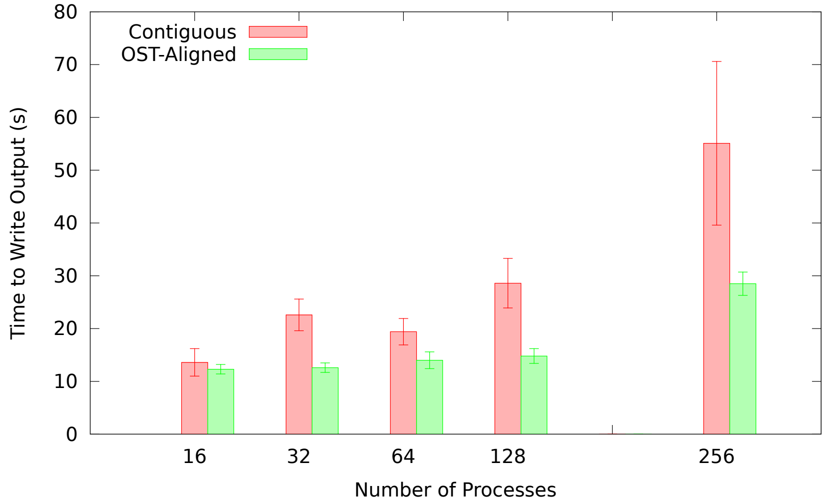


Figure 7.2. OST-Alinged vs. Contiguous Write Pattern Performance. This figure shows a comparison of the contiguous and OST-aligned write patterns on Ranger. For process counts between 16 and 128, the number of OSTs was equal to the number of processes. Due to system limitations, when testing at 256 processes, only 128 OSTs were used.

In our first experiment, we ran the benchmark, varying n (the number of processes) from 16 to 256, and we used 1-way assignment of processes to nodes. We used the system default stripe size of 1MB in all tests, and we vary the stripe count such that the number of OSTs is equal to the number of processes. However, at $n = 256$ processes, we used 128 OSTs, since Ranger does not allow a single file to span more than 160 OSTs, resulting in a two-to-one communication pattern between processes and OSTs. The results of these experiments are shown in Figure 7.2.

As can be seen in Figure 7.2, there is a clear advantage to writing in the OST-aligned pattern. At 128 processes, it took nearly twice as long to write with the contiguous pattern as it did to write with the OST-aligned pattern. It is also interesting to note the consistency of the performance achieved with the two patterns. For the OST-aligned pattern, the write times for between 16 and 128 processes are nearly identical, and very consistent between tests. Also, the write time at 256 processes is roughly twice that at 128, which is expected since twice as much data is being written to each OST. We see significantly more variation with the contiguous pattern, as is evident in the large error bars. This is an interesting result, but is outside the scope of this work.

7.2 Implementation in PNetCDF

Based on the positive results we saw in the previous section, we proceeded to implement the OST-aligned write pattern within the PNetCDF library for use within PISM. Implementing this write pattern in PNetCDF is considerably more complicated than it was in the benchmark, since we must redistribute data between processes to ensure that we achieve the one-to-one communication pattern at write time while still producing a valid CDF5 file.

We use a three-stage algorithm to redistribute data in preparation for the OST-aligned write. In the first stage, the processes exchange metadata so that each process knows which portion of the data is stored at every process. In the second stage, we select the *aggregator* processes, which are the subset of processes that are responsible for writing data to file. We select one aggregator for each OST, using an algorithm that evenly spreads the aggregators over the compute nodes. Each process then calculates which portions of its data must be sent to each aggregator, and each aggregator calculates which data it will receive from every other process. The data is redistributed in the third stage, such that each aggregator collects all of the data that will be written to a particular OST.

After the data is redistributed, the aggregator processes proceed to write data to file in the OST-aligned pattern. Each aggregator writes its data one stripe at a time, performing a seek and an independent write operation for each stripe of data to ensure that it writes to a single OST.

7.3 Results

We tested our PNetCDF implementation on Ranger with the G1km model. In these tests, we varied the number of nodes from 64 to 160, and we maintained the one-to-one correspondence between nodes and OSTs. We used 4-way processing for each test, so one in four processes is selected as an aggregator. For each test, we measured the amount of time required to write the output file using both the PNetCDF/CDF5 write, which uses ROMIO's implementation of MPI-IO collective operations, and our new OST-aligned write technique. Figure 7.3 shows the results of these experiments.

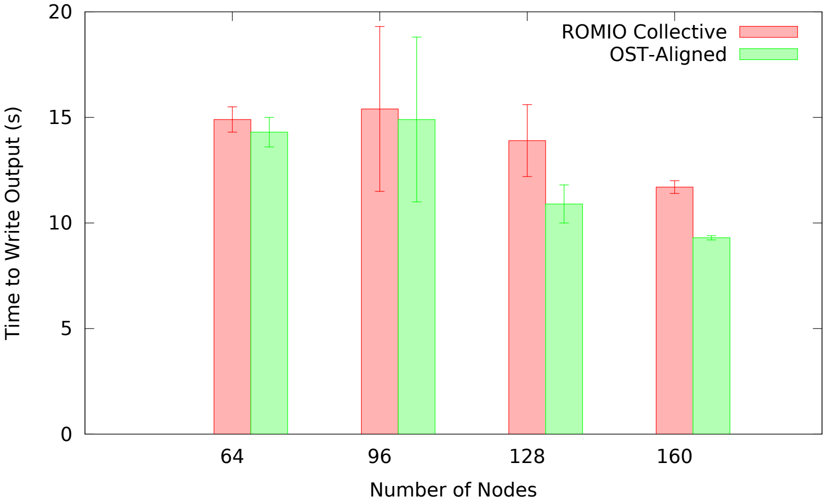


Figure 7.3. OST-Aligned Writes vs. PNetCDF/CDF for G1km. This figure shows the time required to write output for the G1km model using ROMIO's implementation of MPI-IO collective writes and our implementation of OST-aligned writes, as a function of the number of nodes. In each test, the stripe count was equal to the number of nodes, and we executed four processes at each node.

As can be seen in Figure 7.3, there is an advantage to using the OST-aligned write technique, which resulted in a 20% decrease in write time with 128 nodes. While we observe an improvement in performance at each data point, the improvement is not statistically significant for some process counts. This shows that more work is needed to use this technique efficiently in production software systems, such as PISM.

7.4 Discussion

In Figure 7.2, we saw as much as a 50% decrease in write time using the OST-aligned write pattern instead of the contiguous pattern. This gives an upper-bound on the performance improvement that we can expect with the OST-aligned write pattern in PISM, since it will need to perform the additional redistribution stage. As noted, the best performance that we saw with our implementation for PISM was a 20% decrease in write time.

The reason for this more modest improvement has to do with the redistribution of data, which must precede the OST-aligned write. This is a complicated and time consuming operation, involving significant inter-process communication. In our current implementation, the workload is poorly balanced where the aggregator processes must perform most of the work. While all processes must participate in the metadata exchange and calculations must send portions of their data to aggregators, the aggregator processes then have the additional task of receiving this data and organizing it such that it can be written to a single OST. In future work, we will look for more efficient communication patterns and develop techniques that lead to better load balancing.

CHAPTER 8
CONCLUSION

In this thesis, we have presented an in-depth study of the performance and scalability of the Parallel Ice Sheet Model (PISM). Our analysis showed that while PISM's compute performance scales quite well, the overall performance does not. Further investigation showed that it was the poor performance of PISM's I/O mechanisms that dominated overall run time. We then focused on the I/O system, determined the reasons for its poor performance, and then developed approaches to address these issues. The success of our work can be seen by looking at the performance of the G1km model on Ranger, where our initial naive attempt to run the simulation took more than an hour using the default PISM code and default system settings. After our improvements, combined with knowledge of and experiences with the Ranger supercomputer, we were able to complete the equivalent simulation in about three minutes, yielding a 20-fold improvement. In the following sections, we look at the work and accomplishments from each of the chapters that lead to such an improvement.

In Chapter 5, we analyzed the initial performance of PISM on both Ranger and Tessy. We found that the compute operations in PISM scale quite well as we allocated more computational resources, but that the I/O mechanisms used in PISM did not. In fact, when using NetCDF4/HDF, the system performance actually decreased as the number of processes was increased. However, with PNetCDF/Default, the performance did not degrade with increases in processes, and we consistently found that PNetCDF/Default performed significantly better than NetCDF4/HDF. The primary contribution in this chapter was that with a small improvement to PISM's initialization code, we were able to decrease the run time for the G1km simulation from over an hour to under 15 minutes. The work in this chapter also made it clear that writing output was consuming the majority of PISM's run time, so we chose to focus on this area for the next experiments.

In Chapter 6, we further investigated the PNetCDF library since it performed so well in our initial tests. Before our work, PNetCDF could not be used with large simulations, like G1km, so we modified PISM to add large data support via PNetCDF and the CDF5 file format. Doing so revealed that PISM's output procedure led to a significant slowdown with PNetCDF/CDF5. By modifying the way in which PISM wrote data and metadata, we were able to achieve a significant speedup on both of our hardware platforms. After our work, the PNetCDF/CDF5 output technique performed as much as 8 times faster than NetCDF4/HDF.

In Chapter 7, we attempted to validate previous research results showing that modifying the application write pattern can lead to significant performance gains on Lustre filesystems. We implemented the two different write patterns in a simple benchmark application and saw an increase in performance of up to 50% for the Lustre-optimized approach. Given these results, we implemented the new write pattern in PNetCDF for use with PISM. We found that our new implementation was able to perform as much as 20% faster than ROMIO's collective write algorithm with PISM.

8.1 Future Work

There are a number of areas that we would like to investigate further, and these fall into three categories. The first category includes extending our current results to include other hardware systems. The next category includes a more thorough investigation of the tuning parameters available in the NetCDF4 and HDF5 libraries, to attempt to match the performance that we achieved with PNetCDF. Finally, we would like to extend our investigation of the OST-aligned collective write pattern on the Lustre filesystem.

With respect to the first category, we would like to extend our work to other hardware systems. In particular, we would like to validate our results on the Stampede supercomputer, which is Ranger's successor, as well as the Marconi supercomputer at the University of Maine, which is a significantly smaller system with a substantially different architecture from Ranger. We would also like to extend our investigation of the parameters that are available on these systems, such as the number of compute nodes and wayness, and the stripe count and and stripe size on the Lustre filesystem.

Next, we would like to investigate opportunities to optimize the performance of NetCDF4. Our results with the PNetCDF library show that Ranger's I/O hardware is capable of significantly better performance than we observed with NetCDF4. NetCDF4 offers a number of parameters that can influence performance, such as data chunking, and we would like to evaluate how modifications to these parameters can influences the performance. We would also like to use the HDF5 library directly, rather than through the NetCDF4 interface, to see if the NetCDF4 library is responsible for some of the slowdown.

Finally, we would like to continue the development of the OST-aligned write technique that we investigated in Chapter 7. In particular, we would like to investigate techniques that may make our our redistribution algorithm more efficient, and we would like to extend our tests to include larger node and process counts. We would also like to test this algorithm with other applications to see how it handles other access patterns and data models, and to determine if the techniques can be generalized.

This work is the foundation for my dissertation work, which is funded by the Center for Remote Sensing of Ice Sheets (CReSIS) and is investigating embedded simulation techniques to efficiently handle high-resolution data. The work discussed in this thesis provides a solid foundation for our continuing research.

REFERENCES

[1] Adios - Oak Ridge Leadership Computing Facility. www.olcf.ornl.gov/center-projects/adios/, 2012.

[2] Satish Balay, Jed Brown, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.3, Argonne National Laboratory, 2012.

[3] Satish Balay, Jed Brown, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2012. http://www.mcs.anl.gov/petsc.

[4] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[5] R. A. Bindschadler and 27 others. Ice-sheet model sensitivities to environmental forcing and their use in projecting future sea-level (the SeaRISE project). *Journal of Glaciology*, Submitted, 2012.

[6] E. Bueler and J. Brown. Shallow shelf approximation as a "sliding law" in a thermodynamically coupled ice sheet model. *J. Geophys. Res.*, 114, 2009.

[7] Kenin Coloma, Avery Ching, Alok Choudhary, Wei keng Liao, Rob Ross, Rajeev Thakur, and Lee Ward. A new flexible MPI collective I/O implementation. In *Proceedings of the IEEE International Conference on Cluster Computing*, September 2006.

[8] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. In *In Proceedings of the Supercomputing âĂŹ95*, 1995.

[9] Phillip M. Dickens and Jeremy Logan. A high performance implementation of MPI-IO for a Lustre file system environment. *Concurrency and Computation: Practice and Experience - Grid Computing, High Performance and Distributed Application*, Volume 22, August 2010.

[10] Phillip M. Dickens and Rajeev Thakur. Evaluation of collective i/o implementations on parallel architectures. *Journal of Parallel and Distributed Computing*, 61(8):1052 – 1076, 2001.

[11] Elmer/ice. http://elmerice.elmerfem.org/, 2012.

[12] Glimmer community ice sheet model. http://glimmer-cism.berlios.de/, 2012.

[13] Prasad Gogineni. CReSIS radar depth sounder data. http://data.cresis.ku.edu/, 2012.

[14] Hierarchical data format version 5. http://www.hdfgroup.org/HDF5, 2000-2010.

[15] HDF5 user's guide. http://www.hdfgroup.org/HDF5/doc/UG/index.html, 2012.

[16] Mark Howison, Quincey Koziol, David Knaak, John Mainzer, and John Shalf. Tuning HDF5 for Lustre File Systems. In *Proceedings of 2010 Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS10)*, Heraklion, Crete, Greece, September 2010. LBNL-4803E.

[17] Infiniband trade association home. http://www.infinibandta.org/, 2012.

[18] Intergovernmental Panel On Climate Change Ipcc. *Climate Change 2007: Synthesis Report. Contribution of Working Groups I, II and III to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change*, volume 446. IPCC, 2007.

[19] ISSM: Ice Sheet System Model. http://issm.jpl.nasa.gov/, 2012.

[20] Wei keng Liao and Alok Choudhary. Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols. In *SC '08 Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[21] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 39–, New York, NY, USA, 2003. ACM.

[22] Lustre. http://wiki.lustre.org/index.php/Main_Page, 2012.

[23] Message Passing Interface (MPI) Forum Home Page. http://www.mpi-forum.org/, 2012.

[24] MPICH. http://www.mpich.org/, 2012.

[25] MPI-2: Extensions to the message-passing interface. http://mpi-forum.org/docs/mpi-20-html/mpi2-report.html, 1997.

[26] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/, 2012.

[27] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael L. Best. File-access characteristics of parallel scientific workloads. *IEEE Trans. Parallel Distrib. Syst.*, 7(10):1075–1089, October 1996.

[28] Parallel netCDF: A high performance api for NetCDF file access. www.mcs.anl.gov/parallel-netcdf, 2012.

[29] PISM, a Parallel Ice Sheet Model. http://www.pism-docs.org, 2012.

[30] PISM, a Parallel Ice Sheet Model: User's manual. http://www.pism-docs.org/wiki/lib/exe/fetch.php?media=manual.pdf, 2012.

[31] Repository for Parallel Ice Sheet Model (PISM). https://github.com/pism/pism, 2012.

[32] SEACISM: A Scalable, Efficeint and Accurate Community Ice Sheet Model. http://www.csm.ornl.gov/SEACISM/, 2012.

[33] Searise assessment. http://websrv.cs.umt.edu/isis/index.php/SeaRISE_Assessment, 2012.

[34] Ice sheet model sicopolis. http://sicopolis.greveweb.net/, 2012.

[35] Texas Advanced Computing Center. Ranger user guide. http://www.tacc.utexas.edu/user-services/user-guides/ranger-user-guide, 2012.

[36] Rajeev Thakur, William Gropp, and Ewing Lusk. *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation.* Mathematics and Computer Science Division, Argonne National Laboratory, October 1997. ANL/MCS-TM-234.

[37] Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing noncontiguous accesses in mpi-io. *Parallel Computing*, 28:83–105, 2002.

[38] Unidata. NetCDF (Network Common Data Form). http://www.unidata.ucar.edu/software/netcdf/, 2012.

[39] R. Winkelmann, M. A. Martin, M. Haseloff, T. Albrecht, E. Bueler, C. Khroulev, and A. Levermann. The Potsdam Parallel Ice Sheet Model (PISM-PIK) Part 1: Model description. *The Cryosphere*, 5:715–726, 2011.

[40] M. Zingale. Flash i/o benchmark. http://flash.uchicago.edu/ zingale/flash_benchmark_io/.

BIOGRAPHY OF THE AUTHOR

Timothy Morey was born in Portland, Maine on January 17th, 1981. He was raised in Orland, Maine by parents John and Marian Morey, and graduated from George Stevens Academy in Blue Hill, Maine in 1999. He attended Hendrix College in Conway, Arkansas and graduated in May of 2003 with a Bachelor of Arts in Mathematics and Computer Science. He returned to Maine and worked as a pastry chef and software developer until entering the Computer Science graduate program at The University of Maine in the fall of 2010.

Timothy J. Morey is a candidate for the Master of Science degree in Computer Science from The University of Maine in May 2013.