

Flexible and Efficient Parallel I/O for Large-Scale Multi-component Simulations *

Xiaosong Ma[†] Xiangmin Jiao[‡] Michael Campbell[‡] Marianne Winslett[†]

Abstract

In this paper, we discuss our experience of providing high performance parallel I/O for a large-scale, on-going, multi-disciplinary simulation project for solid propellant rockets. We describe the performance and data management issues observed in this project and present our solutions, including (1) support for relatively fine-grained distribution of irregular datasets in parallel I/O, (2) a flexible data management facility for inter-module communication, and (3) two schemes to overlap computation with I/O. Performance results obtained from both development and production platforms show that our I/O optimizations can dramatically reduce the simulation’s visible I/O cost, as well as the number of disk files, and significantly improve the overall performance. Meanwhile, our data management facility helps to provide simulation developers with simple user interfaces for parallel I/O.

1 Introduction

Progress in developing high-performance processors and the use of massively parallel computers have enabled the development of large-scale scientific simulations. These applications often run for days and generate terabytes of output data, normally snapshots and checkpoints.¹ Typically a simulation code compute in discrete time-steps, writing out data to disks at certain time-step intervals. Efficient transfer of such periodic output to secondary storage has been a great challenge.

*Research funded by the U.S. Department of Energy through the Center for Simulation of Advanced Rockets under Subcontract B341494.

[†]Department of Computer Science, University of Illinois at Urbana-Champaign. {xma1, winslett}@cs.uiuc.edu

[‡]Center for Simulation of Advanced Rockets, University of Illinois at Urbana-Champaign. {jiao, mtcampbe}@csar.uiuc.edu

¹A snapshot stores the current “image” of simulation data, to be used for time-based visualization or analysis. A checkpoint saves enough simulation data for a future run to restart from the current time-step. Sometimes, a snapshot can also serve as a checkpoint.

Performance-wise, periodic I/O is now more bottleneck-prone than ever, for several reasons. First, the performance improvements of disks lag behind those of other system components. Second, as today’s supercomputers and clusters make it possible for scientists to explore massive parallelism in their applications with hundreds or even thousands of processors, the scalability of secondary storage becomes a more serious problem. With shared file systems, I/O bandwidth is often limited by the number of I/O channels and the number of disks, which are normally smaller than the number of processors. Further, the advances in processor and network performance have created higher demands for I/O: equipped with stronger processing power, larger disk storage capacity, and better displays, scientists can simulate larger problems and like to save more data at higher output frequencies, to provide a closer view of the simulations and to make more detailed animations.

Further, complex simulations face hard data management problems. Unlike parallel I/O performance, which has been an object of intensive research in the past decade [3, 10, 19, 20], data management issues for parallel I/O were addressed in relatively few previous studies [16, 17]. Typically, simulation data are written to disk files using a pre-defined layout required by post-processing tools. The same data layout is shared by multiple modules written by different groups of developers, and it often needs to evolve as the simulation code evolves. The above issues, plus the fact that the authors of the simulation codes are scientists whose expertise is often in fields other than computer science, often make it unproductive for them to manage their output data and perform I/O in the desired format, not to mention obtaining satisfactory I/O performance.

Through our experience of supporting parallel I/O for a cutting-edge whole-system rocket simulation, we found that some characteristics of this type of applications create new challenges for parallel I/O performance and data management. In this paper, we identify these issues and present our solutions: support for parallel output of large numbers of irregularly distributed

datasets, a data registration and management mechanism to facilitate inter-module communication for both I/O and computation, and flexible schemes for hiding the application-visible I/O cost. Advantages of our proposed approaches are demonstrated by performance results from the rocket simulation.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 introduces the rocket simulation and identifies the I/O issues it imposes. Section 4 describes the parallel I/O architecture and I/O libraries used. Section 5 presents our proposed data management mechanism, and section 6 presents our buffering schemes. Section 7 shows the performance results and section 8 concludes this paper.

2 Related Work

Previous work has also characterized scientific applications' I/O demands [7, 5, 12]. Our work follows the tradition of studying scientific applications' I/O requirements by identifying new issues and problems in an on-going, large-scale, and multi-disciplinary simulation, and by addressing both the performance and data management topics.

Many techniques have been proposed for high performance parallel I/O in general (e.g., [3, 10, 19, 20]). Most of these techniques are based on the concept of *collective I/O*, where processors collaborate to read/write data from/to files on a shared file system. We complement the above works by supporting collective I/O with independent irregular datasets on individual processors.

On data management for parallel I/O, No et al. proposed a scientific data management system [16, 17], which uses MPI-IO files for array data and a database for metadata. We address the same problem with a different approach: using widely supported scientific file formats to store both the array data and the metadata in portable files, and providing a high-level data management interface that facilitates both computation and I/O.

Some previous research addressed irregular datasets or irregular data distribution in parallel I/O context [4, 11, 15, 17]. In this paper, we investigate irregular data distributions in the form of collections of independent mesh blocks, which, to our knowledge, has not been discussed in previous work.

We provide high-level I/O services through an extensible modular software framework. In the literature, a few other frameworks also aim at extensibility and modularity to ease the development and integration of modules for large-scale parallel simulations, but with different focuses (e.g., [1, 2, 18]). In contrast to these frameworks, our approach minimizes changes to application codes and hides from application codes more details of the framework and services.

3 Application Overview

3.1 Integrated Rocket Simulation Code

The rocket simulation discussed in this paper is an on-going, ten-year project taking place at the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois. The goal of CSAR is the detailed, whole-system simulation of solid propellant rockets under both normal and abnormal operating conditions [6]. Such a complex simulation poses significant research problems in computer and computational science, in areas such as parallel programming environments, performance analysis, numerical algorithms, computational geometry, and visualization.

The rocket simulation code developed at CSAR is referred to as GENx, with releases ranging from GEN0 in 1998 to the current GEN2.5. To provide flexibility, GENx allows users to plug in different modules for each utility service and/or physics computation. Figure 1(a) shows the overall architecture of GEN2.5. On the left are the physics modules for gas dynamics, structural mechanics, and combustion. The gas dynamics solvers, Rocflo-MP and Rocflu-MP, are two multi-physics codes using multi-block structured and unstructured meshes, respectively. Rocsolid and Rocfrac are two structural mechanics solvers, both using an Arbitrary Lagrangian-Eulerian formulation. The combustion solver is composed of a two-dimensional framework Rocburn-2D and three nonlinear one-dimensional burn-rate models with integrated ignition models.

On the right are the computer science service modules. Rocface is responsible for transferring data at the fluid-solid interface. Rocblas provides parallel algebraic operators for jump conditions. Rocpanda and Rochdf are interchangeable modules providing parallel I/O services, whose output can be read directly by our in-house visualization tool Rocketeer, or read for restart. At the top is the manager module Rocman, which orchestrates the control- and data-flow of the overall simulation. In the middle are the integration framework Roccom, which glues all modules together, and the communication libraries, which can be either MPI or Charm++. Charm++ provides additional functionality such as dynamic load balancing.

3.2 I/O Challenges in GENx

GENx performs extensive file output once every certain number of time-steps, to write out the current state of computation for multiple computational modules. In contrast, input is performed only for initialization at the beginning of the run, either starting from the initial data

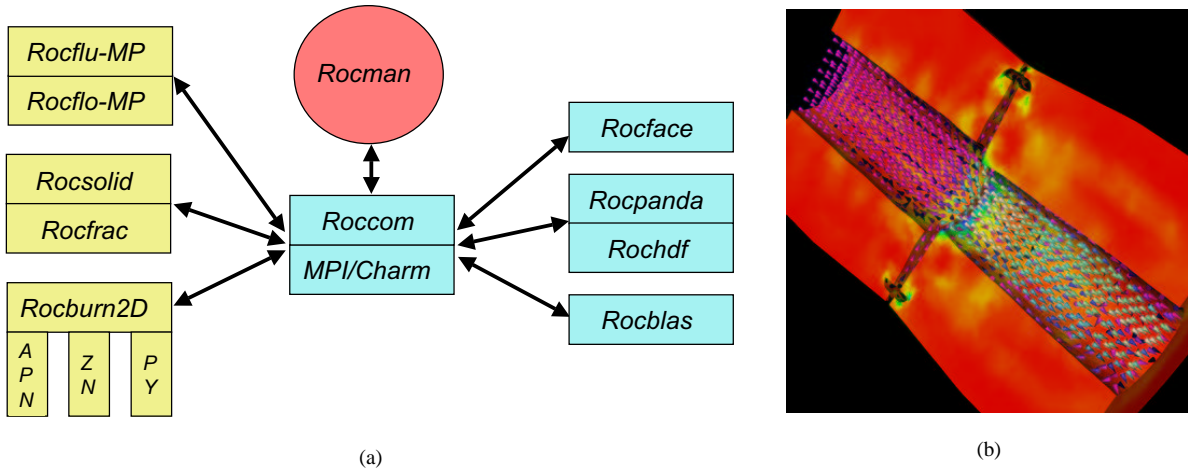


Figure 1. (a) Modules in GEN2.5. (b) Visualization picture generated by Rocketeer that shows gas velocity and solid propellant average stress in a cutaway section of a booster rocket at 0.83 seconds after ignition.

files or restarting from checkpoint files written in a previous run. Therefore, in this paper, we mainly focus on the periodic output operations.

Through our experience of supporting parallel I/O for GENx in the past three years, we have discovered its characteristics that make high-performance I/O and efficient post-processing especially challenging:

First, data are distributed finely and irregularly. The simulation object is pre-partitioned into a large number of mesh blocks and each processor is assigned a number of such blocks. For the same material (e.g., solid or fluid), each block has similar attributes and data organization, but can have different sizes. These mesh blocks change as the propellant burns in the simulation, requiring adaptive refinement over time. During each output phase, GENx outputs a snapshot of current intermediate results, such as updated mesh data and node- or element-centered variables, based on the mesh blocks. This relatively fine-grained, irregular, and dynamically changing data distribution already creates data management problems for the computation and inter-module communication, as well as for parallel I/O. Because there are no global datasets, array-based collective I/O techniques [14, 15, 17, 19] do not apply here. MPI-IO [20] supports definition of this kind of data distribution through its general data distribution directives, but the large number of irregular mesh blocks makes the creation of the MPI file view on each processor a huge pain, and the file view must be re-calculated when the mesh blocks or their distribution change at runtime.

Second, metadata need to be coupled with “real” data for both computation and post-processing. Like many

other scientists, investigators at CSAR prefer to integrate metadata with array data in scientific data formats and the current choice is the Hierarchical Data Format (HDF) [9]. HDF organizes multiple datasets (both array data and metadata) in a single file, supports user-defined attributes for datasets, and is binary-portable. Unfortunately, writing in a scientific format normally takes much longer than in a plain binary format, and the relatively small blocks used in GENx present a further performance problem with HDF as the internal overhead of managing the datasets is significant [13]. Further, the data distribution style mentioned above makes it impractical for the user code to directly use parallel interfaces currently available with HDF5. These interfaces require all processors to collectively create each dataset in a shared file. As each dataset is relatively small and resides on a single processor in GENx, this restriction practically serializes a large part of I/O.

Third, inter-module communication is difficult. In such a large, multi-disciplinary project, the scientists in fields like fluid dynamics or combustion are not familiar with the internals of high-performance parallel I/O, nor do they know the detail about the simulation components from fields other than their own. Similarly, developers of the I/O modules are reluctant to look at the giant computation codes. Many codes in the computation modules are from stand-alone applications developed by different research subgroups, some in C, some in C++, and some in Fortran. These vastly heterogeneous computation modules need to interact with each other, and communicate with the I/O module. A data management scheme that facilitates this interaction both in computa-

tion and I/O is highly desired.

Fourth, the simulation codes keep evolving. There are big milestones such as upgrading from 2-D in GEN0 to 3-D in GEN1, and constant updates as new computation modules join in, test cases become more complicated and realistic, and individual modules change their algorithms or data structures. GENx’s visualization tool, Rocketeer, evolves as the rocket images become more and more detailed and requires corresponding changes in the organization of the HDF output files. Meanwhile, the primary version of HDF itself upgraded from HDF4 to HDF5 on most platforms during the development of GENx. The data management and I/O implementation need to shield developers from updates in other modules and file I/O interfaces as much as possible.

Finally, debugging runs have critical performance requirements. With such a complex and evolving simulation, the number of debugging runs far exceeds that of production runs. Since production runs are large and normally done on huge supercomputers where resources are tight and access is restricted by citizenship, most of the development and testing work is done on local platforms. Compared to production platforms, such local platforms often have inferior shared file system bandwidth. Even worse, in debugging runs the scientists often perform periodic output more frequently than in a typical production run.

In response to the above challenges, we have developed new approaches to I/O. To accommodate fine-grained, irregular, and dynamic distribution of data, we employed scalable I/O architectures for both collective and individual I/O. For data management, we designed and implemented a scheme that benefits both computation and I/O. For performance optimization, we focused on aggressively overlapping computation and I/O to reduce the application-visible I/O cost, rather than speeding up the actual file I/O. These techniques will be discussed in the next three sections respectively.

4 Parallel I/O Architectures for GENx

Before we describe the I/O architectures used in GENx, we first introduce the concept of a *data block*. A data block is a collection of arrays and metadata associated with the arrays. It is also the unit of work distributed to the compute processors. In GENx, a data block contains all the data based on a mesh block, including the mesh coordinates and connectivity data, and element-and/or node-centered variables on this mesh block, such as pressure, velocity, and temperature. HDF files generated by GENx are also organized by data blocks, with data from different arrays in the same data block stored in neighboring HDF datasets.

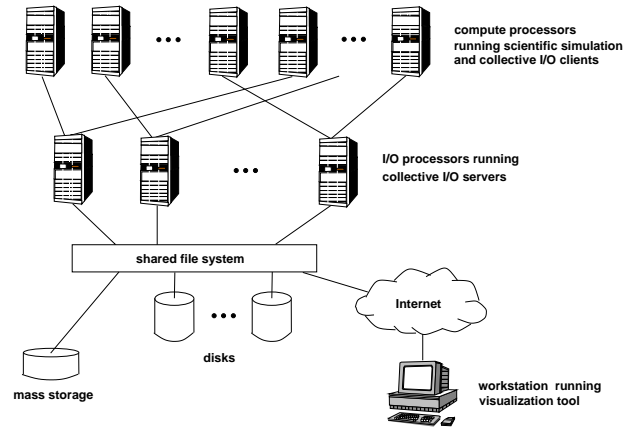


Figure 2. General architecture for collective I/O in GENx.

4.1 Collective I/O

Collective I/O enables global parallel I/O optimizations by having processors collaborate with one another in I/O operations. It also helps reduce the number of files generated by a simulation, which is important to the users. However, as explained previously, existing collective techniques do not easily apply to applications like GENx. Here, in lieu of global data structures that traditional collective I/O techniques rely upon, we designed a client-server scheme for performing collective I/O with fine-grained and irregular data distributions. Figure 2 shows the I/O architecture we used for collective file access. In addition to the compute processors (clients) running the simulation, some additional processors are dedicated as I/O processors (servers) running the server routine of our parallel I/O library. Typically, the client-to-server ratio used in GENx is larger than 8:1.

When the clients issue an output request, all the clients enter the collective output call. Each server is assigned to a number of clients. The servers will collect data blocks from the clients and write them to HDF files. The array data and metadata in these files are organized in the way specified by the post-processing application. Collective input is performed in a similar way for restart. For GENx, snapshot files for visualization also serve as checkpoints for restart. In this case, each client has a list of data blocks to read into memory after initialization. The servers collect the list of block IDs from the clients and build a mapping between block IDs and client process IDs. The restart files are assigned to the servers in a round-robin manner, and each server scans through its assigned files, finds requested data blocks, and sends them to appropriate clients.

The above design enables collective I/O based on irregular data distribution. It allows great flexibility in handling the ever-changing data and their distribution. First, the mesh blocks can expand or shrink over time, the number of mesh blocks can change with adaptive refinement, and the simulation developers need not to redefine the data distribution for I/O. Second, irregular distribution of data is easily handled. Third, it allows dynamic load-balancing, where data blocks may be migrated among processors, without affecting how I/O is done. In turn, dynamic load-balancing in computation benefits parallel I/O performance. In this architecture, the I/O workload is partitioned among the servers by partitioning the clients into equal-sized groups and having each server serve one such group. With fine-grained data distribution and dynamic load-balancing, the clients are likely to receive a balanced data assignment, resulting in a balanced I/O workload at the servers automatically. Fourth, the way restart is performed allows users to restart with a different number of servers than used in the previous run where the restart files were written. Finally, this architecture allows convenient overlap between computation and I/O with the memory and processing resources on the dedicated servers, which will be fully discussed in section 6.

Our collective I/O is implemented in a parallel I/O library called Rocpanda, a special edition of the Panda parallel I/O library [19]. Panda was designed for parallel I/O with multi-dimensional arrays distributed regularly in HPF style. In Rocpanda, we reused Panda’s framework of a client-server protocol, but added support for collective I/O with individual arrays on each client.

Simulations using Rocpanda with n clients and m servers run the job with $n + m$ processors. After MPI initialization, all processors perform Rocpanda initialization, where the processors split into two MPI communicators, for the clients and the servers respectively. The clients are passed the client communicator and continue with the rest of the simulation, using this communicator for communication among themselves. The servers, instead, enter the Rocpanda server routine and wait for clients’ collective I/O requests. To avoid resource contention on SMPs, we place the servers on different nodes as much as possible, by assigning processors with global rank $0, \frac{n}{m}, \frac{2n}{m} \dots$ to be servers. This placement yields an unexpected performance advantage on GENx’s production platform, an IBM SP with 16-way SMP nodes. We found that using 15 processors on each SMP node as compute processors and one processor as an I/O server actually gets visibly better performance in *computation* than using all the 16 processors as compute processors, because many operating system related tasks go to the server processor automatically, where the CPU is mostly idle. This suggests that dedicating one processor on each

node for I/O achieves double effects by off-loading both I/O and operating system tasks.

4.2 Individual I/O

The above collective I/O architecture generates fewer files and offload I/O from the compute processors, but it can also inconvenience simulation developers. First, when existing simulation codes are adapted to use Rocpanda, all the instances of MPI_COMM_WORLD need to be replaced by the client communicator returned by the Rocpanda initialization routine. Second, the decision to dedicate one processor as an I/O server on each SMP node must be made before the mesh is partitioned. Otherwise, the number of problem partitions in the scientific world is typically a power of two, as is the number of processors in an SMP node. Both problems require extra planning and foresight from users. Third, when the number of processors is limited, especially on debugging platforms, it can be difficult to obtain additional processors for I/O servers.

To allow parallel I/O in more general circumstances, we also provide a server-less architecture for individual I/O. In this architecture, each compute processor outputs its own data blocks. In GENx, this is done through another I/O library called Rochdf, which writes local blocks into individual HDF files. This simpler architecture avoids communication during I/O. In some circumstances Rochdf may offer additional performance advantages over Rocpanda because it generates smaller files, and HDF4 read/write performance does not scale well as the number of datasets increases in a file (unlike HDF5) [13]. On the other hand, having all the processors accessing files can create higher contention for I/O resources and cause degradation in I/O performance. The largest disadvantage of the individual I/O architecture, however, is that it creates the same number of files per snapshot as the number of compute processors. While CSAR’s visualization tool has no problem in processing files generated in this mode, having so many files certainly brings file management problems for production runs on a large number of processors.

For Rochdf, performance problems can be handled with the same approach used in Rocpanda: overlapping computation with I/O. Here there are no dedicated I/O servers to perform file I/O when the compute processors are computing, we create an extra I/O thread on the compute processors to issue I/O requests in the background. Details will be given in section 6.

5 Data Management and I/O Interface

We provide high-level parallel I/O services through a component-based integration framework called Roc-

com developed at CSAR. Roccom facilitates flexible inter-module data exchange and function invocation in parallel simulations, and is designed to maximize concurrency in development of code modules, minimize user effort for integration, and provide interoperability between different programming languages (in particular, C, C++, and Fortran 90). Note that Roccom is not specialized for integrating I/O with applications, but is a more general-purpose tool for integrating multi-component simulation codes. In effect, all the GENx components of CSAR are integrated through Roccom.

Roccom provides systematic methods for modules in a complex simulation to keep track of their data and to access data defined by other modules. Besides declaring variables and allocating buffers, each computation module registers its datasets through Roccom. These datasets can later be retrieved from Roccom by the same module or other modules. Functions can be registered and invoked in the same way. This scheme allows great independence in design and development of individual modules and hides the coding details of different research subgroups, which is convenient for both computation and I/O components. For parallel I/O, the computation modules can simply tell the I/O library: “write the mesh coordinates and the pressure value on all the mesh blocks”. Meanwhile, the I/O library never has to know how each data block is defined. A detailed description of Roccom is beyond the scope of this paper. Here we only briefly explain the concepts and interfaces related to parallel I/O.

Roccom’s design addresses several research issues. First, there needs to be an intuitive way for users to organize their data, with the irregular mesh blocks distribution style of GENx. Roccom organizes data and functions into distributed objects called *windows*. A window encapsulates a number of *data members*, such as the mesh (coordinates and connectivities), some associated data attributes, and public functions of a module, any of which can be empty. In a parallel setting, a window is partitioned into *panes*. A pane corresponds to a data block described in the previous section, and is owned by a single process, while a process may own any number of panes. All panes of a window must have the same collection of data members, although the size of each data member may vary. To use Roccom, the computation modules first create windows, then register their local data blocks as panes in appropriate windows, providing a unique pane ID. The components of the data blocks can later be retrieved from Roccom using the appropriate window name, attribute name, and pane ID.

Second, Roccom provides very simple and high-level parallel I/O interfaces through its data and function management mechanism. Scientists writing a computation module simply want to write out a collection of data

blocks to a file with a given name, and they see this as an atomic operation. Roccom enables Rocpanda and Rochdf to encapsulate all lower-level I/O operations into three high-level, file-format-independent, collective operations: `read_attribute`, `write_attribute`, and `sync`. Hidden under these one-step interfaces are file operations such as open, close, and data accesses. The `sync` interface is designed for performance analysis and debugging when I/O is overlapped with computation. When a `sync` request is issued, the compute processes will wait for previously issued I/O operations to complete. Compared to available parallel I/O interfaces such as parallel HDF and MPI-IO, parallel I/O in GENx has been greatly simplified from the users’ point of view.

Further, different I/O approaches used in GENx should have uniform user interfaces. Roccom handles this by having each I/O service module provide its own `load_module` and `unload_module` routines. The `load_module` routine creates a window in Roccom, registers a Rocpanda or Rochdf object in the window, and associates user interface functions as the member functions of the object. An application code invokes the I/O operations through `COM_call_function`, which automatically selects the appropriate function, depending on which module is loaded at the beginning of the run. Switching between collective I/O and individual I/O is done by simply loading a different I/O service module.

Finally, modules written in different languages need to collaborate with each other. In GENx, the main driver is written in C, many computation modules are written in Fortran, and the I/O service modules, Rocpanda and Rochdf, are written in C++. Roccom, written in C++ itself, acts as a bridge between these heterogeneous modules. Its interface routines have different bindings for C, C++, and Fortran 90, with similar semantics. Roccom also handles differences between the languages transparently by performing additional processing such as appending null terminators to Fortran strings.

In general, Roccom facilitates communication between simulation modules, for computation and I/O as well. In the context of parallel I/O, it provides simple, high-level interfaces that make users’ lives much easier. Such conveniences are well received by users in CSAR.

6 Maximizing Overlap between I/O and Computation

Data written in the periodic output phases of scientific simulations are often write-only during the run: the simulation itself normally would only read the data back if it is an out-of-core code, and typically there are no other applications running concurrently and accessing the output files. Therefore, users often do not care when their data actually reach the disk, as long as all the data

are recorded at the end of the whole run. Also, as discussed previously, many characteristics of today’s irregular, multi-component simulations make it harder than before to achieve high throughput in file I/O. As a result, our main approach toward providing high-performance parallel I/O is to *hide* the I/O cost, rather than reduce it, by promoting the overlap between I/O and other tasks.

In this paper, we present two schemes for overlapping computation and periodic output, to work with our collective and individual I/O architectures, respectively. Unlike in traditional asynchronous I/O, the overlap in our schemes is *transparent* to users: it works through the simple blocking I/O interface, and enforces the same semantics as blocking I/O, as users can reuse their output buffers immediately after the output function returns.

6.1 Active Buffering in Rocpanda

With the client-server I/O architecture, it is intuitive that the servers can perform the actual write operations while the compute processors compute. We proposed a scheme called *active buffering* to enable this overlap in a flexible and adaptive way. The details of active buffering and related work on overlapping I/O with computation can be found in a previous paper [13]. Here we only give a high-level description of active buffering, and concentrate on its effects for GENx. Basically, during a collective output operation, the servers buffer data rather than write them out. The clients return to computation when all the output data are buffered at the servers. The major benefits of active buffering are as follows:

First, active buffering can use whatever memory available and handles buffer overflow gracefully. The users do not have to worry about whether there is enough buffer space. If buffers overflow, servers automatically write previously buffered data out to make room for incoming data. The full active buffering scheme has a buffer hierarchy on both the clients and servers for larger buffer capacity and better performance [13]. In GENx, only server-side buffering is used because the servers have enough idle memory to hold all the output data with typical client-server configurations.

Second, active buffering maintains responsiveness to clients’ new output requests. The servers write out data when the clients are computing, and check for new client requests between writing two data blocks. This way, the writing of buffered data yields to the handling of new requests. This is especially helpful in multi-component simulations like GENx, where different modules issue back-to-back output requests between relatively long computation phases. To better utilize the idle CPU resources, we used multiple ways to probe for new messages from the clients. When there are data to write, servers use the non-blocking MPI probe interface, so

that if there are no new messages, they can go ahead and write another data block out. However, when there are no data to write, the servers use the blocking probe interface, so that the server processes block until new client messages arrive and the operating system can use the server CPUs, as mentioned in section 4.1.

6.2 Background I/O in Multi-threaded Rochdf

With the server-less architecture, the overlap between computation and I/O is achieved by using a separate thread to carry out the actual writes. We designed and implemented individual I/O with background writing in a multi-threaded version of Rochdf, T-Rochdf. Instead of writing out the data immediately while the callers wait, T-Rochdf allocates local buffers on each compute processor and copies the output data to these buffers. At this point, the main threads return to computation and the I/O thread on each processor writes out the buffered data, which are shared between the two threads.

In T-Rochdf, there is only one I/O thread on each processor. This thread handles all output function calls on the processor. The use of a single persistent thread helps to reduce thread switching overhead and avoids contention among multiple write requests. The main thread buffers all the data output from multiple write requests in the same time-step snapshot, but will block until the I/O thread finishes writing these data out before buffering data for the next snapshot. This is based on the assumption that each processor has enough memory to buffer its local output data for a snapshot, which is true for CPU-intensive simulations like GENx. In practice, GENx’s main threads usually do not have to wait for the I/O threads, because the computation time between two snapshots is normally long enough to write the files.

Like Rocpanda, T-Rochdf enables transparent overlap between computation and I/O. Because the buffering is done locally, the user-visible write cost is even smaller with T-Rochdf than with Rocpanda, and its performance scales better when the number of processors increases. However, it is currently not used on GENx’s production platforms, where Rocpanda offers the extra benefit of combining output into fewer files.

7 Performance Results

7.1 Performance on Development Platform

We first present results from GENx’s primary development platform, the Turing cluster at University of Illinois. It has 208 compute nodes running Linux 2.4, each with dual 1GHz Pentium III processors and 1GB memory, connected with Myrinet. It has shared file system

on RIESERFS, mounted via NFS and accessed through one server. Turing is built for research development, and is not intended for performance benchmarking. It has no job scheduling mechanisms and cannot guarantee a job’s exclusive use of any dual-processor nodes. On the other hand, performance on Turing is important to scientists at CSAR, because they use it intensively and all runs on Turing are interactive. Because Turing’s nodes are shared by multiple concurrent jobs, performance varies significantly from run to run. In this paper, we show the best results measured in five consecutive runs.

We timed test runs of the latest version of GENx, GEN2.5, in which we simulated a lab-scale solid rocket motor, with design and data obtained from the Naval Air Warfare Center. In this test, we partitioned and distributed the same set of simulation data onto different numbers of compute processors, so the total amount of computation and input/output is fixed regardless of the number of processors used. We executed the simulation for 200 time-steps and performed snapshots every 50 time-steps, resulting in five output phases (including the initial snapshot). This output frequency is typical in GENx’s debugging runs. For each snapshots, GENx wrote approximately 64MB of output data. We measured the performance with three I/O implementations described in this paper: Rochdf (non-threaded version, which serves as a base for comparison), T-Rochdf, and Rocpanda. The results obtained with 16, 32, and 64 compute processors are shown in table 1. When Rocpanda is used, extra processors are dedicated as I/O servers and the client-to-server ratio is fixed at 8:1.

compu. proc.		16	32	64
compu. time		846.64	393.05	203.24
visible I/O time	Rochdf	51.58	83.28	51.19
	T-Rochdf	0.38	0.18	0.11
	Rocpanda	2.40	1.48	1.94
restart time	Rochdf	5.33	1.93	0.72
	Rocpanda	69.9	39.2	18.2

Table 1. Computation and I/O times on Turing cluster, in seconds.

The computation time is the total time spent on time-step iterations to update solutions in GENx. In table 1, the computation time is independent of the I/O approach and scales well as the number of compute processors increases. The visible output time is the total time spent in calls to the output interfaces, i.e., the time the compute processors need to wait for the snapshots to be completed. For Rochdf, this is the time to write all the data to HDF files. For T-Rochdf, this is the time to buffer the output data locally. For Rocpanda, this is the time to send the output data to appropriate servers.

For this test case, the idle memory on the compute processors or on the dedicated servers is large enough to buffer all the output data for each snapshot, so T-Rochdf and Rocpanda are both able to hide the actual I/O cost effectively. Compared to the performance of the non-threaded Rochdf, T-Rochdf and Rocpanda dramatically reduce the application-visible output time, as well as considerably reduce the total run time², by actively overlapping I/O with computation. With the non-threaded Rochdf, more than 50 seconds are spent on taking the snapshots, and the performance does not scale up as the number of processors grows. Especially, with 32 processors the write contention between different processors outweighs the decrease in local data size, causing a large increase in the write time. With T-Rochdf, the visible I/O time is almost eliminated, and the performance scales up well. Compared to the non-threaded Rochdf, Rocpanda reduces the visible I/O time by a factor between 21 and 55, and reduces the number of output files by a factor of 8. Rocpanda’s performance does not scale properly, because on Turing, the message passing system does not scale well and the impact of other concurrent jobs grows as more processors are used. This causes increasing communication cost in both the data transfer and the handshaking protocols of Rocpanda’s client-server framework.

Note that both Rochdf and Rocpanda write HDF4 files, and the performance difference shown in table 1 is not the performance difference between HDF and another file format. The non-threaded Rochdf’s performance is the performance that we would expect from a fine-grained, irregular simulation using a general-purpose scientific I/O library that has no asynchronous I/O support, without any performance optimization. T-Rochdf and Rocpanda show that application-specific I/O libraries built on top of the same scientific I/O library can make a huge difference in performance.

We also measured the restart latency of Rochdf and Rocpanda. Since no computation can be overlapped with restart operations, T-Rochdf performs restart in the same way as Rochdf does. As shown in table 1, Rochdf’s restart cost is considerably lower than Rocpanda’s, due to two reasons. First, each Rocpanda restart file contains much more datasets, and as mentioned before, HDF4’s data access performance does not scale well when the number of datasets grows in a file. Second, the NFS-mounted shared file system shows much better tolerance to concurrent reads than to concurrent writes, so Rochdf gains extra I/O parallelism by having all the processors performing reads.

²The total run time is the sum of the total computation time and the total visible I/O time, which grows along with the number of time-steps computed and the number of snapshots taken respectively, plus a bounded initialization and finalization time.

With Roccom’s ability to load the I/O module at runtime, users can choose between T-Rochdf and Rocpanda for debugging runs. If they do not care about the number of files generated, they can use T-Rochdf for better performance. Otherwise, Rocpanda is a natural choice.

7.2 Performance on Production Platform

Next we present results from GENx’s primary production platform, the ASCI Frost. Frost is an IBM SP located at Lawrence Livermore National Laboratory. It has 63 POWER3 375 MHz 16-way SMP compute nodes running AIX 4.3, each with 16GB memory, connected with SP Switch2. The GPFS file system has 20.6TB disk space, accessed through two GPFS server nodes.

To investigate the performance and scalability on Frost, we used GENx’s “scalability” test, which simulates an extendible cylinder of the rocket body. Unlike the lab-scale test used in section 7.1, here the amount of data is fixed on each processor, and the total data size scales with the number of processors. We measure the apparent aggregate write throughput as the number of compute processors increases. Fifteen processors per SMP node are used for computation, and with Rocpanda, one processor per node is used as an I/O server.

On Frost, Rocpanda demonstrates its advantage over Rochdf by both reducing the number of output files and hiding the periodic output costs. Figure 3(a) shows the apparent aggregate write throughput computed by dividing the total output data size by the total visible output cost. With Rocpanda, when the number of compute processors is smaller than 16, only one 16-way SMP node is used, where one processor is assigned to be the I/O server. The increase in Rocpanda’s throughput from 1 compute processor to 15 compute processors is mainly due to higher utilization of the intra-node message passing bandwidth. After the number of compute processors grows past 15, the number of servers starts increasing, and the apparent throughput scales up too. The apparent throughput reaches 875MB/s with a total of 512 processors, more than five times higher than the highest parallel HDF5 throughput with the same number of processors measured on Frost by other researchers [8].

We mentioned earlier that the client-server architecture offers extra benefits in SMP environments. Figure 3(b) demonstrates this effect by showing the computation time with different processor configurations, again with fixed amount of work per compute processor. In the “16NS” case, 16 processors per SMP node are used for computation. In the “15NS” case, 15 processors per node are used for computation and the one processor left is idle. In these two cases, I/O is done through Rochdf. In the “15S” case, 15 processors per node are used for computation and the one processor left on each node is

used as a Rocpanda I/O server. When the total number of compute processors is 8 or less, all the three cases use the same number of compute processors, and the “15S” case uses one extra processor as an I/O server.

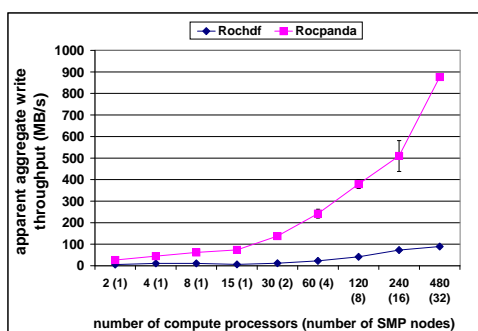
As the number of compute processors grows, the computation time when using all 16 processors per node as compute processors becomes visibly longer than when using only 15 processors. When one processor on each node is used as an I/O server, the computation time is slightly longer than when that processor is left idle, but is considerably shorter than the “16NS” case. Note that with more than 32 processors, the “15S” computation time is less than 15/16 of the “16NS” computation time, while the computation work done in the “15S” case is 15/16 of that in the “16NS” case. This indicates that our client-server I/O architecture not only improves the apparent aggregate I/O throughput, but also improves the aggregate computation throughput on SMPs.

8 Conclusion

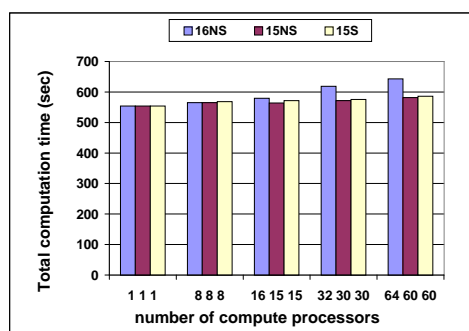
This paper identifies the gap between application-specific I/O requirements and general-purpose parallel interfaces, through analyzing the data management and performance problems presented by GENx, a complex, cutting-edge simulation. By carefully wrapping the underlying I/O services provided by a general-purpose scientific data I/O library with optimizations, and providing high-level parallel I/O interfaces, we have obtained dramatic performance gain while alleviating users’ burden to understand the low-level I/O details and to use efficiently general-purpose parallel I/O libraries. We believe that the real-world I/O challenges that we observed from GENx exist in other large-scale, multi-component simulations, and expect that many of our solutions will apply to those applications as well.

References

- [1] G. Allen, T. Dramlitsch, I. Foster, N. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting efficient execution in heterogeneous distributed computing environments with Cactus and Globus. In *Proc. of SC '01*, Nov. 2001.
- [2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proc. of HPDC*, Nov. 1999.
- [3] R. Bordawekar, J. Rosario, and A. Choudhary. Design and evaluation of primitives for parallel I/O. In *Proc. of SC '93*, Nov. 1993.



(a) Apparent aggregate write throughput in GENx



(b) Computation time using different number of processors per SMP node

Figure 3. Performance results on Frost. Results are averaged over three experiments and the error bars show the 95% confidence intervals. Computation time on Frost has little variance.

- [4] B. Broom, R. Fowler, and K. Kennedy. KelpIO: A telescope-ready domain-specific I/O library for irregular block-structured applications. In *Proc. of CCGrid*, May 2001.
- [5] P. Crandall, R. Aydt, A. Chien, and D. Reed. Input/output characteristics of scalable parallel applications. In *Proc. of SC '95*, Nov. 1995.
- [6] W. Dick and M. Heath. Whole system simulation of solid propellant rockets. In *Proc. of the 38th AIAA/ASME/SAE/ASEE Joint Propulsion Conference and Exhibit*, Jul. 2002.
- [7] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proc. of SC '93*, Nov. 1993.
- [8] http://flash.uchicago.edu/~zingale/flash_benchmark_io. *FLASH I/O Benchmark Routine*.
- [9] <http://hdf.ncsa.uiuc.edu/UG41r3.html/>. *HDF 4.1r3 User's Guide*.
- [10] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proc. of OSDI*, Nov. 1994.
- [11] D. Kotz. Expanding the potential for disk-directed I/O. In *Proc. of SPDP*, Oct. 1995.
- [12] D. Kotz and N. Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proc. of SC '94*, Nov. 1994.
- [13] X. Ma, M. Winslett, J. Lee, and S. Yu. Faster collective output through active buffering. In *Proc. of IPDPS*, Apr. 2002.
- [14] J. Nieplocha, I. Foster, and R. Kendall. Chemio: High-performance parallel I/O for computational chemistry applications. *The Intl. Journal of Supercomputer Applications and High Performance Computing*, 12(3):345–363, 1998.
- [15] J. No, S. Park, J. Carretero, and A. Choudhary. Design and implementation of a parallel I/O runtime system for irregular applications. *Journal of Parallel and Distributed Computing*, 62(2):193–220, 2002.
- [16] J. No, R. Thakur, and A. Choudhary. Integrating parallel file I/O and database support for high-performance scientific data management. In *Proc. of SC '00*, Nov. 2000.
- [17] J. No, R. Thakur, D. Kaushik, L. Freitag, and A. Choudhary. A scientific data management system for irregular applications. In *Proc. of the 8th Intl. Workshop on Solving Irregularly Structured Problems in Parallel*, Apr. 2001.
- [18] J. Reynders et al. Pooma: A framework for scientific simulations on parallel architectures. In G. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 553–594, 1996.
- [19] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proc. of SC '95*, Nov. 1995.
- [20] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proc. of IOPADS*, May 1999.