Parallel I/O Performance for Application-Level Checkpointing on the Blue Gene/P System

Jing Fu,[†] Misun Min,[§] Robert Latham,[§] Christopher D. Carothers[†]

[†]*Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180*
{fuj,chrisc}@cs.rpi.edu

[§]*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439*
{mmin,robl}@mcs.anl.gov

***Abstract*—As the number of processors increases to hundreds of thousands in parallel computer architectures, the failure probability rises correspondingly, making fault tolerance a highly important and challenging task. *Application-level checkpointing* is one of the most popular techniques to proactively deal with unexpected failures because of its portability and flexibility. During the checkpoint phase, the local states of the computation spread across thousands of processors are saved to stable storage. Unfortunately, this approach results in heavy I/O load and can cause an I/O bottleneck in a massively parallel system. In this paper, we examine application-level checkpointing for a massively parallel electromagnetic solver system called *NekCEM* on the IBM Blue Gene/P at Argonne National Laboratory. We discuss an application-level, two-phase I/O approach, called "reduced-blocking I/O" (rbIO), and a tuned MPI-IO collective approach (coIO), and we demonstrate their performance advantage over the "1 POSIX file per processor" approach. Our study shows that rbIO and coIO result in 100× improvement over previous checkpointing approaches on up to 65,536 processors of the Blue Gene/P using the GPFS. Our study also demonstrates a 25× production performance improvement for NekCEM. We show how to optimize parameter settings for those parallel I/O approaches and to verify results by I/O profilings. In particular, we examine the performance advantage of rbIO and demonstrate the potential benefits of this approach over the traditional MPI-IO routine, coIO.**

Keywords **Parallel I/O, checkpointing, fault tolerance, Blue Gene/P**

I. Introduction

As current leadership-class computing systems such as the IBM Blue Gene series [1] move closer to exascale capability, the likelihood of an unrecoverable node or network failure is high [2]. When a component fails, the application in progress loses valuable work and must be restarted, thereby wasting computing time, power, and staff effort.

Another trend in current petascale systems is that they share a high degree of hardware, including memory and caches within nodes, network infrastructure between nodes, and a shared storage I/O system for the whole machine. During the checkpointing phase, gigabytes or even terabytes of checkpoint data from hundreds of thousands of processors can be written into the shared back-end storage system, making I/O a bottleneck. In extreme cases, traditional I/O approaches such as "1 POSIX file per processor" (1PFPP) for checkpointing on a 128K-processor partition render poor performance or even lock the file system and hang the application until it is removed from the job queue [3]. Thus, scalable and effective I/O approaches are needed so that users can better utilize the computing cycles allocated on massively parallel systems, yielding more productive science per compute cycle.

The key contribution of this paper is a performance study of different parallel I/O approaches applied to application-level checkpointing for a production petascale electromagnetics solver NekCEM (Nekton for Computational ElectroMagnetics) [4]. In particular, we implement an application-level, two-phase I/O approach called "reduced-blocking I/O" (rbIO), and a tuned MPI-IO collective approach (coIO). We demonstrate their performance advantage against previously used 1PFPP approach. Our objective is to provide an approach that reduces the checkpointing time, gives users more flexibility with checkpoint-restart data files, and provides guidance for further I/O performance tuning on different systems.

This paper is organized as follows. In Section II, we provide a general overview of checkpointing and some work that has been done in this area. In Section III-A, we introduce the petascale application code, NekCEM, used in our study. In Section IV, we discuss several parallel I/O approaches. In Section V, we describe the Blue Gene/P system, compare different approaches, and provide detailed experiment results. In Section VI, we compare our approaches with related work in the literature. In Section VII, we give our conclusions and discuss some future work.

II. Overview of Checkpointing

Application checkpoint/restart is an effective fault tolerance technique in distributed systems. Checkpoint/restart allows a program to save local states periodically so that, in the event of a system crash, the program can roll back to the most recently saved state, avoiding total loss of work. This technique is especially important for those computational science and engineering applications (e.g., parallel partitioned solvers) that normally iterate for many steps and require a long time to complete. Checkpointing can happen either at the operating system level or at the application level.

System-level checkpointing typically provides checkpointing in an user-transparent manner, where the data is managed by the operating system and checkpointing can happen at any time. While this approach requires no additional effort from the application programmer and sees the application as a black box, none of the internal semantics or characteristics of the application are recognized. Thus, the whole state of the computation—including CPU register information and memory information—must be stored during each checkpointing. This approach dramatically increases the total amount of data to be stored, especially on large-scale systems. Examples of this approach include IODC [5] , which will be reviewed in Section VI.

Assume that we have 150 MB of data in memory for each processor for checkpointing. A 65,536-processor partition will generate roughly 10 TB data at each checkpoint step, which is too heavy for a typical shared-I/O subsystem in such large parallel systems. Moreover, because system-level checkpointing records a snapshot for a specific system (e.g., register information, software stacks, memory layout), it is not portable between different platforms.

On the other hand, although *application-level checkpointing* requires more manual effort from an application programmer, it takes the content and semantics of an application into consideration; the application programmer decides which critical data needs to be stored to disk. The application programmer also has the freedom to choose a safe time and appropriate frequency for checkpointing. Since these checkpoint data files are user-defined, they are easily ported to different platforms. Also, these files can be used for other purposes, such as data visualization or other postprocessing analysis, which are extremely useful for many computational applications. Examples of this approach include ADIOS [6] and data partitioning techniques [7], [8], which are reviewed in Section VI.

In this paper, we focus on application-level checkpointing. Specifically, our applications involve checkpointing certain data in a *coordinated* manner, where all processors start and end checkpointing synchronously. (Throughout the paper, we use the term "processors" to mean "cores.") In such a situation, no processor begins the next iteration until the last processor completes its checkpointing, and thus any significant I/O latency on a single processor can result in keeping all other processors in the partition waiting. Our main motivation in developing efficient parallel I/O approaches is to balance the I/O latency among all processors and reduce the overhead or even completely hide the I/O latency by using dedicated I/O communicators in the optimal case.

III. Software and I/O File Format

We consider the production code NekCEM, which is a single, comprehensive electromagnetic software package, currently capable of scalable simulations up to more than 131K processors on leadership-class machines such as the IBM Blue Gene/P. In this section, we describe the key features of NekCEM including some of its capabilities.

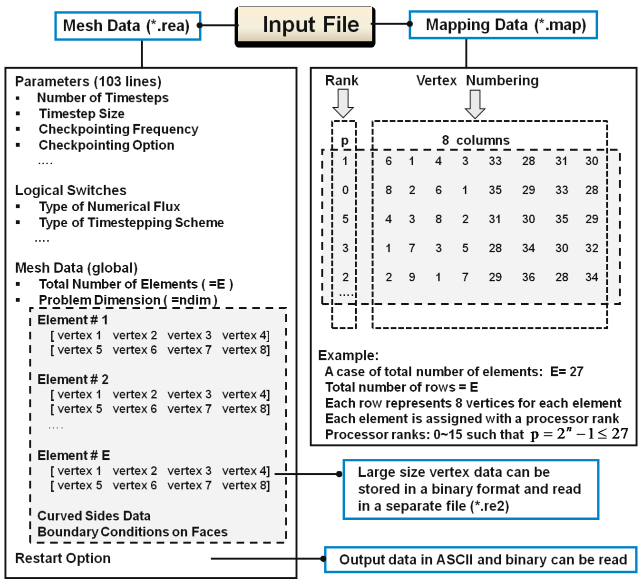
A. NekCEM

Highly efficient and accurate modeling on advanced computing platforms will enable the relevant science and engineering communities to advance their understanding of complex systems that are too large for experimental study and will reduce both the cost and the risk involved in conventional trial-and-error procedures.

NekCEM is an Argonne-developed, high-order, spectral-element discontinuous Galerkin (SEDG) code [4] designed for simulation-based investigations for understanding the fundamental optical properties and predicting optimal designs of electromagnetic devices in particle accelerator physics and nanoscience applications [9]. This code features spectrally accurate solutions with less numerical dispersion for long time simulations with geometric flexibility using body-fitted conforming meshes [10].

NekCEM solves the two- and three-dimensional Maxwell curl equations in the time domain. Spectral-element discretizations are used based on hexahedral element meshes. For the time-advancing, the code currently supports explicit time-stepping schemes such as the five-stage, fourth-order Runge-Kutta [11] and the exponential time integration methods [12]. Tensor product bases of the one-dimensional Lagrange interpolation polynomials using the Gauss-Lobatto-Legendre grid points result in a diagonal mass matrix, which requires no additional cost for mass matrix inversion [13], making the code highly efficient. The stiffness matrix is a tensor product form of the one-dimensional differentiation matrix [13].

The discontinuous Galerkin scheme based on the domain decomposition approach performs communication only at the element faces (excluding the information of vertices and edges) between neighboring elements through a numerical flux [14]. The face values at the interfaces are saved in a single array for the six components of the electric field $E{=}(E_x, E_y, E_z)$ and the magnetic field



Input File
Mesh Data (*.rea)
Mapping Data (*.map)
Parameters (103 lines)
Number of Timesteps
Timestep Size
Checkpointing Frequency
Checkpointing Option
Logical Switches
Type of Numerical Flux
Type of Timestepping Scheme
Mesh Data (global)
Total Number of Elements (=E)
Problem Dimension (=ndim)
Element # 1
[vertex 1 vertex 2 vertex 3 vertex 4]
[vertex 5 vertex 6 vertex 7 vertex 8]
Element # 2
[vertex 1 vertex 2 vertex 3 vertex 4]
[vertex 5 vertex 6 vertex 7 vertex 8]
Element # E
[vertex 1 vertex 2 vertex 3 vertex 4]
[vertex 5 vertex 6 vertex 7 vertex 8]
Curved Sides Data
Boundary Conditions on Faces
Restart Option
Rank
Vertex Numbering
p
8 columns
1 6 1 4 3 33 28 31 30
0 8 2 6 1 35 29 33 28
5 4 3 8 2 31 30 35 29
3 1 7 3 5 28 34 30 32
2 2 9 1 7 29 36 28 34
Example:
A case of total number of elements: E= 27
Total number of rows = E
Each row represents 8 vertices for each element
Each element is assigned with a processor rank
Processor ranks: 0~15 such that $p = 2^n - 1 \leq 27$
Large size vertex data can be stored in a binary format and read in a separate file (*.re2)
Output data in ASCII and binary can be read

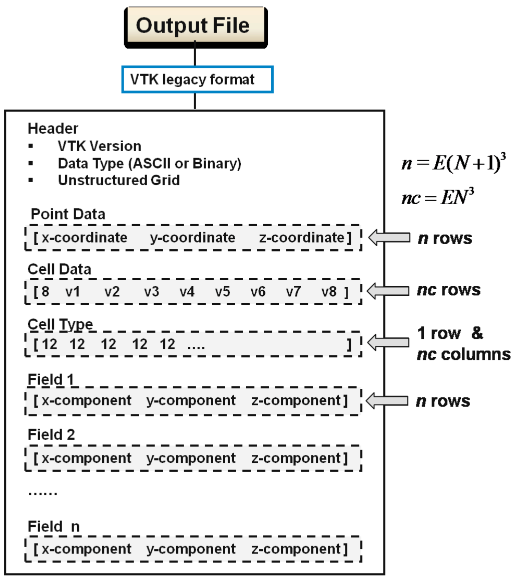
Fig. 1. Input files for NekCEM.

$H{=}(H_x, H_y, H_z)$ so that communication can occur only once at each time step between neighboring elements. Thus, communication latency can be reduced by a factor of six compared to the case of saving the face values into six different arrays for each component of the fields.

NekCEM is written in Fortran and C. The code uses the core infrastructure of the incompressible Navier-Stokes solver Nek5000, awarded the Gordon Bell prize in 1999 [15]. NekCEM uses the distributed-memory message-passing interface (MPI) programming model and the single program, multidata model (SPMD) so that each processor independently executes a copy of the same program on distinct subsets of data.

NekCEM has an instruction for the following three tasks that are performed consecutively at run time: *presetup, solver*, and *checkpointing*. Presetup includes initialization of processors, setting compile-time data sizes, reading run-time parameters and global mesh data from input files, distributing mesh data to each processor, and assigning numbering for nodal points and coordinates for a geometry. Solver involves the SEDG spatial operator evaluation and time iterations. Checkpointing generates output files for the global field data computed from the solver; these files can be used for restarting.

Parallel performance and scalability of NekCEM are discussed in [16], showing strong scaling performance of three-dimensional cylindrical waveguide simulations using the number of elements E=136K and E=273K with the polynomial approximation orders N=5 and N=15, resulting in a total number of grid points n from 29 million to 1.1 billion. As the number of grid points per processor increases, the efficiency increases. Specifically, NekCEM achieves 75% efficiency on 131,072 processors for n/P=8,530 (still not a large amount) grid points per processor, compared to a base case on 16,384 processors for n/P=68,250 grid points per processor. Even with an unrealistically small number of grid points per processor n/P=224, NekCEM achieves 58% efficiency on 131,072 processors, compared to a base case on 16,384 processors for n/P=1,793 grid points per processor. CPU time per time step is $\approx$0.13 seconds on 131K processors for the case of E=273K with 1.1 billion grid points.



Output File
VTK legacy format
Header
VTK Version
Data Type (ASCII or Binary)
Unstructured Grid
Point Data
[x-coordinate y-coordinate z-coordinate]
Cell Data
[8 v1 v2 v3 v4 v5 v6 v7 v8]
Cell Type
[12 12 12 12 12 ....]
Field 1
[x-component y-component z-component]
Field 2
[x-component y-component z-component]
......
Field n
[x-component y-component z-component]
$n = E(N+1)^3$
$nc = EN^3$
n rows
nc rows
1 row & nc columns
n rows

Fig. 2. Output file format for NekCEM.

B. I/O File Format

NekCEM has two input data files, as shown in Figure 1, providing the information on global mesh (*.rea) data and global mapping (*.map) for vertices including processor distribution for each element. For simplicity, data files are kept in global format so that users are not required to deal with mesh partition before compile/runs with easier management for many different mesh configurations. Input files are autogenerated from meshing tools such as prex and genmap [15], which are also included in the NekCEM package [4].

Data files are read at the very beginning stage, before the actual solver runs. Reading the global data for a mesh takes from 7.5 seconds to 28 seconds, with the numbers of element E=136K and 546K on P=32,768 and 131,072 processors of BG/P, respectively. Since read only occurs once during the whole execution, our optimization focus would be on the more frequent write operations.

Figure 2 shows the structure of an output file of NekCEM. A self-defined output format generally provides users more flexibility, although users have to maintain a special reader interface for data analysis tools. An open file format such as *vtk legacy* can be directly read by postprocessing tools for visualization using ParaView or VisIt. NekCEM uses the vtk legacy format, where the master header includes the application name, file type (binary or ASCII), application type, grid point coordinates, cell numbering, and cell type.

Since NeKCEM checkpoints the local state on each processor in a *coordinated* manner, the output files share a similar structure as well: every file has a master header followed by data blocks. The *master header* typically specifies metadata information such as application name, version, local state list, and offset table. The *data blocks* are sorted mostly in the order of fields. In each data block, there is a header recording metadata such as data block size and field name. The data block contains the actual values of the field from the NekCEM computation.

This "bursty" I/O access of NekCEM is a very typical pattern for many computational science applications [17], especially parallel partitioned solvers. In our previous paper [3], we did an extensive amount of *strong scaling* tests on tuning different parallel I/O techniques for a CFD application to get best raw I/O bandwidth. In this paper, we apply optimal configurations of these techniques

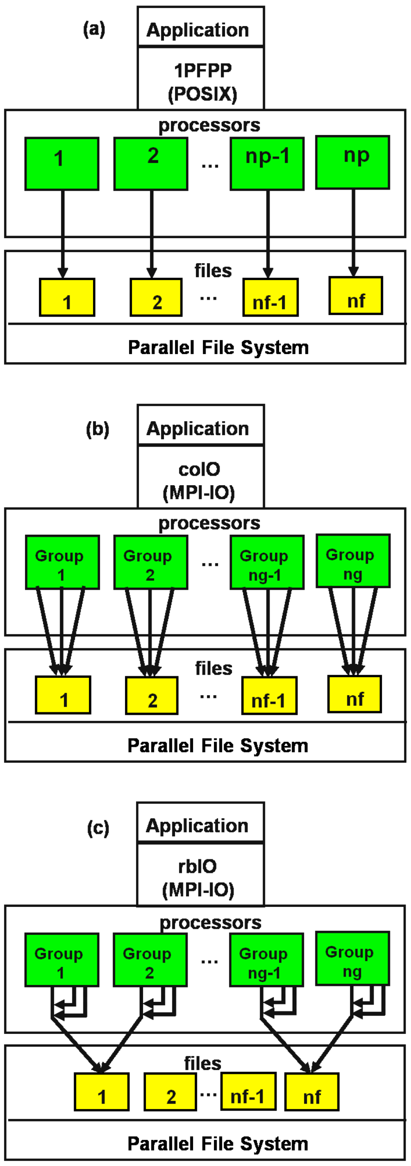


Fig. 3. Architecture diagrams for different I/O approaches: (a) 1PFPP, (b) coIO, and (c) rbIO.

to NekCEM and focus on NekCEM (*weak scaling*) production performance improvement and potential performance benefit of rbIO optimizations.

IV. Parallel I/O Approaches

In this section, we discuss three I/O approaches, namely, 1 POSIX File Per Processor, collective I/O, and reduced-blocking I/O, for generating checkpointing files. An architecture diagram for each approach is given in Figure 3. On a fixed number of processors (***np***), each approach specifies the number of *group* (***ng***) processors that access the file system and the number of output files (***nf***) generated by those *writer* processors.

A. 1 POSIX File Per Processor: 1PFPP

With the 1PFPP approach, all processors access a file system, and each processor generates one file output (i.e., ***np***=***nf***) based on the traditional POSIX I/O. POSIX stands for portable operating system interface for Unix [18]. It is a standard application programming interface that defines the I/O interface for standard services on the operating system using functions such as $open()$, $read()$, $write()$, and $close()$. POSIX I/O has a simple access capability that was designed originally for a single machine with a single memory space to a streaming device. Because of its well-defined semantics and portability, 1PFPP has been widely used by many parallel application codes for decades. However, since the semantics of this interface prevent the sharing of file descriptors between nodes and each processor has to write its own file output, performance becomes very limited when shared storage is accessed by hundreds of thousands processors simultaneously, generating huge overheads from excessive metadata traffic, disk block locking, and so on.

B. Collective I/O: coIO

The collective I/O approach is based on the MPI-IO library, which defines a set of routines that transfer data to and from external storage. This approach offers a number of advantages over the traditional POSIX I/O approach. MPI-IO provides mechanisms for collective access (i.e., many processors collectively read and write to shared file descriptors), asynchronous I/O, and strided data access. Many platforms support the MPI-IO interface. Applications run essentially unchanged, and the files written by MPI-IO are portable between platforms.

Our implementation uses ROMIO [19] which is a high-performance, portable implementation of the MPI-IO library. ROMIO is supported by many platforms including IBM SP, Cray T3E, NEC SX-4, and SGI Origin2000 with variable file systems such as PVFS, IBM PIOFS, NEC SFS, and SGI XFS. Details can be found in the parallel I/O chapter of MPI-2 [20].

ROMIO is optimized for collective I/O and noncontiguous access patterns, which are important in parallel I/O. In our implementation, all processors call the collective I/O routine to write data to a number of files. The number of output files, typically ***nf***$=2^m <$ ***np***, m=0,1,2,..., is a user-tunable parameter. If ***nf***=1, all processors in `MPI_COMM_WORLD` follow the instructions

call `MPI_File_write_at_all_begin()`,

call `MPI_File_write_at_all_end()`,

which are collective nonblocking functions to write all data into one shared file. If ***nf***>1, the processors are divided evenly into ***nf*** (=***ng***) groups, and the ***np***/***nf*** processors in each group (i.e., local MPI communicator) collectively write to one file in parallel. Collective routines involve only processes that open the file and thus generally perform better than noncollective ones since smaller requests would be merged into fewer bigger requests when a group of processes coordinate. Nonblocking routines initiate a request and return immediately, provided with a request identifier that will be completed by calling an additional routine. This allows for I/O operations to be overlapped with computation or communication.

C. Reduced-Blocking I/O: rbIO

The reduced-blocking I/O approach also utilizes the MPI-IO library. This approach divides compute processors into two categories: workers (application compute node) and writers (I/O aggregator node). Among ***np*** processors, only ***ng*** groups (or ***ng*** writers) are allowed to access the file system. In each group, one writer node takes charge of the other application compute nodes in the group. When the workers want to write data to a disk, they send their data to their dedicated writer (i.e., the processor at the left side of each "*Group*" as indicated in Figure 3(c)) with

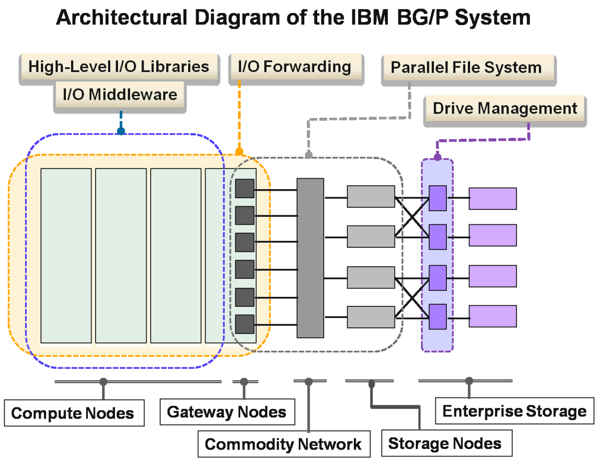


Fig. 4. Architectural diagram of the 557 TFlop IBM Blue Gene/P system.

call `MPI_Isend()`

and return from this nonblocking call quickly, without any interruption for I/O. The writer aggregates the data from all workers in its group, reorders data blocks, and writes to disk using either a noncollective function

call `MPI_File_write_at()`

or a collective nonblocking function set

call `MPI_File_write_at_all_begin()`,

call `MPI_File_write_at_all_end()`.

`MPI_File_write_at()` uses the `MPI_COMM_SELF` communicator to produce one file per writer. `MPI_File_write_at_all_begin()` and `MPI_File_write_at_all_end()` produce ***nf*** files, where ***nf*** can be between 1 and ***ng***. The number of writers (***ng***) and number of files (***nf***) are both user-tunable parameters in the I/O subsystem; in our case we set ***nf***=***ng***.

V. Performance and Analysis

In this section, we describe experiment test cases and demonstrate I/O performance of the different parallel I/O approaches that we implemented in NekCEM. We carry out the performance tests using the GPFS [21] file system on the Blue Gene/P "Intrepid" at Argonne National Laboratory.

A. Overview of the Blue Gene/P System

Blue Gene's philosophy is to balance the computing power of a massive number of processors with fast network that interconnect processors and other auxiliary networks for I/O and so forth. The Intrepid system has 40 racks of 40,960 quad-core compute nodes (with a total of 163,840 processors) and 80 TB of memory, with a peak performance of 557 TF. There are five networks in BG/P system: a 3-D torus network that interconnects compute nodes (point-to-point network), a collective network that connects compute nodes and I/O nodes, a tree-based collective network for barrier operations, a Gigabit Ethernet that delivers data between I/O nodes and data storage serves, and a JTAG network that connects compute nodes to service nodes. The bandwidth for each torus link is 425 MBps per direction and 5.1 GBps bidirectional bandwidth per node [1]. BG/P compute nodes run a special lightweight kernel called the compute node kernel (CNK); I/O nodes run a different kernel that has more functions than CNK. In order to improve the scalability of the Blue Gene architecture, dedicated I/O nodes (IONs) act as system call proxies between the compute nodes and the storage nodes; the collection of an ION and its compute nodes called a "pset". Each BG/P pset contains one ION and 64 4-core compute nodes; in total, 640 IONs are connected to the parallel storage system.

The parallel storage system consists of 16 Data Direct Network (DDN) 9900 SAN storage arrays, where each DDN exports the disk block as LUNs. Each LUN is directly connected to 8 file servers, and there are 128 file servers in total. These servers are shared by the file systems, GPFS [21] and PVFS [22]. They are connected to I/O nodes via 10 Gigabit Ethernet. The theoretical reading peak is 60 GB/s and that of writing is 47 GB/s [22].

We note that the file systems are shared between Intrepid, Eureka (a visualization system), and some other clusters whose I/O workload may affect the I/O performance observed on Intrepid. Also, all our tests were done under normal load, where there might be noise from other online users.

An architectural diagram of Intrepid is shown in Figure 4.

B. Parallel I/O Performance for NekCEM

We implemented two parallel I/O approaches in NekCEM and performed weak scaling on the Argonne Blue Gene/P. We carried out 3D cylindrical waveguide simulations for different sizes of meshes and different numberss of processors with (E, P)=$(68K, 16K)$, $(137K, 32K)$, and $(273K, 65K)$, where E is the number of elements for a mesh and P is the number of processors. The total number of grid points is n=$E(N + 1)^3$, where N is the approximation order. The number of grid points per processor is approximately n/P. We used N=15 so that the number of grid points per element is fixed at 16^3. The total numbers of grid points n and the file sizes S are (n, S)=(275M,39GB), (550M,78GB), and (1.1B,156GB) per I/O step.

We investigated the write bandwidth for different I/O approaches for NekCEM. The bandwidth was measured as the total amount of data across all processors divided by overall wall-clock time (including *open*, *write*, and *close* file) of the slowest processor to finish. Note that most of these experiments were run *multiple times* and the data points were sampled from the ***median number***.

Figure 5 shows the write bandwidth as a function of the number of processors. As expected from previous experience [3], 1PFPP renders poor performance with 16K and 32K processors because of a high number of metadata operations. This is caused by generating all output files ***nf***=***np*** in one directory. Better performance may be achieved by producing a single file per directory. However, most parallel file systems are not designed to deal with hundreds of thousands of small files, and manageability becomes a significant issue.

With collective routines, all processes in a given MPI communicator call the routine together. The MPI implementation, with the knowledge of which tasks participate in a call, can then perform significant optimizations. These collective routines provide tremendous performance benefits for both networking and I/O [23]. The Blue Gene MPI-IO library makes some adjustments to the ROMIO collective buffering optimization [24]. First, data accesses are aligned to file system block boundaries. Such an alignment reduces lock contention in the write case and can yield big performance improvements [25]. Second, and perhaps most important from a scalability perspective, the "I/O aggregators" are a small subset of the total number of processors. On Blue Gene, the MPI-IO hint "bgp_nodes_pset" defines

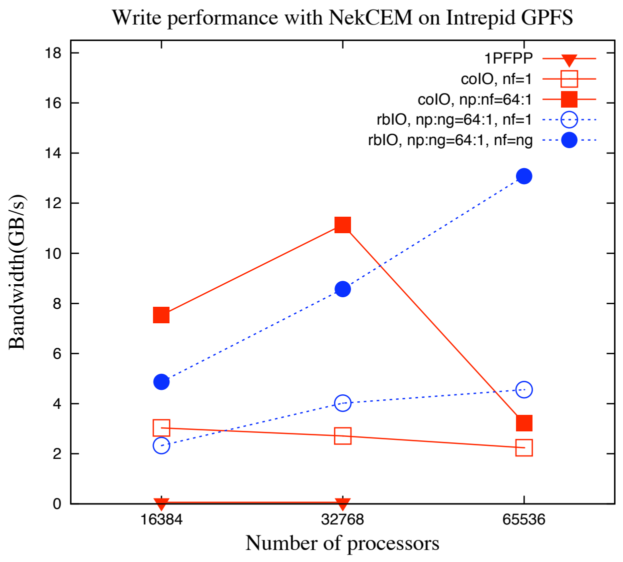


Fig. 5. Write performance of different I/O approaches in NekCEM on GPFS file systems of Intrepid, as a function of processor number for the problem sizes of (np, n, S)= (16K,275M,39GB), (32K,550M,78GB), and (64K,1.1B,156GB).

a ratio. For each pset allocated to a parallel job, that many nodes will be designated as aggregators. The default ratio for a job running in "virtual node" is one aggregator for every 32 MPI processes. Furthermore, these aggregators are distributed over the topology of the application so that no node has more than one aggregator and no pset contains more than "bgp_nodes_pset" aggregators. In Figure 5, we show the case of one I/O aggregator per every 64 MPI processes (***np:ng=64:1***). The performance comparison study for 64:1, 32:1, and 16:1 for rbIO is discussed later in this section.

For coIO, we examine two cases: ***nf=1*** and ***np:nf = 64:1***. For ***nf=1***, all processors in `MPI_COMM_WORLD` will call MPI-IO collective routines in a synchronized manner. For ***np:nf=64:1***, all processors are evenly divided into ***np/64*** groups (split collective). These 64 processors (i.e., one from each group) write its data into one common file in a synchronized manner using an MPI-IO collective routine, but between groups these function calls are independent of each other. Figure 5 shows that the bandwidth for ***np:nf=64:1*** case is larger than the one for ***nf=1*** on 16K, 32K and 64K processors. The case with ***nf=1*** experiences a file locking overhead where all processors need communication and synchronization during write operations, while the case of ***np:nf=64:1*** independent collective writers experiences less locking overhead.

For rbIO, we tested the cases of ***nf=1*** and ***np:ng=64:1***. The rbIO writers use independent `MPI_File_write_at()` for the case of ***nf=ng***, with the `MPI_COMM_SELF` communicator because writing is not performed across all processors. For ***nf=1***, rbIO issues a collective routines file. In both cases, workers send a data package only to their corresponding writer in a group with `MPI_Isend()`. The performance in Figure 5 shows that, for rbIO, the bandwidth for ***nf=ng*** is two times better than the case of ***nf=1***, as a result of less file locking overhead. In addition, the writers utlized the buffer in a more efficient manner, yielding a lower I/O flush rate. Recall that in the file format, each file is written by fields (to maintain grid points numbering consistency in a file scope); thus, for ***nf=1*** writers must commit each field to disk before processing the next field, while for ***nf=ng*** writers can keep multiple fields until the buffer is full before

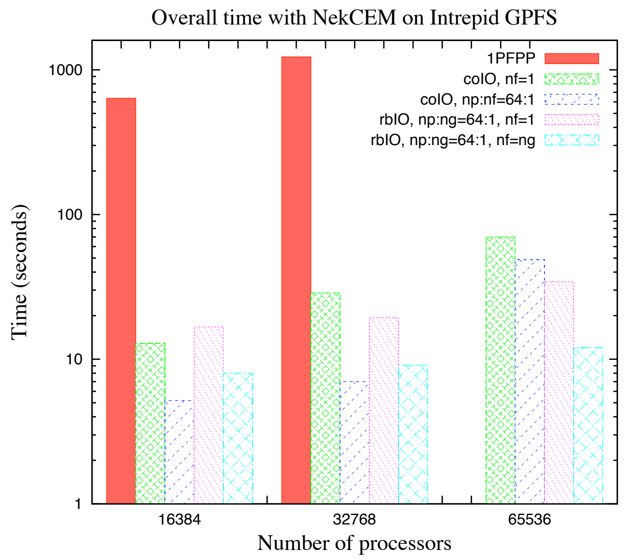


Fig. 6. Overall time per checkpointing step for different I/O approaches in NekCEM on GPFS on Intrepid, as a function of processor number for the problem sizes of (np, n, S)=(16K,275M,39GB), (32K,550M,78GB), and (64K,1.1B,156GB).

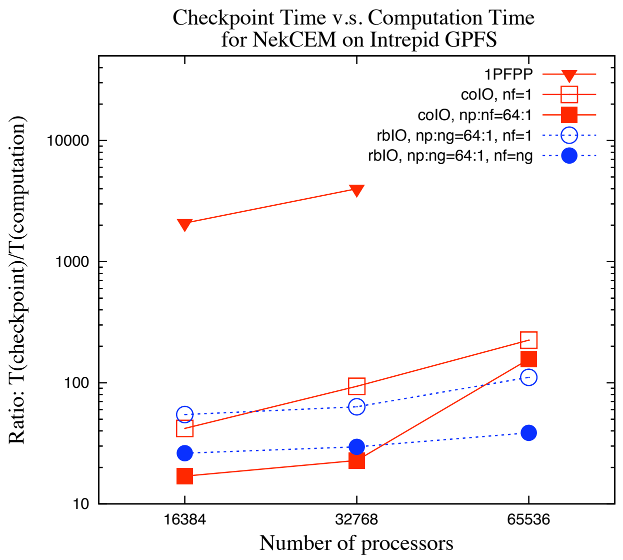


Fig. 7. Ratio of checkpoint time per I/O step over computation time per single time step for different I/O approaches of NekCEM on Intrepid. A smaller ratio of an I/O approach represents a shorter I/O time per single computation step for the problem sizes of (np, n, S)=(16K,275M,39GB), (32K,550M,78GB), and (64K,1.1B,156GB).

committing data to disk (similar to internal data sieving optimization in MPI-IO collectives). In both cases of coIO, all the processors all commit data by fields.

The performance of ***nf=1*** for coIO and rbIO is similar, demonstrating that the application's two-phase optimization does not interfere with MPI-IO's two-phase optimization. We note that rbIO in the case of ***nf=ng*** performs no worse than coIO in the case of ***np:nf** = 64:1* at larger scale. This implies that if the application manages its own data aggregation carefully with the other optimization techniques, it is possible to get comparable raw I/O bandwidth without blocking the majority of processors (63/64 = 98%) in I/O operations. We further discuss potential performance gains in Section V-C2. We notice a significant performance drop for coIO with ***np:nf** = 64:1* at 64k processors and will explain this later from I/O time distribution in Figure 10.

Figure 6 demonstrates the overall time per checkpointing step for different I/O approaches. It shows a significant reduction in time with the rbIO and cIO parallel I/O approaches compared with the 1PFPP approach. The relatively flat time bars for rbIO also implies that rbIO scales well on up to 65K processors, compared with those by other approaches.

Figure 7 shows the ratio of checkpoint time over computation time for different I/O approaches. Note that NekCEM's computational performance scales well on Intrepid so the computation time is almost the same for 16K, 32K, and 64K processors. We observe that the ratio for rbIO stays flat when compared with others, meaning that it is more scalable than other approaches. Denoting the checkpoint frequency by n_c, one can express the production time improvement as

$$\frac{T_c^{\text{1pfpp}} + n_c * T_{\text{comp}}}{T_c^{\text{rbIO}} + n_c * T_{comp}} = \frac{\text{Ratio}^{\text{1pfpp}} + n_c}{\text{Ratio}^{\text{rbIO}} + n_c}, \quad (1)$$

where T_c is the checkpoint time and T_{comp} is the computation time. For n_c=20, Ratio$^{\text{1pfpp}}$ is generally above 1000 while Ratio$^{\text{rbIO}}$ is under 20. Thus, the end-to-end performance of rbIO over 1PFPP achieves approximately 25× improvement.

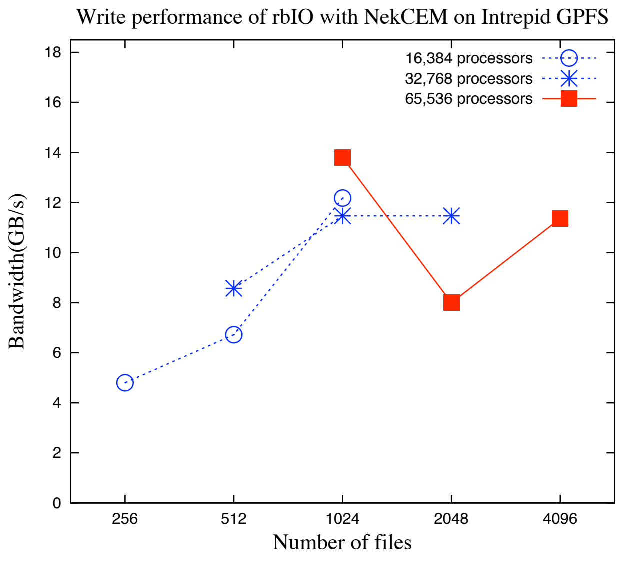


Fig. 8. Performance of rbIO (with nf=ng) as a function of number of files under different numbers of processors on GPFS on Intrepid.

It has been shown in Figure 5 that rbIO with ***nf=ng*** scales well. Figure 8 provides a closer look by varying the number of files ***nf*** on different numbers of processors ***np*** and showing the write performance of rbIO with ***nf=ng*** at different ***np:ng*** ratios. We observe that better performance at different scales occurs for the case of ***nf***=1024. This indicates that the file system has a preference for having larger numbers of files written into the file system concurrently. In Figure 5, performance is poor when this number is too small (***nf=1***) or too big (***nf=np***). For the GPFS file system deployed on Intrepid, this number stays around 1,024 when running on 16K, 32K, and 64K processors. This optimal number could vary from one file system to another, as

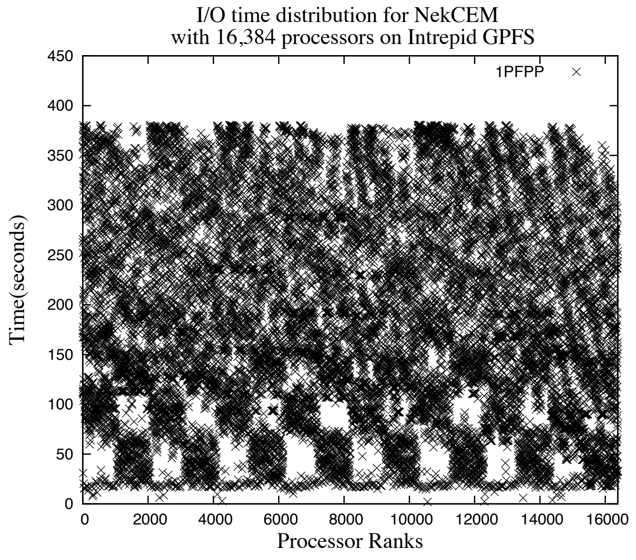


Fig. 9. I/O time distribution among 16,384 processors for one checkpointing step, with 1PFPP in NekCEM on GPFS of Intrepid.

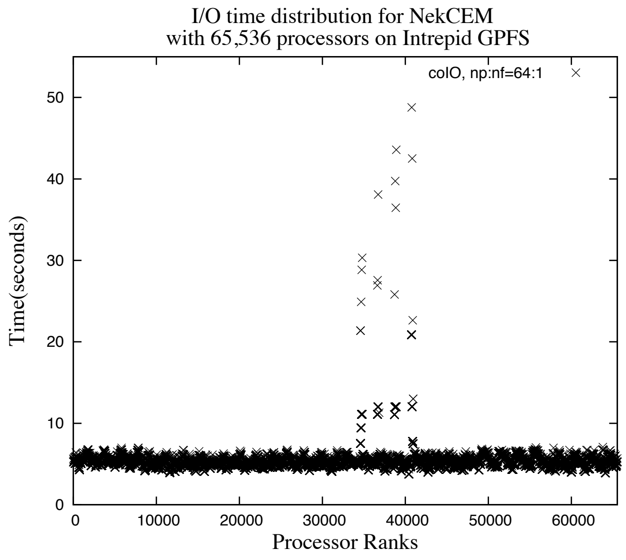


Fig. 10. I/O time distribution among 65,536 processors for one checkpointing step, with coIO in NekCEM on GPFS of Intrepid.

we previously showed a different optimal number for the GPFS on a Blue Gene/L at different scales [3].

C. I/O Time Distribution and Speedup Analysis

1) I/O Time Distribution on the GPFS System: Figure 9 shows the I/O time distribution with a 16,384-processor partition for 1PFPP. Each '×' point in the graph stands for a timing value for a certain processor. Clearly, some processors finish I/O within several seconds, while others take more than 300 seconds. Heavy metadata access (request to create, write, and close 16,384 small files simultaneously) to the file system can be a reason for the high degree of variance in per processor I/O performance.

Figure 10 shows the I/O time distribution with a 65,536-processor partition for coIO. Note the y-axis range is much smaller than in the 1PFPP case and thus is more synchronized. Several outliers spend longer than the average time to finish their writing part. Because of these outliers (caused by noise and/or other factors under normal user load), although most of the processors finish within 10 seconds, the performance was hindered since all the other processors had to wait for the slowest processor to finish.

TABLE I
PERCEIVED WRITE PERFORMANCE WITH RBIO FOR NEKCEM UNDER 16K, 32K, AND 64K PROCESSORS ON THE INTREPID.

# Procs	Time (CPU cycle)	Bandwidth (TB/s)
16K	10152	251
32K	11539	442
64K	9346	1091

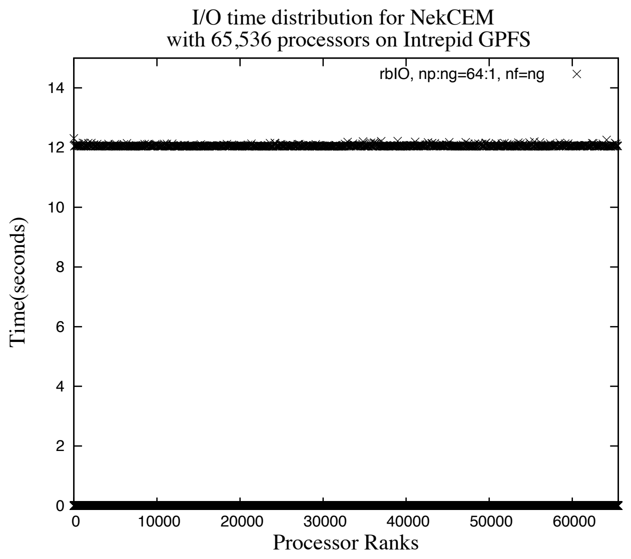


Fig. 11. I/O time distribution among 65,536 processors for one checkpointing step, with rbIO in NekCEM on GPFS on Intrepid.

Figure 11 shows the I/O time distribution with a 65,536-processor partition for rbIO. There appears to be two "lines" in this plot. The upper line represents the finish time for writers, while the lower one is the finish time plot for workers. Most workers spent a small amount of time sending their data to writers. We also notice that the upper line for writers is almost flat, showing good synchronization even though these writers use the independent `MPI_File_write_at()` routine to commit data to disk. One reason can be that in this ***np:ng=64:1*** case, the file system access concurrency is only 50% of the concurrency in the coIO case (MPI-IO default ratio is 32:1).

Table I shows the average perceived write performance with rbIO for NekCEM on 16K, 32K, and 64K processors on Intrepid. The notion of *perceived* writing speed is defined as *the speed at which worker processors can transfer their data.* This measure is calculated as the total amount of data sent across all workers over the maximum time that `MPI_Isend()` takes to complete. So rbIO leverages the Blue Gene/P's internal memory bandwidth and powerful torus network to transfer I/O data to the writer. From a worker's perspective, this operation finishes when returning from non-blocking `MPI_Isend()`, which significantly cuts application execution time.

Figure 12 shows the Darshan [26] log analysis for rbIO and coIO. Clearly, the performance of the two approaches is not significantly different. However, because coIO's writing activity is not as synchronized, we can see there are lock contentions in collective writes, even though the group size is relatively small and the total file number is

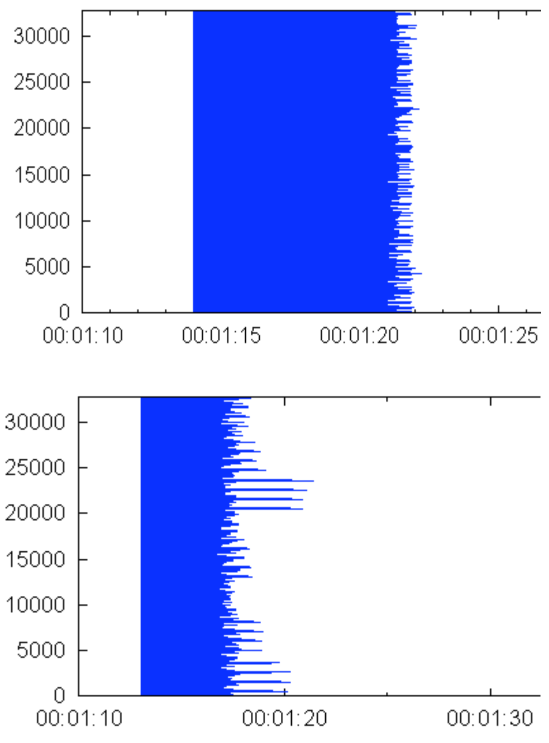


Fig. 12. Write activity of rbIO with nf = ng (top) and coIO with np:ng = 64:1 (bottom) in the 32K processor case.

not too large. From Figure 5 we can confirm that coIO is not as scalable as rbIO.

Initially we investigated the performance characteristics of these I/O configurations on PVFS as well and intended to compare GPFS performance with lock-free PVFS. However, at the time we ran these tests, significant hardware configuration differences, e.g. cache was (and still is) turned off on PVFS, make the comparision very weak and pointless.

2) Speedup Analysis: Here, we analyze the speedup of rbIO over coIO by computing the overall I/O block time spent on each processor:

$$Speedup = \frac{T_{coIO}}{T_{rbIO}}. \quad (2)$$

The total time of all processors blocked by I/O operations, T_{coIO} and T_{rbIO}, can be defined by

$$T_{coIO} = np\frac{S}{BW_{coIO}}, \quad (3)$$

$$T_{rbIO} = (np - ng)(\frac{S}{BW_p} + \frac{\lambda S}{BW_{rbIO}}) + ng\frac{S}{BW_{rbIO}}, \quad (4)$$

where BW_{coIO}, BW_{rbIO}, and BW_p represent the bandwidths of coIO, rbIO, and perceived write speed, respectively, and λ is the percentage of writer's write time that workers are blocked for and S is the file size. Then we have

$$\frac{T_{coIO}}{T_{rbIO}} = \frac{np\frac{S}{BW_{coIO}}}{(np-ng)(\frac{S}{BW_p} + \frac{\lambda S}{BW_{rbIO}}) + ng\frac{S}{BW_{rbIO}}}, \quad (5)$$

where $\frac{np-ng}{np} \approx 1$ and $\frac{BW_{coIO}}{BW_p}\frac{np-ng}{np}$ is typically a very small number (e.g., $BW_{coIO}/BW_p \approx 10^{-6}$ for the test cases in this paper). This leads to

$$Speedup \approx \frac{1}{(\lambda + \frac{ng}{np}(1-\lambda))(\frac{BW_{coIO}}{BW_{rbIO}})}. \quad (6)$$

In our NekCEM case, the writers can flush their I/O requests roughly in the time between writes, which is O(10) seconds. In this case, λ is a very small number, close to zero, which yields a speedup roughly

$$\left(\frac{np}{ng}\right)\left(\frac{BW_{rbIO}}{BW_{coIO}}\right). \quad (7)$$

BW_{rbIO} and BW_{coIO} are close to each other, as shown in the previous graphs. Even in the worst case where BW_{rbIO} is roughly half of BW_{coIO}, the speedup is still half of the ratio (i.e., 30×).

VI. Related Work

Our previous work showed initial results for rbIO and coIO on a 32K processor Blue Gene/L for checkpointing a computational fluid dynamics solver system [3]. In that work, we demonstrated 2.3 GB/s write bandwidth and 21 TB/s perceived bandwidth and explored tuning possibilities.

Lofstead et al. [6] designed a portable metadata-rich I/O architecture (called ADIOS), which chooses between I/O libraries for application programmer with minimal effort. The authors demonstrate performance improvement on up to 8K processors on the Cray Jaguar system at ORNL. Both their work and ours optimize I/O in the application layer. However, their main focus is to provide ease for generic application programmers, whereas our focus is to analyze the detailed I/O pattern from "bursty" I/O applications and leverage techniques within MPI-IO to maximize application performance.

Choudhary et al. [7] used several file domain partitioning techniques to improve collective I/O performance on the Cray XT4 and on clusters. These techniques include aligning partitions to file system lock boundaries and using I/O aggregators. The techniques were applied to both GPFS and Lustre using I/O benchmark tests and up to 14 GB/s bandwidth with 2,048 processors were observed. However, I/O bandwidth does not increase with processor count after 2K.

Nisar et al. [5] developed an I/O delegate and caching system (IODC) that aggregates data by using 10% of processors as I/O aggregators below the MPI-IO layer and improved I/O benchmarks using up to 400 processors. Their approach requires thread support from OS, which is a limiting factor on supercomputers. Our rbIO differs from this approach in that rbIO works on the application level with very good scalability and is easy to implement.

Dickens et al. [8] reported poor MPI-IO performance on Lustre, with an improved I/O performance by redistributing data. Yu et al. [27] characterized several I/O benchmarks on Jaguar using the Lustre filesystem and demonstrating the efficacy of their I/O tuning approaches, such as the two-phase collective I/O. Shan et al. [28] analyzed the disk access patterns for I/O-intensive applications at the NERSC and selected parameters for IOR benchmarks [29] to emulate the application behavior and overall workloads. Fahey et al. [30] also investigated four I/O approaches (MPI I/O, agg, ser, and swp) performing subsetting experiments on the Cray XT4 on up to 12,288 processors, achieving a write performance of about 40% of peak. The rbIO and coIO approaches allow the number of files to be a tunable parameter. In their subsetting benchmark, however, only a single file output approach was considered.

Borrill et al. [31] investigated file I/O performance for the MADbench2 benchmark on different systems, such as Lustre on Cray and GPFS on Blue Gene/L. They demonstrated I/O performance depending on concurrency, I/O library and file number, based on simulation up to 1K processors.

Another benchmark study of large-scale parallel I/O was conducted by Lang et al. [22]. They used the Blue Gene/P at Argonne and demonstrated I/O bandwidths of nearly 60 GB/s (read) and 47 GB/s (write) using up to 131K processors, using benchmarks including BTIO and MADbench2 etc. on PVFS. However, the researchers did not explore the mechanisms of nonblocking I/O for applications, which is our main focus in this paper.

The Scalable Checkpoint Restart (SCR) library [32] provides a multi-level checkpointing capability that can leverage local node storage in the form of RAM disk or SSD. The developers report a checkpoint performance speedup over writing to a parallel filesystem of betwen 14x and 234x for the pF3D benchmark using up to 8K cores. A current barrier to using SCR, however, is that it requires a compute-side OS that is RAM disk capable; the Blue Gene/P compute node kernel is not. This barrier will disappear as future leadership computing systems provide more full-featured OS capabilities.

VII. Conclusions and Future Work

We demonstrate how rbIO and coIO can improve checkpoint performance for a data-intensive scientific application compared with traditional POSIX I/O approaches. We also show significant production performance improvement using these new parallel I/O techniques. CoIO uses MPI-IO optimized collective calls to do smart user-transparent buffering in MPI-IO layer and achieve good performance and scalability when finely tuned. By utilizing the fast point-to-point torus network to aggregate writes and reduce file system access concurrency, rbIO can use application-level, two-phase I/O to achieve improved performance and better scalability with over 13 GB/s write speed on 65K processors. We discuss how to select parameters on a specific machine in order to get the best performance by examining I/O log data from both user profiling and system profiling. We observe a significant performance improvement for splitting writers from workers in rbIO.

We plan to investigate how rbIO performs on platforms such as the Cray XT with other file systems such as Lustre and how to tune rbIO to obtain the best performance.

VIII. Acknowledgments

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357, while Jing Fu worked as a Givens Associate at the Mathematics and Computer Science Division at Argonne National Laboratory. Christopher Carothers was supported by the NSF PetaApps Program, OCI-0749152. Additionally, the authors acknowledge the INCITE allocations from the Nek5000 project led by Paul Fischer, as well as the computing resources and staff efforts provided by the Argonne Leadership Computing Facility.

References

[1] IBM Blue Gene team, "Overview of the IBM Blue Gene/P project," IBM Journal of Research and Development Volume 52 Issue 1/2, 2008.

[2] DARPA, "DARPA ExaScale Software Study: Software Challenges in Extreme Scale System," http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ecss%20report%20101909.pdf, September 2009.

[3] J. Fu, N. Liu, O. Sahni, K. Jansen, M. Shephard, and C. Carothers, "Scalable parallel I/O alternatives for massively parallel partitioned solver systems," in *LSPP workshop of IPDPS*, April 2010.

[4] "NekCEM: Computational ElectroMagnetic Code," https://svn.mcs.anl.gov/repos/NEKCEM.

[5] A. Nisar, W. Liao, and A. Choudhary, "Scaling parallel I/O performance through I/O delegate and caching system," in *Supercomputing*, 2008.

[6] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan, "Adaptable, metadata rich IO methods for portable high performance IO," in *IPDPS 09*, Washington, DC, USA, 2009.

[7] A. Choudhary, W. Liao, K. Gao, A. Nisar, R. Ross, R. Thakur, and R. Latham, "Scalable I/O and analytics," *Journal of Physics: Conference Series*, vol. 180, no. 1, 2009.

[8] P. M. Dickens and J. Logan, "Y-lib: A user level library to increase the performance of MPI-IO in a Lustre file system environment," in *HPDC*, 2009.

[9] M. S. Min, T. W. Lee, P. F. Fischer, and S. K. Gray, "Fourier spectral simulations and gegenbauer reconstructions for electromagnetic waves in the presence of a metal nanoparticle," *Journal of Computational Physics*, vol. 213, no. 2, pp. 730–747, 2006.

[10] J. Hesthaven, S. Gottlieb, and D. Gottlieb, *Spectral methods for time-dependent problems,* Volume 21 of Cambridge monographs on applied and computational mathematics. Cambridge University Press, 2007.

[11] M. Carpenter and C. Kennedy, "Fourth-order $2N$-storage Runge-Kutta schemes," *NASA Report TM 109112*, 1994.

[12] E. Gallopoulos and Y. Saad, "Efficient solution of parabolic equations by Krylov approximation methods," *SIAM J. Sci. Stat. Comput.*, vol. 13, no. 5, pp. 1936–1964, 1992.

[13] M. Deville, P. Fischer, and E. Mund, *High-order methods for incompressible fluid flow,* Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2002.

[14] J. Hesthaven and T. Warburton, *Nodal discontinuous Galerkin methods, algorithms, analysis, and applications.* Springer, 2008.

[15] "Nek5000: Computational Fluid Dynamics Code," https://svn.mcs.anl.gov/repos/nek5000.

[16] M. S. Min and J. Fu, "Performance analysis on the IBM BG/P for the spectral-element discontinuous Galerkin method for electromagnetic modeling," 2010, MCS, ANL, Preprint ANL/MCS-P1802-1010.

[17] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Storage access characteristics of computational science applications," in *Proceedings of 27th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2011.

[18] IEEE/ANSI Std. 1003.1, "Portable operating system interface (POSIX)–Part 1: System application program interface (API) [C language]," 1996.

[19] R. Thakur, E. Lusk, and W. Gropp, "Users guide for ROMIO: A high-performance, portable MPI-IO implementation," 2004, technical Memorandum ANL/MCS-TM-234, MCS, ANL.

[20] "Message Passing Interface," http://www.mcs.anl.gov/research/projects/mpi.

[21] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *FAST*, 2002.

[22] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *Supercomputing*, November 2009.

[23] R. Thakur, W. Gropp, and E. Lusk, "Optimizing noncontiguous accesses in MPI-IO," *Parallel Computing*, vol. 28, no. 1, pp. 83–105, 2002.

[24] H. Yu, R. K. Sahoo, C. Howson, G. Almasi, J. G. Castanos, M. Gupta, J. E. Moreira, J. J. Parker, T. E. Engelsiepen, R. Ross, R. Thakur, R. Latham, and W. D. Gropp, "High performance file I/O for the BlueGene/L supercomputer," in *HPCA-12*, February 2006.

[25] W. Liao and A. Choudhary, "Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols," in *Supercomputing, Austin, Texas*, November 2008.

[26] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale i/o workloads," in *Workshop on Interfaces and Architectures for Scientific Data Storage*, September 2009.

[27] W. Yu, J. S. Vetter, and S. Oral, "Performance characterization and optimization of parallel I/O on the Cray XT," in *IPDPS*, 2008.

[28] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark," in *Supercomputing*, 2008.

[29] "IOR Benchmark," https://asc.llnl.gov/sequoia/benchmarks/#ior.

[30] M. R. Fahey, J. M. Larkin, and J. Adams, "I/O performance on a massively parallel Cray XT3/XT4," in *IPDPS*, 2008, pp. 1–12.

[31] J. Borrill, L. Oliker, J. Shalf, H. Shan, and A. Uselton, "HPC global file system performance analysis using a scientific-application derived benchmark," *Parallel Computing*, vol. 35, no. 6, pp. 358–373, 2009.

[32] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Supercomputing, New Orleans*, November 2010.