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

Jing Fu
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180
fuj@cs.rpi.edu

Misun Min
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
mmin@mcs.anl.gov

Robert Latham
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
robl@mcs.anl.gov

Christopher D. Carothers
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180
chrisc@cs.rpi.edu

## ABSTRACT

As the number of processors increases to hundreds of thousands in the recent parallel computer architectures, the failure probability rises correspondingly, making fault tolerance a highly important yet 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 electromagnetics 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

## 1. INTRODUCTION

As current leadership-class computing systems such as the IBM Blue Gene series [?] move closer to exascale capability, the likelihood of an unrecoverable node or network failure is high [?]. 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 [?]. 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 Electro-Magnetics) [?]. In particular, we developed a 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 2, we provide a general overview of checkpointing and some work that has been done in this area. In Section 3.1, we introduce the petascale application code, NekCEM, used in our study. In Section 4, we discuss several parallel I/O approaches. In Section 5, we describe the Blue Gene/P system, compare different approaches, and provide detailed experiment results. In Section 6, we compare our approaches with related work in the literature. In Section 7, we give our conclusions and discuss some future work.

## 2. 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 [?] and SCR [?], which are reviewed in Section 6.

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 [?] and data partitioning techniques [?, ?], which are reviewed in Section 6.

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.

## 3. 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.

### 3.1 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 [?] that is 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 [?, ?]. This code features spectrally accurate solutions with less numerical dispersion for long time simulations with geometric flexibility using body-fitted conforming meshes [?].

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 [?] and the exponential time integration methods [?]. 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 [?], making the code highly efficient. The stiffness matrix is a tensor product form of the one-dimensional differentiation matrix [?].

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 [?]. 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 $H{=}(H_x, H_y, H_z)$ 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 [?]. NekCEM uses the distributed-memory message-passing interface (MPI) programming model [?] and the single program, multidata