Understanding Checkpointing Overheads on Massive-Scale Systems: Analysis on the IBM Blue Gene/P System

Rinku Gupta, Harish Naik and Pete Beckman

Mathematics and Computer Science Division

Argonne National Laboratory

{*rgupta, hnaik, beckman*}*@mcs.anl.gov*

Abbreviated Title : **CheckPointing Trends On IBM BGP**

Author Listing:

Rinku Gupta

Mathematics and Computer Science Department

Argonne National Laboratory

9700 South Cass Avenue, Argonne, IL 60439

Phone number: 630-252-6266

Email: rgupta@mcs.anl.gov

Harish Naik

Mathematics and Computer Science Department

Argonne National Laboratory

9700 South Cass Avenue, Argonne, IL 60434

Phone number: 630-252-2029

Email: hnaik@mcs.anl.gov

Pete Beckman

Mathematics and Computer Science Department

Argonne National Laboratory

9700 South Cass Avenue, Argonne, IL 60439

Phone number: 630-252-9020

Email: beckman@mcs.anl.gov

Queries should be addressed to Rinku Gupta

LIST OF CHANGES

This paper is an extended version of the paper presented at the International Workshop on Parallel Programming Models and Systems Software for High-End Computing Systems (P2S2), 2009, titled "Analyzing Checkpointing Trends for Applications on the IBM Blue Gene/P System.". In this section, we highlight some of the additions done in this version of the paper, as compared to the one presented at the workshop (several other minor changes and explanation details added in the paper are skipped here):

Additional Applications: Together with the different applications demonstrated in the workshop version of the paper, this version also includes a new Computational Fluid Dynamics application, the Parallel Ocean Program (POP), that is a popular application for Ocean modeling. POP is an integral part of the SPEC 2007 benchmark suite, and represents a wide-class of applications.

Additional Experimentation and Analysis: This paper also presents several additional results, including experiments on larger system sizes and more analysis of the time taken for checkpointing, evaluation of checkpointing parameters, and other trends,

Theoretical Modeling: The paper also presents a new simplified optimal checkpointing model and uses this model to evaluate checkpointing parameters for all the different applications.

Abstract

Providing fault tolerance in high-end peta-scale systems, consisting of millions of hardware components and complex software stacks, has becoming an increasingly challenging task. Checkpointing continues to be the most prevalent technique for providing fault tolerance in such high-end systems. Considerable research has focussed on optimizing checkpointing; however, in practice, checkpointing still involves a high-cost overhead for users. In this paper, we study checkpointing overhead seen by various applications running on leadership-class machines like the IBM Blue Gene/P at Argonne National Laboratory. In addition to studying popular applications, we design a methodology to help users understand and intelligently choose an optimal checkpointing frequency to reduce the overall checkpointing overhead incurred. In particular, we study the Grid-Based Projector-Augmented Wave application, the Carr-Parrinello Molecular Dynamics application, the Nek 500 Computational Fluid Dynamics application and the Parallel Ocean Program application — and analyze their memory usage and possible checkpointing trends on 65,536 processors of the Blue Gene/P system. Keywords: Checkpointing, Blue Gene, Fault Tolerance, I/O, Massive scale systems

I. Introduction

The past two decades have seen tremendous growth in the scale, complexity, functionality, and usage of high-end computing (HEC) machines. The Top500 [1] list shows that performance offered by high-end systems has increased by over eight times in the past five years. Current petascale machines consist of hundreds of thousands of hardware components and complex software stacks. Future exascale systems will exponentially increase this complexity, resulting in systems with hundreds of millions of hardware components. As failure rates increase with the hardware and software component count and software complexity, large-scale faults become unavoidable. This situation has led to fault tolerance re-emerging as a prominent issue for these systems.

Various methods for fault tolerance have been researched in the recent past. Fault tolerance at the hardware level has been achieved mostly through redundancy or replication of resources. Examples of this approach span from physical redundancy (RAIDed disks [2], backup servers) to information redundancy (ECC memory codes, parity memory). From the vasts amounts of research devoted to software fault tolerance, checkpointing [3] has become a popular method for achieving fault tolerance in the software. Checkpointing requires saving the local state of a process at a specific time and then rolling back (recovery) to the latest saved state in the event of a crash during the execution lifetime of a process. Since the checkpointing process impacts the overall process running time, researchers have sought ways to optimize checkpointing [4] [5] [6].

New petascale machines, such as the IBM Blue Gene series [7], offer enormous computational power. Large-scale applications using these machines continue to use checkpointing for proactive fault tolerance. Saving states of thousands of processes during checkpoint can result in a heavy demand of I/O and network resources. Since these machines have limited I/O and network resources, frequent checkpoint, at this scale, can result in longer execution times, especially if the I/O or network resources become a bottleneck. For applications to use these machines in the most efficient manner, we must understand the feasibility of checkpointing and memory trends of these applications on petascale machines. While measurement and analysis of memory trends have received some study, investigations have been limited to small-scale systems of up to 64 processes [8]. Challenges for checkpointing exhibited by large leadership machines are completely different and on a different scale.

In this paper, we study similar checkpointing trends for applications on the IBM Blue Gene/P system at Argonne National Laboratory . Specifically, we present memory trends of four popular applications: Grid-Based Projector-Augmented Wave application (GPAW), Carr-Parrinello Molecular Dynamics application (CPMD), Nek5000 computational fluid dynamics application and the Parallel Ocean Program (POP). We also present an analytical model for efficiently computing the optimum checkpoint frequencies and intervals, and we determine the limitations of checkpointing such applications on large systems. Our study and analysis are based on a “full checkpointing” technique, in which the entire program state of the processes is stored during the checkpoint operation. This technique is used by the IBM Checkpointing library [9] and is the only checkpointing software currently available for the IBM Blue Gene/P machines.

Our study was conducted on 65,536 cores of the Argonne BG/P system. In the future, we plan to run the applications with full checkpointing as well as "incremental checkpointing" on all 163,840 nodes of the BG/P system.

The rest of the paper is organized as follows. Section II discusses the background and related work performed in the area of checkpointing. Section III provides an overview of the IBM Blue Gene/P system and its checkpointing software. Section IV presents the applications used in this study. Section V discusses our checkpointing model, experiments performed and analysis of the results. In Section VI, we present the conclusions and outline the future work.

II. Overview of Checkpointing

Checkpointing methods and optimization techniques have been studied and summarized by several researchers [10] [11] [12]. In this section, we briefly discuss some checkpointing concepts and techniques common to distributed computing environments. We will also discuss work done in the area of checkpointing in MPI programs [13] on large scale High Performance Computing(HPC) systems. In distributed systems, checkpointing can typically occur at the operating system level or the application level.

Operating system-level checkpointing: Operating system (OS)-level checkpointing is a user-transparent way of implementing checkpointing. In this approach, the user typically needs to specify only the checkpointing interval, with no additional programming effort; other details such as checkpoint contents are handled by the operating system [14]. OS-level checkpointing for an application involves saving the entire state of the application, inclusive of all processes and temporary data, at the checkpoint time. Since this type of checkpointing does not take into account the internal characteristics and semantics of the application, the total size of the checkpointing data can dramatically increase with system size. On petascale systems, which are I/O bound, this can cause a heavy overhead on the runtime of the application.

Application-level checkpointing: Application-level checkpointing [12], also called user-defined checkpointing, is a checkpointing method that enables the user to intelligently decide the placement and contents of the checkpointing. The primary advantage of this approach is that users can semantically understand the nuances of the applications and place checkpoints, using libraries and preprocessors, at critical areas, potentially decreasing the size of the checkpoint contents and checkpointing time. While this approach requires more programmer effort, it is more portable, since checkpoints can be saved in a machine-independent format, and thus offers better performance and flexibility as compared to the OS-level approach.

An important difference between the two types of checkpointing is that an OS-level checkpoint can be performed at any point during the execution of the application. But application-level checkpoints can on be performed at specified points in the program during the execution [13].

Compiler-level checkpointing [15] also exists. In this case, the compiler selects optimal checkpointing locations to minimize checkpointing content. Hybrid techniques, such as compiler assisted checkpointing coupled with application checkpointing, also have been studied. These provide a certain degree of transparency to the user.

Checkpointing in distributed systems requires that global consistency be maintained across the entire system and the

domino effect be avoided. One way to achieve this consistency is through *coordinated checkpointing* [16]. In coordinated checkpointing, once the decision to checkpoint is made, the program does not progress unless all the checkpoints of all the processes are saved. Coordinated checkpointing requires that system or process components communicate with each other to establish checkpoint start and end times in single or multiple phases. Recovery in this technique is achieved by rolling back all processes to the latest state. Another method to achieve global consistency is through *independent checkpointing* [12], which uses synchronous and asynchronous logging of interprocess messages along with independent process checkpointing. In this method, recovery is achieved by rolling back to the faulty process and replaying the messages received by the faulty process. Other techniques such as *adaptive independent checkpointing* [12] are based on hybrid coordinated and independent checkpointing techniques.

Checkpointing optimizations have received considerable attention. Two primary techniques that have emerged are *full-memory checkpointing* and *incremental-memory checkpointing* [17]. In full-memory checkpointing, during each checkpoint instance the entire memory context for that process is saved. In incremental-memory checkpointing, pages that have been modified since the last checkpoint are marked as dirty pages and are saved. Incremental-memory checkpointing thus can reduce the amount of memory context that needs to be saved, especially for large systems.

Performing checkpointing operations on large scale systems becomes very challenging in context of large scale distributed memory systems. Study [18] by Liu et. al. proposes a checkpoint and recovery model for large scale HPC systems. There have also been studies on checkpointing and recovery mechanisms applied to Network of Workstations(NOW) [19]. But in a system like Blue Gene/P with 163840 cores available for computation, the challenges are far more severe.

III. Overview of the Blue Gene/P System

In this section, we discuss the architecture of the IBM Blue Gene/P [7] system at Argonne National Laboratory named *Intrepid*. We also provide an overview of the IBM Blue Gene/P checkpointing library.

A. The BG/P Hardware Architecture

The *Intrepid* [20] system at Argonne National Laboratory is IBM's massively parallel supercomputer consisting of 40 racks of 40,960 quad-core compute nodes (totaling 163,840 processors), 80 TB of memory and a system peak performance of 556 TF. Each compute node consists of four PowerPC 450 processors, operates at 0.85 GHz, and has 2 GB memory. Compute nodes run a lightweight Compute Node Kernel (CNK), which serves as the base operating system. In addition to the compute nodes, 640 input/output (I/O) nodes are used to communicate with the file system. The I/O nodes physically are the same as compute nodes but differ from them in functional. The Argonne system is configured to have a 1:64 ratio with a single I/O node managing 64 compute nodes. In addition to the compute and I/O nodes, there exist service and login nodes that allow diagnostics, compilation, scheduling, interactive activities, and administrative tasks to be carried out. This system architecture is depicted in Figure 1[1].

[1]This figure was taken from the 'IBM System Blue Gene/P Solution: Blue Gene/P Application Development' redbook [9]

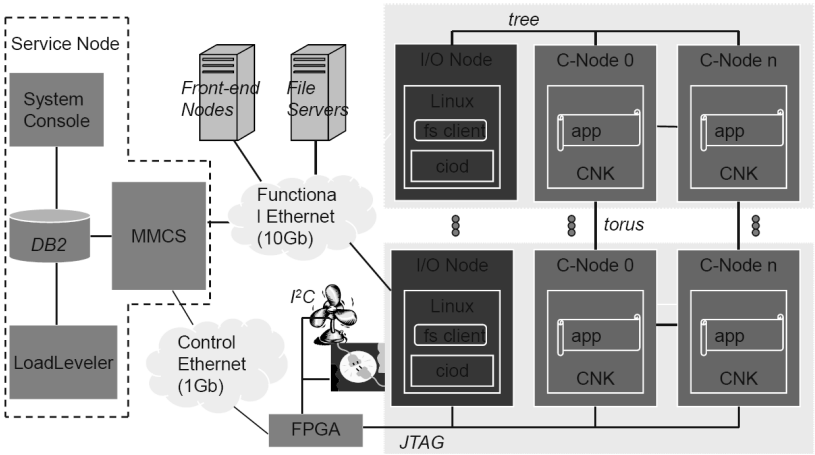

Figure 1. The ANL BG/P System Architecture

The Intrepid system supports five different networks. A three-dimensional torus network, based on adaptive cut-through routing, interconnects the compute nodes and carries the bulk of the communication while providing low-latency, high bandwidth point-to-point messaging. A collective network interconnects all the compute nodes and I/O nodes. This network is used for broadcasting data and forwarding file-system traffic to I/O nodes. An independent, tree-based, latency-optimized barrier network also exists for fast barrier collective operations. In addition, a dedicated Gigabit Ethernet and JTAG network that connects the I/O nodes and compute nodes to the service nodes is used for diagnostics, debugging and monitoring. Lastly, a 10-Gigabit Ethernet network, consisting of Myricom switching gear connected in a nonblocking configuration, provides connectivity between the I/O nodes, file servers, and several other storage resources.

As seen in Figure 2, the Intrepid back-end file system architecture includes 16 DataDirect 9900 SAN storage arrays (each with 480 1 TB disks) offering around 8 petabytes of raw storage. Each array is directly connected to 8 file servers through 8 Infiniband DDR ports, each with a theoretical unidirectional bandwidth of 16 Gbps. The entire Intrepid system consists of 128 dual-core file servers, each having 8 GB memory each. These file servers, as earlier noted, connect to the 10-Gigabit Ethernet network through their 10-Gigabit Ethernet ports. The I/O nodes connect to this 10-Gigabit Ethernet network as well. We note that the peak unidirectional bandwidth of each 10-Gigabit Ethernet port of the I/O node is *limited to 6.8 Gb/s* by the internal collective network that feeds it [7]. From a theoretical performance standpoint, however, the network links connecting the file servers to the 10-Gigabit Ethernet network will become a bottleneck.

B. The IBM BG/P Checkpointing Library Overview

IBM provides a special user-level *full checkpointing* library [9] for BG/P applications. This library provides support for *user-initiated checkpointing*; where a user can insert checkpoint calls manually at critical areas in the application. Restarting an application is transparent and can be easily achieved by the user or system by setting certain environment variables (which point to the checkpoint file to be used for restart) during application re-launch time. The checkpoint calls are made available

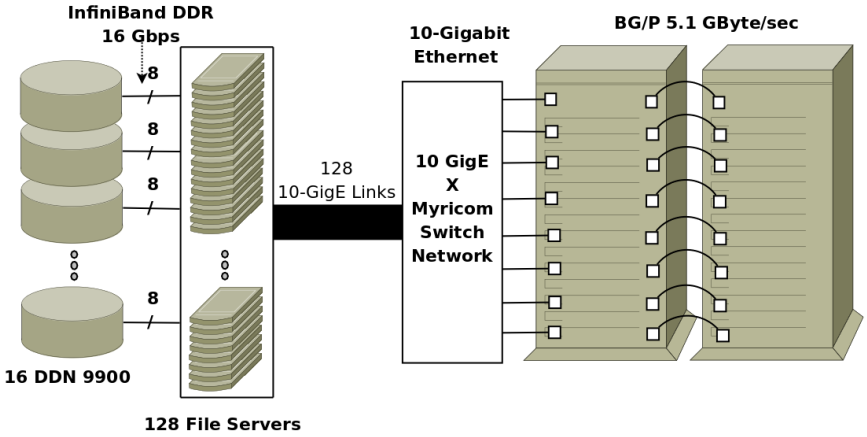

Figure 2. The ANL BG/P File System

to the user through a simple checkpointing API provided by IBM. Checkpoint-enabled applications are expected to call the *int BGCheckpointInit(char *path_to_checkpoint_directory)* routine at the beginning of the application. This routine initializes all the relevant data structures and is also used for the transparent restart of the applications when they are re-launched. The *int BGCheckpoint()* library call is used to take a snapshot/checkpoint of the program state at the instant at which it is called. All processes of the application should make this call to take a consistent global checkpoint. The user needs to ensure that no outstanding messages are present in the system when this checkpoint call is being made. It is also recommended that a Message Passing Interface (MPI) collective operation routine such as barrier synchronization [21] be called just before the *BGCheckpoint()*.

In addition to the above routines, IBM provides various other routines which allows applications to perform various functionalities such as registering functions that can be called just before checkpointing (*int BGAtCheckpoint(args..)*), registering functions that can be called when continuing after a checkpoint (*int BGAtContinue(args..)*) or marking regions that can be excluded from the program state when the checkpoint is taken (*int BGCheckpointExcludeRegion(args..)*).

The IBM Checkpointing library only provides support for full checkpointing. It provides no support for incremental checkpointing. Thus, for practical applicability, in this paper, we currently focus on studying applications, their memory usage and checkpointing trends with respect to the full checkpointing method.

IV. Overview of the Applications Used

In this section, we describe the applications that we run on the IBM Blue Gene machines. We study four popular applications from the fields of Molecular Dynamics and Computational Fluid Dynamics.

A. Molecular Dynamics Simulations

In classical molecular dynamics [22], a single potential energy surface is represented by the force field. However, most times, this level of representation is deemed in-efficient. More accurate representations, involving atomic structure,

electron/proton distribution, chemical reactions, electronic behavior can be generated using quantum mechanical methods such as Density Functional Theory. This area of molecular dynamics is known as Ab Initio (first principles) Molecular Dynamics (AIMD) [23]. GPAW (Grid Projected-Augmented method) and CPMD (Carr-Parrinello Molecular Dynamics) are two very popular Al Initio Molecular Dynamics codes. AIMD-based simulations tend to be highly complex with more stringent requirements for computational and memory resources as compared to traditional classical molecular dynamics code. The emergence of peta-scale machines like the IBM Blue-Gene series has resulted in significant provisions for running such AIMD simulations efficiently and accurately. We thus choose two applications from this field, as described below, for our study of checkpointing trends on the IBM Blue Gene supercomputer.

1) Grid-Based Projector-Augmented Wave Application: The GPAW [24] application is a density-functional theory (DFT) based code that is built on the projector-augmented wave (PAW) method and can use real-space uniform grids and multigrid methods. In the field of material and quantum physics, the Schrodinger equation [25] is considered an important equation since it describes how the quantum state of a physical system changes in time. Various electronic structure methods [26] can be used to solve the Schrodinger equation for the electrons in a molecule or a solid, to evaluate the resulting total energies, forces, response functions, atomic structure, electron distributions and other attributes of interest. The Projector Augmented Wave method is an electronic structure method for ab-inito molecular dynamics with full-wave functions. The term PAW is often also used used to describe the CP-PAW code developed originally by Blochl [27]. An advantage of PAW over other electronic structure methods is that PAW allows end-users to get rid of core electrons and work with soft pseudo valence wave functions.

The GPAW application allows users to represent these pseudo wave functions on uniform real-space orthorhombic grids. This makes it possible to run these modelling codes on very large systems, the results of which could be used to provide a theoretical framework for interpreting experimental results and even to accurately predict the material properties before experimental data actually becomes available. More information on GPAW, can be found in [28].

2) Carr-Parrinello Molecular Dynamics Application: The CPMD [23] [29] code is another electronic structure method and a parallelized plane wave/pseudo-potential implementation of density functional theory, which targets ab inito quantum mechanical molecular dynamics plane wave basis sets. The CPMD code is based on the Kohn-Sham [30] DFT code, and it provides a rich set of features that have been successfully applied to calculate static and dynamic properties of many complex molecular systems such as water, proteins, and DNA bases, as well as various processes such as photoreactions, catalysis, and diffusion. Reactions and interactions in such systems are too complicated to be handled by classic molecular dynamics; but they can be successfully handled in the Carr-Parrinello method because they are calculated directly from the electron structure in every time step. CPMD's flexibility and high performance on many computer platforms have made an optimal tool for the study of liquids, surfaces, crystals, and biomolecules.

CPMD runs on many computer architectures. Its well-parallelized nature, based on MPI, makes it a popular application

that can take advantage of petascale systems, motivating us to choose it for this study.

B. Computational Fluid Dynamics Applications

Computational Fluid Dynamics (CFD) is a method of analyzing fluid and gas flow in structures and surfaces [31] using numerical methods and algorithms. CFD applications are used to predict how fluids flow and to predict the transfer of heat, mass, structural deformations, radiation, chemical reactions of the fluids and the solids with which they are in contact. Such CFD problems, which involves millions of calculations, place an immense demand on computing resources. The goal of a CFD application is to obtain a valid solution for a given CFD problem (constrained by certain boundaries) in a reasonable amount of time. With the availability of large scale machines like Blue Gene/P, it is possible to achieve a very high accuracy in computations involving CFD principles. CFD applications are used in various fields such as ocean modeling, thermal hydraulics, polymer processing, weather simulation etc. In this paper, we study two popular CPD applications; namely; the *Nek5000* application and *Parallel Ocean Program(POP)* application.

1) Nek5000: NEK5000 [32] is an open source, Gordon Bell prize winning, spectral element Computational Fluid Dynamics (CFD) code developed at the Mathematics and Computer Science Division at Argonne National Laboratory. The C and Fortran code, which won a Gordon Bell prize, focuses on the simulation of unsteady incompressible fluid flow, convective heat with species transport, and magnetohydrodynamics. It can handle general two- and three-dimensional domains described by isoparametric quad or hex elements. In addition, it can be used to compute axisymmetric flows. Nek5000 is a time-stepping-based code and supports steady Stokes and steady heat conduction. It also features some of the first practical spectral element multigrid solvers, which are coupled to a highly scalable, parallel, coarse-grid solver.

The Nek5000 application was chosen for this study because it is highly scalable and can scale to processor counts of over 100,000, typical of petascale computing platforms. In addition, the Nek5000 software is used by many research institutions worldwide with a broad range of applications, including ocean current modeling, combustion, spatiotemporal chaos, the interaction of particles with wall-bounded turbulence, thermal hydraulics of reactor cores, and transition in vascular flows.

2) The Parallel Ocean Program(POP): The Parallel Ocean Program (POP) [33], developed at the Los Alamos National Laboratory, is a three-dimensional ocean circulation model, designed to study ocean climate system. The POP application is an integral part of the SPEC 2007 Benchmark suite. The POP application has been used to perform the highest resolution global ocean simulation at Los Alamos National Laboratory. POP is used in a variety of applications, including very high resolution eddy-resolving simulations of the ocean and as the ocean component of coupled climate models like the Community Climate System Model.

POP is a descendent of the Bryan-Cox model, that has been used frequently for ocean climate simulations. POP has substantial improvements [34] over the earlier model. The POP application solves the three-dimensional equations for fluid motions on the sphere under hydrostatic and Boussinesq approximation. The spatial derivatives are computed using finite-difference discretizations, which are formulated to handle any generalized orthogonal grid on a sphere. These grids

include the dipole and tripole grids which shift the North Pole singularity into land masses to avoid time step constraints due to grid convergence. Time integration of the model is split into two parts. The three-dimensional vertically-varying (baroclinic) tendencies are integrated explicitly using a leapfrog scheme. The very fast vertically-uniform (barotropic) modes are integrated using an implicit free surface formulation in which a preconditioned conjugate gradient solver is used to solve for the two-dimensional surface pressure. A wide variety of physical parameterizations and other features are also available in the model.

Portability in POP is achieved by isolating all communication routines into a small set of modules which can be modified for specific architectures [34]. POP has been validated on many platforms including the IBM Blue Gene/L [35]. The combination of fine-granularity simulations, provided in POP, to resolve ocean eddies (which can impact dynamics of the ocean significantly) and long time scales required for climate and deep ocean circulation necessitates many computational cycles; making POP a good candidate for our study on BG/P.

V. Theory and Experiments

In this section we describe the experimental methodology used to conduct this study. We also describe our checkpointing model and use it in conjunction with the observed application memory trends to gain insight into optimal checkpointing parameters.

The scope of our study measures and analyzes the below trends.

1) How the application memory usage varies over application execution time and.
2) How the application memory usage varies with system size.

These memory trends were observed for the GPAW, CPMD, Nek5000 and the POP application for varying system size upto 65,536 cores. These trends were then analyzed to determine the checkpoint frequency and checkpoint duration using the optimum checkpoint model. In the next few subsections, we discuss the mechanism used to measure the memory usage and the details of our checkpointing model.

A. Recording Memory Usage Patterns

In the 'Full memory checkpointing' technique, a snapshot of the entire process memory for every process is taken. In our experiments, we run the various application codes and record the data memory, stack memory and process-related information. Applications running on the IBM BG/P are statically linked, with the instruction-specific text code remaining remaining constant. We, thus, disregard the text sections of the program in our measurements. The data (which includes the heap memory) and stack memory, manage the local, global, static and declared variables, as well as memory requested by various system (new, malloc, calloc) calls - which constitutes the data that needs to be checkpointed.

In our setup, minimal amount of instrumentation code was inserted at the startup of each application that could record the memory usage for a process. An optimal method to achieve this, transparently to the application, would have been through

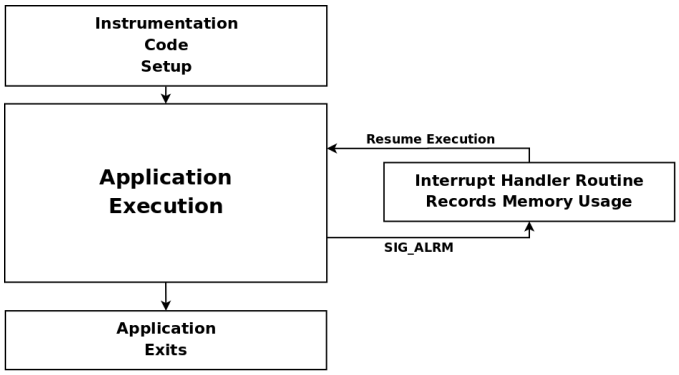

Figure 3. Instrumentation Code

constructor methods made available in a statically linked external library. However, the BG/P compilers, provided by IBM, do not provide this option. As an alternative, we invoke this measuring code soon after the entry point of the application.

From an implementation perspective, the memory-usage measuring code is based on a timer function that sets up a timer interrupt for a certain defined interval. As shown in Figure 3, the interrupt handler method which when invoked, measures the amount of memory used and records it. The *getrusage()* function, available in the POSIX.1-2001, is used to track the memory being used.

*1) The **getrusage()** function:* The getrusage function has the following the signature: **int getrusage(int who, struct rusage *r_usage)**. The *who* argument determines whether the values reported are for the calling process or the waited-for process. For our purposes we set the value of *who* to **RUSAGE_SELF**. This returns the resource usage for the calling process. The structure of *rusage* is given in table I

The *ru_maxrss* field returns the maximum resident set size of the program. This is the number used for recording of memory usage by the instrumentation code.

B. The Optimum Checkpointing Model

Runtime on large systems like Intrepid is a valuable commodity, which users wish to use in the most optimal manner. While checkpointing is a necessary activity, users wish to devote only a certain percentage of application execution time to this procedure. Knowledge about checkpoint duration and frequency, based on such constraints, would help the user make more informed decisions and tradeoffs between their resource usage and application resiliency. We attempt to provide this information through the "optimum checkpoint model" discussed in this section. Our checkpoint model has been largely influenced by past research done with other checkpointing models [36] [37] in this field.

To arrive at the analytical model of the optimum checkpointing scheme on the Blue Gene/P system, let us consider an example application as shown in Figure 4. Like a majority of scientific applications, this application has a constant memory pattern for a majority of its execution period. The figure is a diagrammatic representation of the application with

Type	Field	Function
struct timeval	ru_utime;	user time used
struct timeval	ru_stime;	system time used
long	ru_maxrss;	maximum resident set size
long	ru_ixrss;	integral shared memory size
long	ru_idrss;	integral unshared data size
long	ru_isrss;	integral unshared stack size
long	ru_minflt;	page reclaims
long	ru_majflt;	page faults
long	ru_nswap;	swaps
long	ru_inblock;	block input operations
long	ru_oublock;	block output operations
long	ru_msgsnd;	messages sent
long	ru_msgrcv;	messages received
long	ru_nsignals;	signals received
long	ru_nvcsw;	voluntary context switches
long	ru_nivcsw;	involuntary context switches

Table I
THE *rusage* STRUCTURE

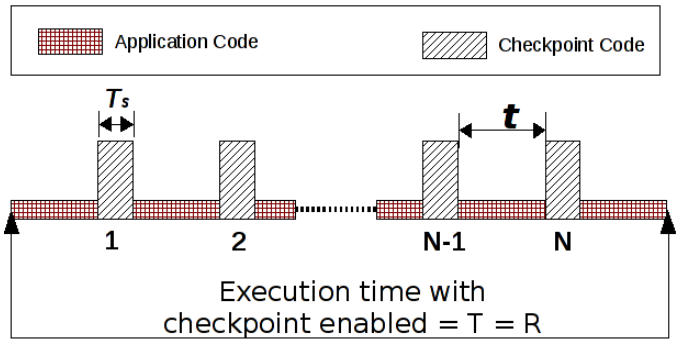

Figure 4. Application Execution with Checkpoint Enabled

checkpointing feature enabled.

Here T is the total execution time, including the time required to perform all the checkpoints; T_s is the time required to complete one full checkpoint operation; N is the optimum number of checkpoints to be performed during the length of the application execution; and t is the optimum time interval between two checkpoints when the application resumes execution.

Let *n* be the number of cores the application is run on, *M* be the mean memory usage per core, and *B* be the unidirectional bandwidth from all the compute nodes to storage disks that is available to the entire application. Based on these parameters, the time T_S required to save the state of all processes(cores) during a single checkpoint can be given by:

$$T_s = \frac{M \times n}{B}. \tag{1}$$

If T_A is the actual length of time the application needs to complete without checkpointing enabled, the total application

runtime with checkpointing enabled i.e. T is given by the below equation.

$$T = T_A + (N \times T_s), \tag{2}$$

Note that, for long running applications on large systems, end-users tend to have information on the total execution times (i.e T_A) based on historical data or past runs performed. Frequently, users wish to devote only a certain percentage (X) of application execution time T_A for checkpointing (Note that in such situations, users increase the reservation time by $X\%$ as well). Thus, the overall execution time of the application with checkpoint enabled can also be described as:

$$T = T_A + (X \times T_A), \tag{3}$$

Based on equations 1, 2 and 3, one can estimate the total number of checkpoints N possible as follows:

$$T_A + (N \times T_s) = T_A + (X \times T_A) \tag{4}$$

$$N \times T_s = X \times T_A \tag{5}$$

Therefore *N* can be computed as follows:

$$\therefore N = \left\lfloor \frac{XT_AB}{Mn} \right\rfloor \tag{6}$$

We can, thus, derive the number of optimum checkpoints based on the (a) percentage of application execution time dedicated for checkpointing, (b) bandwidth from the compute nodes to file servers and (c) the total amount of data to be checkpointed. Having this information provides users the flexibility to choose areas they wish to checkpoint. In practice, is may be difficult to predict the accurate value for *B* since several applications may use the I/O and network resources at the same time. However, the user can made educated guesses based on their system size and file system architecture, as we show in the next section.

Finally, the frequency of checkpointing or time interval between two checkpoints can be calculated as follows:

$$t = \frac{T_A}{N} \tag{7}$$

Simplifying the above equation with substitutions from 6

$$t = \frac{Mn}{XB} \tag{8}$$

Having checkpoint interval information makes it easier to develop independent libraries that can be compiled into the

application and can used timer-based methods to checkpoint periodically.

C. Application Evaluation

In this section, we carry our the evaluation of the GPAW, CPMD, Nek5000 and the POP application on the Intrepid system.

Note that all the applications that we used for the experiments exhibit strong scaling characterestics since each run instance had the same problem size. Strong scaling is a methodology used where the problem size is kept constant. In this case the performance of the application with respect to time varies as the number of processors vary. Such a method is limited by *Amdahl's law*. Note that in order to over come this shortcoming, weak scaling is used. (Weak scaling entails increasing the problem size with the increasing the number of processors while keeping the execution time constant.)

Applications on the Intrepid system can be run in three modes: (a) SMP mode: In this mode, only one process, with a maximum of four threads can be launched on each compute node with each thread using a core, (b) Dual mode: In this mode, two processes with a maximum of two threads each can be launched with each thread using a core. The 2 GB is equally divided between the two processes; (c) Virtual mode: In this mode, a single process per core (i.e four processes per compute node) can be launched. The 2 GB memory is equally divided among all the four processes.

The GPAW application (version 0.4), being evaluated, consists of 256 water molecules with 2,048 electrons, 1,056 bands and 112^3 grid points, with a grid spacing of 0.18. This application was run on the Intrepid system, with compute node count varying from 32 cores to 1,024 cores. The problem size was kept constant between all runs of the application. Note that memory in GPAW is dynamically allocated. The application was run in the smp mode with single thread. Figure 5(a) shows the memory trends for upto 256 cores and Figure 5 shows it for 512 and 1,024 cores. The x-axis shows the total application execution time and the y-axis shows the memory usage per core. As can be seen from the graphs, the memory requirements for the GPAW application grows relatively slowly against the total time execution as the system size increases. We also observe that the application execution time decreases with increasing system size. The graph for GPAW clearly shows the decrease in memory footprint per core as the number of processors (system size) increase. The memory usage per core remains constant once the peak is attained, indicating a constant possible checkpointing time in later stages of the code.

Figure 6 shows memory trends for the CPMD application. The CPMD application was run in the SMP mode with four threads on a system size of 8,192 cores. While the CPMD memory consumption trend is similar to the GPAW application, we notice that CPMD memory consumption increases a little slowly as system size increases. The important difference is that the memory consumption for the majority of the execution time remains the same irrespective of the system size.

Figure 7 shows the memory consumption trend for the Nek5000. The Nek5000 application was run in virtual mode on a system size ranging from 8,192 cores to 32,768 cores. The 3-D graph in Figure 7 shows system size on the z-axis and the application execution time and memory usage (for the various system size), on the x-axis and y-axis respectively. In

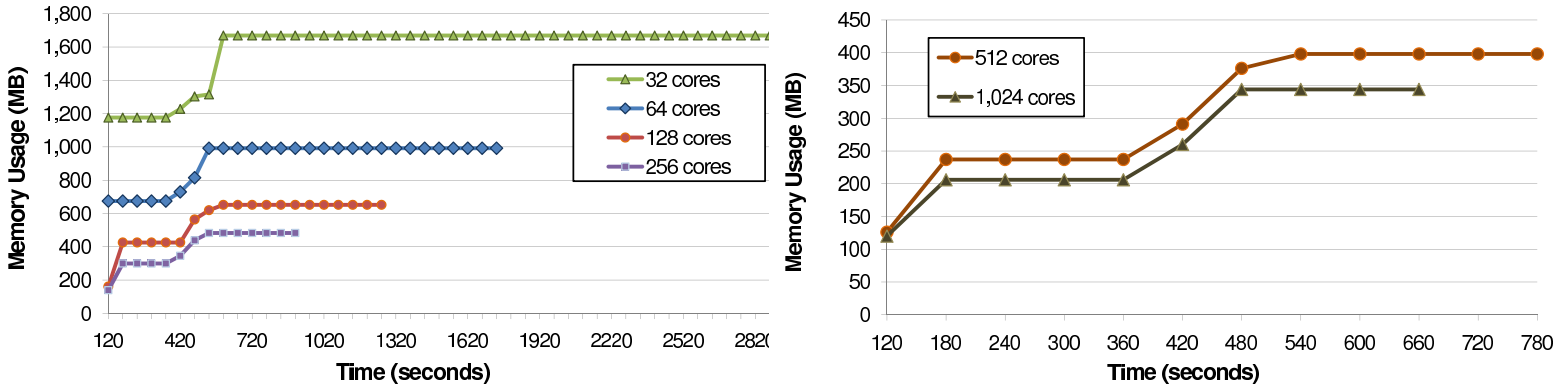

Figure 5. GPAW Memory Consumption (a) Small Systems; (b) Large Systems

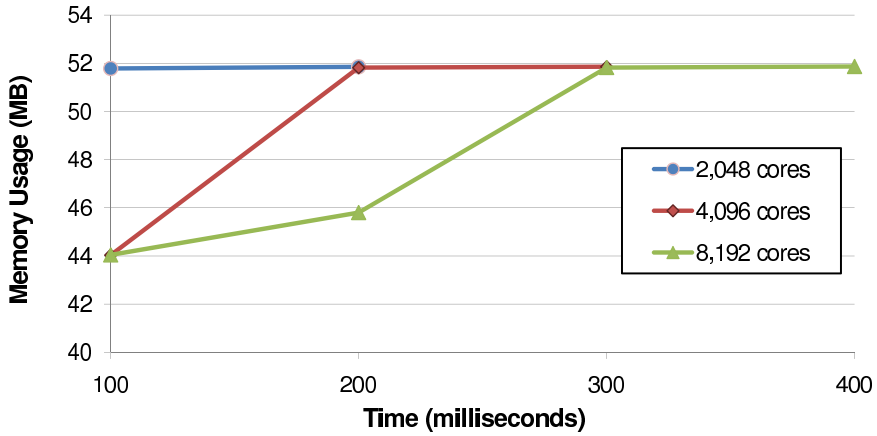

Figure 6. CPMD Memory Consumption

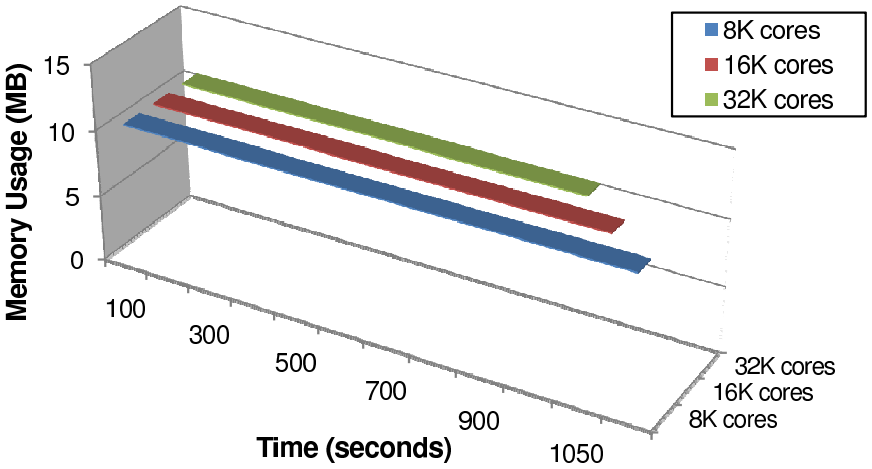

Figure 7. Nek5000 Memory Consumption

Nek5000, the memory is allocated as soon as the application starts up and remains the same for the execution lifetime of the process, irrespective of the number of processors the application is finally meant to run on or the size of the problem set assigned to a particular node. Like CPMD, Nek5000 exhibits that the memory consumption does not vary with a change

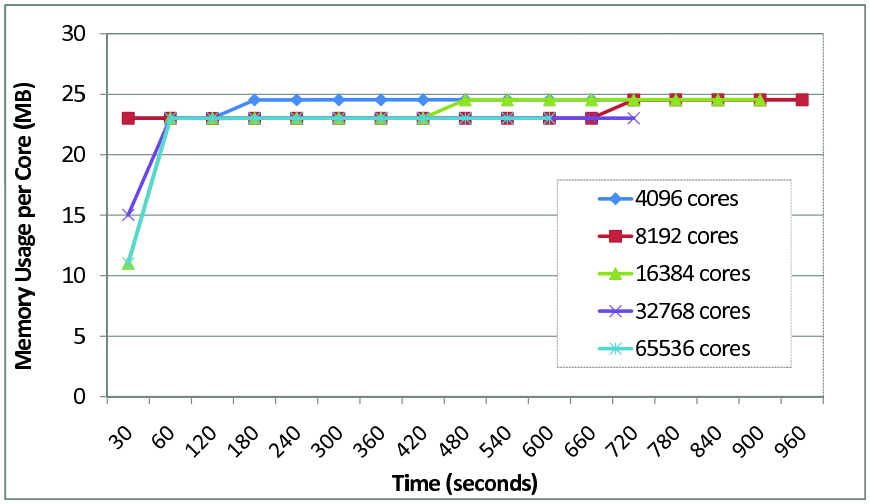

Figure 8. POP Memory Consumption

in system size.

Figure 8 shows memory for the POP ocean modeling application. The POP application was run in the virtual mode for system sizes ranging from 4,096 cores to 65,536 cores. The trends seen for the POP application show that memory usage increases rapidly once the application starts up and it remains constant for the lifetime of the application.

D. Computing Optimum Checkpointing Values

We briefly discussed the I/O and the network infrastructure of the Intrepid system in Section III. As discussed, the bandwidth between the 10-Gigabit Ethernet network and the file servers is theoretically 2 Tbps. When an application uses the entire BG/P system, all the 640 I/O nodes can theoretically deliver up to a maximum of 4.25 Tbps. This indicates that the bandwidth bottleneck for the maximum data throughput lies more towards the file servers than the I/O nodes when the system is running full capacity. However since our study involves only using upto 16 racks (65,536 cores)) out of the 40 racks (163,840 cores) available, we only use a maximum of 256 I/O nodes, thus limiting the I/O node bandwidth to 1740 Gbps (i.e. 256 x 6.8Gbps).

Moreover, as we are utilizing only a fraction of the total system capacity, the available file server I/O bandwidth for our runs is also limited by the other applications that are performing I/O bound operations. Each I/O node is equipped with network interface capable of delivering 6.8 Gb/s. We based our optimum checkpoint value calculation on two cases. In the first case(B_{30}), we assume that we are able to makes use of 30% of the total bandwidth provided by each I/O node interface. In the second case(B_{60}), we optimistically assume that we make use of 60% of the total bandwidth provided by each I/O node interface.

On the Intrepid system, the I/O node to compute node ratio is 1:64. Each compute node consists of a quad-core processor. When an application is run in the virtual node mode or SMP mode with four threads, the 64 compute nodes provide a total of 256 cores. The Nek5000 and POP application were run in virtual node mode and CPMD application was run in SMP

n	M	T_A	X	30% Bandwidth			60% Bandwidth		
				B_{30}	N_{30}	t_{30}	B_{60}	N_{60}	t_{60}
32	1650	3168	0.3	127.5	2	1584	255	4	792
64	1000	1914	0.3	255	2	957	510	4	478.5
128	650	1386	0.3	510	2	693	1020	5	277.2
256	475	990	0.3	1020	2	495	2040	4	247.5
512	400	858	0.3	2040	2	429	4080	5	171.6
1024	350	726	0.3	4080	2	363	8160	4	181.5

Table II
COMPUTED VALUES FOR GPAW

n	M	T_A	X	30% Bandwidth			60% Bandwidth		
				B_{30}	N_{30}	t_{30}	B_{60}	N_{60}	t_{60}
2048	51.82	220	0.4	2040	1	220	4080	3	73.33
4096	51.85	330	0.4	4080	2	165	8160	5	66
8192	51.87	440	0.4	8160	3	146.67	16320	6	73.33

Table III
COMPUTED VALUES FOR CPMD

n	M	T_A	X	30% Bandwidth			60% Bandwidth		
				B_{30}	N_{30}	t_{30}	B_{60}	N_{60}	t_{60}
8192	10.52	1080	0.05	8160	5	216	16320	10	108
16384	10.52	1050	0.05	16320	4	262.5	32640	9	116.67
32768	10.52	1000	0.05	32640	4	250	65280	9	111.11

Table IV
COMPUTED VALUES FOR NEK5000

mode with 4 threads.

Assuming, $B_{I/O}$ to be the total bandwidth available per I/O node and *n* to be the number of cores the application is running on , the total bandwidth available to the Nek5000 application, the CPMD application and the POP application can be computed by:

$$B = \left(\frac{n}{64 \times 4}\right) \times B_{I/O}. \tag{9}$$

n	M	T_A	X	30% Bandwidth			60% Bandwidth		
				B_{30}	N_{30}	t_{30}	B_{60}	N_{60}	t_{60}
4096	25.13	960	0.07	4080	2	480	8160	5	192
8192	25.13	960	0.07	8160	2	480	16320	5	192
16384	25.13	900	0.07	16320	2	450	32640	4	225
32768	23.57	900	0.07	32640	2	450	65280	5	180
65536	23.57	900	0.07	65280	2	450	130560	5	180

Table V
COMPUTED VALUES FOR POP

The GPAW application is executed in SMP mode with one thread, with only core being used on each compute node. The total bandwidth available to the GPAW application can be computed by:

$$B = \left(\frac{n}{64}\right) \times B_{I/O}. \tag{10}$$

For case 1 with 30% of total bandwidth available, we have $B_{I/O} = 0.30 \times 6.8Gb/s = \frac{0.30 \times 6800}{8} = 255$ MB/s.

For case 2 with 60% of total bandwidth available, we have $B_{I/O} = 0.60 \times 6.8Gb/s = \frac{0.60 \times 6800}{8} = 510$ MB/s.

Based on the the equations from Section V-B and the observations in Section V-C, we can compute the optimum checkpointing parameter values for each of these applications.

Let us assume that the user is willing to dedicate 3% to 7% of the application execution time for performing checkpoint operations. So we have X varying from 0.03 to 0.07 for the different applications.

Based on equations 6, 8, 9 and 10 and the graphs obtained in the previous section, we compute the optimum checkpointing parameters for the applications and tabulate them. Table II, III, IV and V show the calculated checkpointing values for the four applications; where n is the number of cores, M is the average memory usage per core in MBytes, T_A is the application execution time in seconds (milliseconds for the CPMD application), B is the unidirectional bandwidth from all the compute nodes to storage disks in MB/s, N is the number of optimal checkpoints, t is the checkpoint interval in seconds (milliseconds for the CPMD applications).

Table II shows the various checkpoint values for node count ranging from 32 to 1024 nodes and X set to 3%. We see that for B_{30}, the number of optimal checkpoints is 2 for all node ranges and for B_{60}, the optimal checkpoint count lie between 4 and 5. Thus, we see that checkpoint frequency will heavily depend on the available I/O bandwidth, which can be challenging to determine.

The remaining tables show similar calculations for the various other applications.

VI. Conclusions and Future Work

In this paper, we discussed checkpointing trends for applications running on leadership class machines such as the IBM Blue Gene/P Intrepid system at Argonne National Laboratory. Ranked #5 on the Top500 November 2008 ranking, the Intrepid has 163,840 processors and a peak performance of 557 teraflops. We also presented an analytical model for efficiently computing the optimum checkpoint frequencies and intervals and studied three applications: the Grid-Based Projector-Augmented Wave application (GPAW), the Carr-Parrinello Molecular Dynamics application (CPMD), the Nek5000 computational fluid dynamics application and the Parallel Ocean Program (POP). We also showed with the help of experimental data and computed values how application scaling characteristics influence checkpoint-related decisions. Our current work considered "full checkpointing",where the entire program state of the processes is stored during the checkpoint operation. We chose this approach because the IBM checkpointing library currently supports only full checkpointing. We are conducting a similar

study for incremental checkpointing of applications on IBM BG/P, which will be useful for incremental checkpointing libraries built in the future for this machine. Our current study has been conducted on upto 65,536 processes of the Intrepid system.

VII. Acknowledgements

This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. This research also used resources of the Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] J. Dongarra and H. W. Meuer and E. Strohmaier. Technical report.

[2] D. Patterson and G. Gibson and R. Katz. A case for redundant arrays of inexpensive disks RAID. In *ACM SIGMOD International Conference on Management of Data*, 1988.

[3] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery For Distributed Systems. In *ACM Fall Joint Computer Conference*, 1986.

[4] J. Plank and Y. Chen and K. Li and M. Beck and G. Kingsley. Memory Exclusion: Optimizing the performance of Checkpointing Systems. In *Software: Practice and Experience*, volume 29-2, 1999.

[5] G. Bronevetsky and D. Marques and K. Pingali and R. Rugina. Compiler-Enhanced Incremental Checkpointing. In *LNCS series: Languages and Compilers for Parallel Computing*, 2008.

[6] S. Garg and Y. Huang and C. Kintala and K. Trivedi. Minimizing completion time of a program by Checkpointing and Rejuvenation. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1996.

[7] IBM Blue Gene Team. Overview of the IBM Blue/Gene Project. 52-1/2, 2008.

[8] J. Sancho and F. Petrini and G. Johnson and E. Frachtenberg. On the Feasibility of Incremental Checkpointing for Scientific Computing. In *IPDPS*, 2004.

[9] C. Sosa and B. Knudson. IBM System Blue Gene/P Solution: Blue Gene/P Application Development. 2007. Available at http://www.redbooks.ibm.com/abstracts/sg247287.html.

[10] J. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, University of Tennesse, Knoxville, 1997. Available at http://www.cs.utk.edu/ plank/plank/papers/CS-97-372.html.

[11] L. Silva and J. Silva. System-Level versus User-Defined Checkpointing. In *SRDS*, 1998.

[12] A. Zomaya. Parallel and Distributed Computing Handbook. 1995.

[13] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Implementation and Evaluation of a Scalable Application-Level Checkpoint-Recovery Scheme for MPI Programs. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 38, Washington, DC, USA, 2004. IEEE Computer Society.

[14] G .Barigazzi and L. Strigini. Application-Transparent Setting of Recovery Points. In *FTCS*, 1983.

[15] C. Li and W. Fuchs. CATCH - Compiler Assisted Techniques for Checkpointing. In *FTCS*, 1990.

[16] C. Guohong and M. Singhal. On Coordinated Checkpointing in Distributed systems. In *TPDS*, volume 9-12, pages 1213–1225, 1998.

[17] J. Plank and M. Beck and G. Kingsley and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, 1995.

[18] Y. Liu, R. Nassar, C. Leangsuksun, M. Paun, and S. L. Scott. An Optimal Checkpoint/Restart Model for Large Scale High Performance Computing System. pages 1–9, 2008.

[19] W. Dongsheng, Z. Weimin, W. Dingxing, and S. Meiming. Checkpointing and Rollback Recovery for Network of workstations. *Science in China Series E: Technological Sciences*, 42:207–214.

[20] N. Desai and R. Bradshaw and C. Lueninghoener and A. Cherry and S. Coghlan and W. Scullin. Petascale System Management Experiences. In *LISA*, 2008.

[21] M. Snir and S. Otto. *MPI-The Complete Reference: The MPI Core*. 1998.

[22] R. Car and M. Parrinello and J. Schmidt and D. Sebastiai et al. Unified Approach for Molecular Dynamics and Density-Functional Theory. 1985.

[23] D. Marx and J, Hutter. Ab-initio Molecular Dynamics: Theory and Implementation. NIC series, Volume 3, 2000.

[24] J. J. Mortensen, L. B. Hansen, and K. W. Jacobsen. Real-space grid implementation of the projector augmented wave method. *Phys. Rev. B*, 71(3):035109, Jan 2005.

[25] T. Cazenave. An Introduction to Nonlinear Schrodinger Equations. 1989.

[26] J. Foresman and E. Frisch. Exploring Chemistry with Electronic Methods. 1996.

[27] P. Blochl and C. Forst and J. Schimpl. Projector Augmented Wave Method: Ab Initio Molecular Dynamics with Full Wave Functions. 26-1, 2003.

[28] GPAW Website: https://wiki.fysik.dtu.dk/gpaw.

[29] W. Andreoni and A. Curioni. New Advances in Chemistry and Materials Science with CPMD and Parallel Computing. *Parallel Computing*, 26(7-8):819–842, 2000.

[30] W. Koch and M. Holthausen. A Chemist's Guide to Density Functional Theory. 2001.

[31] B. Laval. Numerical Computation of Internal and External Flows: The Fundamentals of Computational Fluid Dynamics. 2008.

[32] J. Lottes and P. Fischer. Hybrid Multigrid/Schwarz Algorithms for the Spectral Element Method . *Journal of Scientific Computing*, pages 45–78, 2004.

[33] The Parallel Ocean Program(POP) Overview, website: http://climate.lanl.gov/Models/POP.

[34] P. W. Jones and P. H. Worley and Y. Yoshida and J. B. White,III and and J. Levesque. Practical performance portability in the Parallel Ocean Program (POP). *Concurrency and Computation : Practice and Experience*, 17(10):1317–1327, 2005.

[35] Kerbyson and J. Darren and P. W. Jones. A Performance Model of the Parallel Ocean Program. *International Journal of High Performance Computing Applications*, 19(3):261–276, 2005.

[36] J. W. Young. A First Order Approximation to the Optimum Checkpoint Interval. *Communications of the ACM ACM*, 17(9):530–531, 1974.

[37] J. T. Daly. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.