Damaris: Leveraging Multicore Parallelism to Mask I/O Jitter

Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, Leigh Orf

Damaris: Leveraging Multicore Parallelism to Mask I/O Jitter

Matthieu Dorier — Gabriel Antoniu —

Franck Cappello — Marc Snir — Leigh Orf

N° 7706

August 2011

Domaine 3

ISSN 0249-6399 ISRN INRIA/RR--7706--FR+ENG

Damaris: Leveraging Multicore Parallelism to Mask I/O Jitter

Matthieu Dorier[*] , Gabriel Antoniu[†] ,
Franck Cappello[‡] , Marc Snir[§] , Leigh Orf[¶]

Domaine : Réseaux, systèmes et services, calcul distribué
Équipes-Projets KerData
Joint INRIA/UIUC Laboratory for Petascale Computing

Rapport de recherche n° 7706 — August 2011 — 25 pages

Abstract: With exascale computing on the horizon, the performance variability of I/O systems presents a key challenge in sustaining high performance. In many HPC applications, I/O is performed concurrently by all processes; this produces I/O bursts, which causes resource contention and substantial variability of I/O performance, significantly impacting the overall application performance. In this paper, we utilize the IOR benchmark to explore the influence of user-configurable parameters on I/O variability. We then propose a new approach, called Damaris, leveraging dedicated I/O cores on each multicore SMP node to efficiently perform asynchronous data processing and I/O. We evaluate our approach on two different platforms with the CM1 atmospheric model, one of the BlueWaters HPC applications. By gathering data into large files while avoiding synchronization between cores, our solution increases the I/O throughput by a factor of 6, hides all I/O-related costs, and enables a 600% compression ratio without any additional overhead.

Key-words: Exascale Computing, Multicore Architectures, I/O, Variability, Dedicated Cores

[*] ENS Cachan Brittany, IRISA - Rennes, France. **matthieu.dorier@irisa.fr**
[†] INRIA Rennes Bretagne-Atlantique - France. **gabriel.antoniu@inria.fr**
[‡] INRIA Saclay - France, University of Illinois at Urbana-Champaign - IL, USA. **fci@lri.fr**
[§] University of Illinois at Urbana-Champaign - IL, USA. **snir@illinois.edu**
[¶] Central Michigan University - MI, USA. **leigh.orf@cmich.edu**

Damaris: Exploitation du Parallelisme Multi-cœur pour Masquer la Variabilité des E/S

Résumé : Alors que l'ère de l'exascale se profile à l'horizon, la variabilité des performances des systèmes d'entrées-sorties (E/S) présente un défi majeur pour permettre d'atteindre des performances élevées et soutenues. Dans de nombreuses applications HPC, les E/S sont effectuées par tous les processus de manière concurrente. Cela engendre des piques d'utilisation des E/S, créant des conflits d'accès aux ressources et une variabilité substantielle dans les performances des E/S, impactant les performances globales de l'application de manière significative. Dans ce rapport, nous utilisons l'application de tests IOR pour étudier l'influence de certains paramètres, configurables par l'utilisateur, sur la variabilité des E/S. Nous proposons ensuite une nouvelle approche, nommée Damaris, qui exploite des coeurs dédiés aux E/S sur chaque nœud SMP de manière à permettre des traitements et des mouvements de données efficaces et asynchrones. Nous évaluons notre approche sur deux plateformes différentes à l'aide de l'application atmosphérique CM1, l'une des applications prévues pour le future supercalculateur BlueWaters. Grâce à l'agrégation de données dans de plus grands fichiers tout en évitant la synchronisation des cœurs, notre solution permet l'amélioration du débit d'écriture par un facteur 6, cache complètement tout coût lié aux E/S et autorise un taux de compression de 600% sans surcoût pour l'application.

Mots-clés : Exascale, Architectures Multi-cœurs, E/S, Variabilité, Cœurs Dédiés

Contents

1 Introduction

As HPC resources approaching millions of cores become a reality, science and engineering codes invariably must be modified in order to efficiently exploit these resources. A growing challenge in maintaining high performance is the presence of high variability in the effective throughput of codes performing I/O. A typical I/O approach in large-scale simulations consists of alternating computation phases and I/O phases. Often due to explicit barriers, all processes perform I/O at the same time, causing network and file system contention. It is commonly observed that some processes will exploit a large fraction of the available bandwidth and quickly terminate their I/O, remaining idle (typically from several seconds to several minutes) while waiting for slower processes to complete their I/O. Even at 1000-way scale this high I/O jitter is common, where observed I/O performance can vary by several orders of magnitude between the fastest and slowest processes [26]. Exacerbating this problem is the fact that HPC resources are typically running many different I/O intensive jobs concurrently. This creates file system contention between jobs, further increasing the variability from one I/O phase to another and creating unpredictable overall run times.

While most studies address I/O performance in terms of aggregate throughput and try to improve this metric by optimizing different levels of the I/O stack from the file system to the simulation-side I/O library, few efforts have been made in addressing I/O jitter. Yet it has been shown [26] that this variability is highly correlated with I/O performance, and that statistical studies can greatly help addressing some performance bottlenecks. The origins of this variability can differ substantially due to multiple factors, including the platform, the underlying file system, the network, and the application I/O pattern. For instance, using a single metadata server in the Lustre file system [10] causes a bottleneck when following the file-per-process approach, a problem that PVFS [5] or GPFS [22] do not exhibit. In contrast, byte-range locking in GPFS or equivalent mechanisms in Lustre cause lock contentions when writing to shared files. To address this issue, elaborate algorithms at the MPI-IO level are used in order to maintain a high throughput [20].

In this paper, we first perform an in-depth experimental evaluation of I/O jitter for various I/O-intensive patterns exhibited by the Interleaved Or Random (IOR) benchmark [23]. These experiments are carried out on the Grid'5000 testbed with PVFS as the underlying file system. We propose a new graphical method to study this variability and show the impact of some user-controllable parameters on this variability.

Based on the observation that a first level of contention occurs when all the cores of a multicore SMP node try to access the network for intensive I/O at the same moment, we propose a new approach to I/O optimization called Damaris (Dedicated Adaptable Middleware for Application Resources Inline Steering). Damaris leverages one core in each multicore SMP node to perform data processing and I/O asynchronously. This key design choice builds on the observation that because of memory bandwidth limitations, it is often not efficient to use all cores for computation, and that reserving one core for kernel tasks such as I/O management may help reducing jitter. Our middleware takes into account user-provided information related to the application behavior, the underlying I/O systems and the intended use of the output in order to perform "smart" I/O and data processing within SMP nodes. The Damaris approach completely removes I/O jitter by performing scheduled asynchronous data processing and movement. We evaluate this approach with the Bryan Cloud Model, version 1 (CM1) [3] (one of the target applications for the Blue Waters [1] system) on Grid'5000 and on BluePrint, the IBM Power5 Blue Waters interim system running at the National Center for Supercomputing Applications (NCSA). By gathering data into large files while avoiding synchronization between cores, our solution increases the I/O throughput almost by a factor of 6 compared to standard approaches, hides all I/O-related costs, and enables a 600% compression ratio without any additional overhead.

This paper is organized as follows: Section 2 presents the background and motivations of our study as well as the related work. In Section 3 we run a set of benchmarks in order to characterize the correlation between the I/O jitter and a set of relevant parameters: the file striping policy, the I/O method employed (file-per-process, collective or grouped) and the size of the data. Our new approach is presented in Section 4 together with an overview of its design and its API. We finally evaluate this approach with the CM1 atmospheric simulation running on 672 cores of the *parapluie* cluster on Grid'5000, and on 1024 cores on the BluePrint machine.

2 Background and related work

Understanding I/O jitter Over the past several years, chip manufacturers have increasingly focused on multicore architectures, as the increase clock frequencies for individual processors has leveled off, primarily due to substantial increases in power consumption. These increasingly complex systems present new challenges to programmers, as approaches which work efficiently on simpler architectures often do not work well on these new systems. Specifically, high performance variability across individual components becomes more of an issue, and it can be very difficult to track the origin of performance weaknesses and bottlenecks. While most efforts today address performance issues and scalability for specific types of workloads and software or hardware components, few efforts are targeting the causes of performance variability. However, reducing this variability is critical, as it is an effective way to make more efficient use of these new computing platforms through improved predictability of the behavior and of the execution run time of applications. In [24], four causes of jitter are pointed out:

1. Resource contention within multicore SMP nodes, caused by several cores accessing shared caches, main memory and network devices.

2. Communication, which imposes a level of synchronization between processes that run within a same node or on separate nodes. In particular, access contention for the network causes collective algorithms to suffer from variability in point-to-point communications.

3. Kernel process scheduling, together with the jitter introduced by the operating system.

4. Cross-application contention, which constitutes a random variability coming from simultaneous access to shared components in the computing platform.

While issues 3 and 4 cannot be addressed by the end-users of a platform, resource contention (issue 1) and communication jitter (issue 2) can be better handled by tuning large-scale applications in such way that they make a more efficient use of resources. As an example, parallel file systems represent a well-known bottleneck and a source of high variability [26]. While the performance of computation phases of HPC applications are usually stable and only suffer from a small jitter due to the operating system, the time taken by a process to write some data can vary by several orders of magnitude from one process to another and from one iteration to another. In [13], the variability is expressed in terms of *interferences*, with the distinction between *internal interferences*, which are caused by access contention between all the processes of an application (which corresponds to issue 2), and *external interferences* due to sharing the access to the file system with other applications, possibly running on different clusters (issue 4). As a consequence, adaptive I/O algorithms have been proposed [13] to allow a more efficient and less variable access to the file system.

Approaches to I/O management in HPC simulations Two main approaches are typically used for performing I/O in large-scale HPC simulations:

The *file-per-process* approach consists in having each process write in a separate, relatively small file. Whereas this avoids synchronization between processes, parallel file systems are not well suited for this type of load at a large scale. Special optimizations are then necessary [4] when scaling to hundreds of thousands of files. File systems that make use of a single metadata server, such as Lustre, suffer from a bottleneck when performing simultaneous creation of a very large number of files. Simultaneous file creations are handled serially, leading to immense I/O variability. Moreover, reading such a huge number of files for post-processing and visualization becomes intractable.

Using *collective I/O*, e.g. in MPI applications, all processes synchronize together to open a shared file, and each process writes particular regions of this file. This approach requires a tight coupling between MPI and the underlying file system [20]. Algorithms termed as “two-phase I/O” [8, 25] enable efficient collective I/O implementations by aggregating requests and by adapting the write pattern to the file layout across multiple data servers [7]. When using a scientific data format such as pHDF5 [6] or pNetCDF [12], collective I/O avoids metadata redundancy as opposed to the file-per-process approach. However, collective I/O imposes a high degree of process synchronization, leading to a loss of efficiency and to an increase of variance in the time to complete I/O operations. Moreover, it is currently not possible with pHDF5 to perform data compression.

It is usually possible to switch between these approaches when a scientific format is used on top of MPI; going from HDF5 to pHDF5 is a matter of adding a couple of lines of code, or simply changing the content of an XML file when using an adaptable layers such as ADIOS [14]. But users still have to find the best specific approach for their workload and choose the optimal parameters to achieve high performance and low variability. Moreover, the aforementioned approaches create periodic peak loads in the file system and suffer from contention at several levels.

Other efforts are focused on overlapping computation with I/O in order to reduce the impact of I/O latency on overall performance. Overlap techniques can be implemented directly within simulations [19], using asynchronous communications. Yet non-blocking primitives in MPI are less efficient than blocking ones, and lead to additional jitter due to more threads running in the same cores. A quantification of the potential benefit of overlapping communication and computation is provided in [21], which demonstrates methodologies that can be extended to communications used by I/O. The approach we introduce in this paper uses dedicated I/O cores and exploits the potential benefit of this overlap *without* carrying the burden of asynchronous MPI calls.

Other approaches leverage data-staging and caching mechanisms [18, 11], along with forwarding approaches [2] to achieve better I/O performance. Forwarding routines usually run on top of dedicated resources in the platform, which are not configurable by the end-user. Moreover, these dedicated resources are shared by all users, which leads to multi-application access contention. However, the trend towards I/O delegate systems underscores the need for new I/O approaches. Our approach follows this idea but relies on dedicated I/O cores at the application level rather than hardware I/O-dedicated or forwarding nodes.

Finally, some research efforts have focused on reserving some computational resources as a bridge between the simulation and the file system or other back-ends such as visualization engines. In such approaches, I/O at the simulation level is replaced by asynchronous communications with a middleware running on a separate set of computation nodes, where data is stored in local memory and processed prior to effective storage. PreDatA [27] is such an example: it performs in-transit data manipulation on a subset of compute nodes prior to storage, allowing more efficient I/O in the simulation and more simple data analytics, at the price of reserving dedicated computational resources. The communication between simulation nodes and PreDatA nodes is done through the DART [9] RDMA-based transport method. However, given the high ratio between the number of nodes used by the simulation and the number of nodes used for data processing, the PreDatA middleware is forced to perform streaming data processing, while our approach using dedicated cores in the simulation nodes allows us to keep the data longer in memory and to smartly schedule all data operations and movements.

3 Impact of User-controllable parameters on I/O variability

The influence of all relevant parameters involving the I/O stack is not possible to ascertain in a single study: changing the file system used together with its configuration, the network, the number of servers or clients, leads to too many degrees of freedom. Moreover, users usually have access to only a restricted set of these parameters. Thus we will focus our study on the ones that the end-user can control: the amount of data written at each I/O phase, the I/O approach used (collective write to a single shared file or individual files-per-process approach), and the file striping policy inside the file system. Our goal in this section is to emphasize the role of theses parameters on the I/O performance variability. We first present the methodology and metrics we use in order to

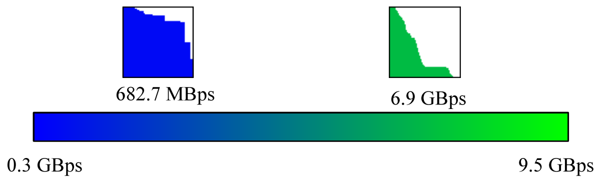

Figure 1: Visual representation of runs variability. The left run has a smaller throughput but less overall idle time than the right one.

compare experiments, then we perform an experimental characterization of the I/O variability on the Grid'5000 French testbed with PVFS, using the IOR benchmark.

3.1 Methodology

The main problem when studying variability instead of performance involves choosing the right metric and the proper way to quickly get valuable insight into the I/O systems behavior in a large number of experiments. Choosing the write variance as a statistical metric for the variability is arguably not appropriate for three reasons: first, it highly depends on the underlying configuration; second, the values are hard to interpret and compare; finally, providing decent confident intervals for such a statistic would require thousands of runs for each tested configuration.

3.1.1 Visual representation of runs

We propose the following methodology to study the variability of I/O systems: we define a *run* as a single I/O phase of an application under a particular configuration. An *experiment* is a set of runs with the same configuration. For a particular run, we measure the time taken by each process to write its data. This time is reported on a small graph (that we call a *tile*), in which processes are sorted vertically by their write time, this time being represented by an horizontal line. We call the *variability* pattern the shape within a particular tile. As will be shown, for a given configuration we find recurrent patterns. An example is provided in Figure 1. The throughput of a run, defined as the total amount of data divided by the time of the slowest process, is shown using a color scale for drawing these graphs, and also printed under each tile. Thus, blue tiles correspond to slow writes, while green tiles correspond to fast writes. Compared to a computation of variance only, this method presents a more complete picture, enabling us to concurrently visualize different types of jitter including variability from a run to another and from a process to another.

We have conducted 20 experiments of 20 runs each, using 576 processes writing concurrently, thereby leading to 230,400 time measurements. This kind of visual representation of experiments let us quickly draw some conclusions regarding overall performance and variability, as the ratio between the white part and the colored part of a tile gives an overview of the time wasted by processes waiting for the slowest to complete the write. The shape within tiles provides some clues about the nature of the variability. The range of colors within a set of tiles provides a visual indication of the performance variability of a configuration for a given experiment, and also let us compare experiments when the same color scale is used. In the following discussion, only a subset of representative runs are presented for each experiment.

3.1.2 Platform and tools considered

The experiments are conducted on two clusters in the Rennes site of the French Grid'5000. The *parapluie* cluster, featuring 40 nodes (2 AMD 1.7 GHz CPUs, 12 cores/CPU, 48 GB RAM), is used for running the IOR benchmark on a total of 576 cores (24 nodes). The latest version OrangeFS 2.8.3 of PVFS is deployed on 16 nodes of the *parapide* cluster (2 Intel 2.93 GHz CPUs, 4 cores/CPU, 24 GB RAM, 434 GB local disk), each node used both as I/O server and metadata server. All the nodes from both clusters are communicating through a 20G InfiniBand 4x QDR link connected to a common Voltaire switch. The MPI implementation used here is MPICH [16], with ROMIO [25] as implementation of MPI-IO, compiled against the PVFS library.

IOR [23] is a benchmark used to evaluate I/O optimizations in high performance I/O libraries and file systems. It provides backends for MPI-IO, POSIX and HDF5, and lets the user configure the amount of data written per client, the transfer size, the pattern within files, together with other parameters. We modified IOR in such a way that each process outputs the time spent writing; thus we have a process-level trace instead of simple average of aggregate throughput and standard deviation. We used IOR to make 576 processes writing either a single shared file using collective I/O or a file per process.

3.2 Results with the IOR benchmark

Following the methodology previously described, this section presents the results achieved with the IOR benchmark and the conclusions that we have drawn from them.

3.2.1 Impact of the data size

The first set of experiments aims at showing the influence of the size of the data written by the processes on overall performance and performance variability. We perform a set of 5 experiments with a data size ranging from 4 MB to 64 MB per process. These data are written in a single shared file using collective I/O. Figure 2 represents 5 of these runs for each data size. We notice that for small size (4 MB) the behavior is that of a waiting queue, with a nearly uniform probability for a process to write in a given amount of time. However, the throughput is highly variable from one run to another, ranging from 4 to 8 GB/s. An interesting pattern is found when each process output 16 MB: the idle time decreases, the overall throughput is higher (around 7 GB/s) and more regular from a run to another. Moreover the variability pattern shows a more fair sharing of resources. Performance then decreases with 32 and 64 MB per process. When writing 64 MB per process, the throughput is almost divided by 7 compared to writing 16 MB.

IOR allows configuration of the transfer size, i.e., the number of atomic write calls in a single phase. When this transfer size is set to 64 KB (which is equal to the MTU of our network and the stripe size in the file system), we observed

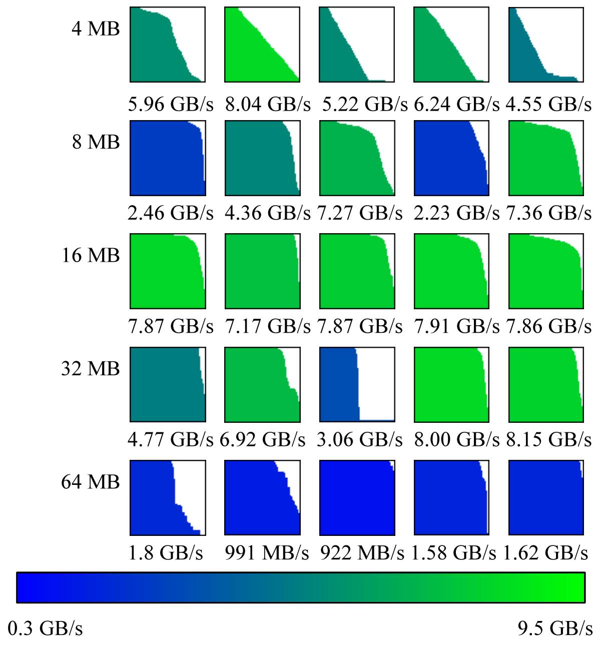

Figure 2: Influence of the data size on I/O performance variability.

that writing 4, 8 or 16 MB per process leads to the same variability pattern and throughput (around 4 GB/s). Increasing the total data size to 32 or 64 MB shows however a decrease of performance and an increase of variability from one run to another. With 64 MB, a staircase pattern starts to appear, which might be due to the amount of data reaching cache capacity limits in the file system.

3.2.2 Impact of the striping policy

PVFS allows the user to tune the file's striping policy across the I/O servers. The default distribution consists in striping data in a round robin manner across the different I/O servers. The stripe size is one of the parameters that can be tuned. The PVFS developers informed us that given the optimizations included in PVFS and in MPICH, changing the default stripe size (64 KB) should not induce any change in aggregate throughput. We were however curious to see if this would produce any change in the variability.

We ran IOR three times, making each of the 576 processes writes 32 MB in a single write within a shared file using collective I/O. We observed that a similar maximum throughput of around 8 GBps is achieved for both 64 KB, 4 MB and 32 MB for stripe size. However, the throughput was lower on average with a stripe size of 4 MB and 32 MB, and the variability patterns, which can be seen in Figure 3, shows more time wasted by processes waiting.

While the stripe size had little influence on collective I/O performance, this was not the case for the file-per-process approach. When writing more than 16 MB per process, the lack of synchronization between processes causes a huge access contention and some of the write operations to time out. The generally-

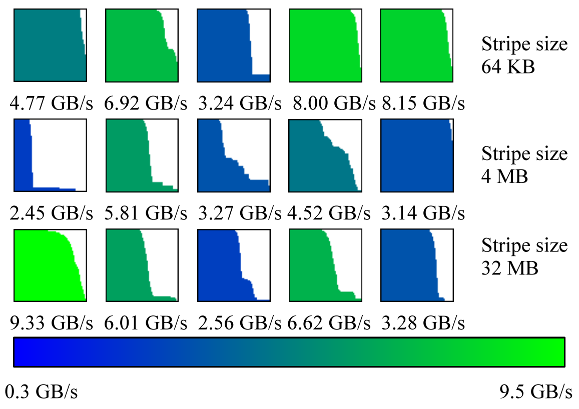

Figure 3: Throughput variability for three experiments with different stripe size.

Data size per client	**Shared file**	**File per process**
4 MB	5.54 GB/s	1.45 GB/s
8 MB	5.8 GB/s	1.58 GB/s
16 MB	7.15 GB/s	4.64 GB/s

Table 1: Throughput comparison: collective I/O against file-per-process, average over 20 runs. Each process output 32 MB in a single write. The stripe size in the file system is 64 KB.

accepted approach of increasing the stripe size for the many-file approach is thus justified, but as we will see, this approach still suffers from high variability.

3.2.3 Impact of the I/O approach

We finally compared the collective I/O approach with the file-per-process approach. 576 processes are again writing concurrently in PVFS. With a file size of 4 MB, 8 MB and 16 MB and a stripe size of 64 KB in PVFS, the file-per-process approach shows a much lower aggregate throughput than collective I/O. Table 1 compares the collective and file-per-process approaches in terms of average aggregate throughput for these three data sizes. Figure 4 presents the recurrent variability patterns of runs for 8 MB per process (runs writing 4 MB and 16 MB have similar patterns). We observe a staircase progression, which suggests communications timing out and processes retrying in series of bursts.

As previously mentioned, writing more data per client required to change the stripe size in PVFS. Yet the throughput obtained when writing 32 MB per process using a 32 MB stripe size and the file-per-process approach is found to be highest over all experiments we did, with a peak throughput of more than 21 GB/sec. Thus Figure 5 shows the results with a different color scale in order not to be confused with other experiments. We notice that despite the high average and peak throughput, there are periods where a large number of processes are idle, and there is a high variance from a run to another.

In conclusion, the experiments presented above show that the performance of I/O phases is extremely variable, and this variability, together with the average

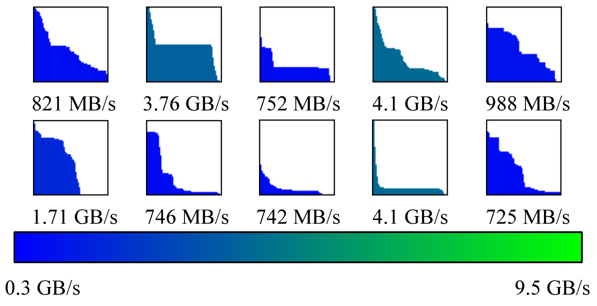

Figure 4: Throughput variability with the file-per-process approach.

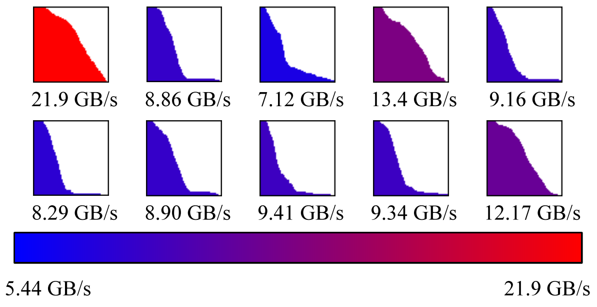

Figure 5: Throughput variability with the file-per-process approach, writing 32 MB per process with a 32 MB stripe size.

and peak throughput achieved, highly depends on the data size and on the I/O approach. Whereas changing the stripe size in PVFS has almost no influence when using collective I/O, it greatly helps achieving a very high throughput with the file-per-process approach. We have successfully demonstrated that the chosen parameters and the I/O approach have a big impact on the performance and variability of I/O phases. The last experiment revealed that while a high throughput of at least 21 GB/s can be expected of the file system, poor configuration choices can lead to barely a few hundreds of MB/s.

4 The dedicated core approach

One conclusion of our study of I/O performance variability is that, in order to sustain a high throughput and a lower variability, it is preferable to write large data chunks while avoiding as much as possible access contentions at the level of the network interface and of the file system. Collective I/O reduces I/O variability, but introduces additional synchronizations that limit the global I/O throughput. On the other hand, the file-per-process achieves a better throughput, but leads to a high variability and to complex output analysis. Considering that the first level of contention occurs when several cores in a single SMP node try to access the same network interface, we propose to gather the I/O operations into one single core that will perform writes of larger data in each SMP node. Moreover, this core will be dedicated to I/O (i.e. will not perform computation) in order to overlap writes with computation and avoid contention for accesses to the file system. This section describes an implementation of this approach, called Damaris, and presents an overview of its design and of its API.

4.1 Principle

Damaris consists of a set of servers, each of which runs on a dedicated core of every SMP node used by the simulation. This server keeps data in a shared memory segment and performs post-processing, filtering and indexing in response to user-defined events sent either by the simulation or by external tools.

The buffering system running on these dedicated cores is configured to match the expected I/O pattern. Based on this knowledge, it initializes a metadata-rich buffer for incoming datasets and schedules all write requests from its clients. Clients (computation cores) write their data concurrently in a shared memory segment instead of writing files.

4.2 Architecture

The architecture of Damaris is shown in Figure 6, representing a multicore SMP node in which one core is dedicated to Damaris, the other cores (only three represented here) being used for computation. As the number of cores per CPU increases, dedicating one core has a diminishing impact. Thus, our approach primarily targets SMP nodes featuring more than 8 cores.

As indicated above, the I/O pattern of large-scale applications is predictable, thus additional knowledge regarding the data that is expected to be written can be provided by the user. An important component in the design of Damaris is the external configuration file. This file is written by the user and provides a

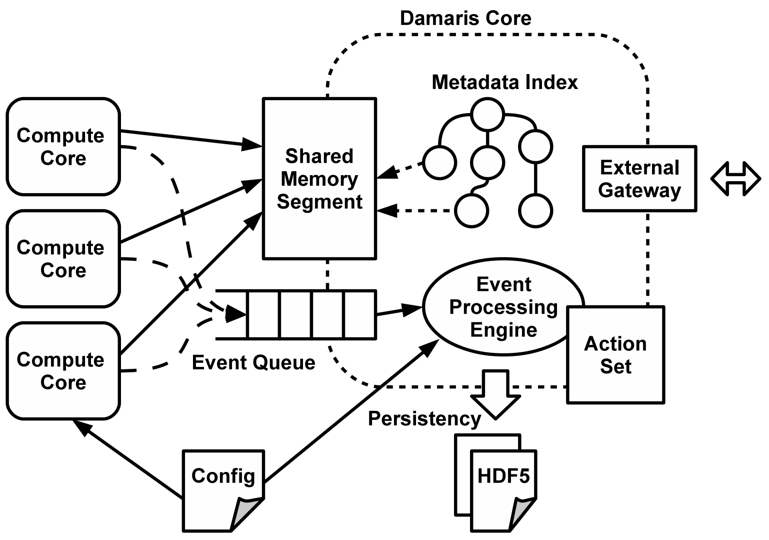

Figure 6: Design of the Damaris approach.

model of the data generated by the application. It follows an idea introduced by the ADIOS data format in [14], which allows the configuration of the I/O layers to be into an external file, avoiding code recompilation when the user wants a different output. However, not only data-related information is provided by our configuration file. As we will further explain, each component in the design takes advantage of *a priori* knowledge.

Communication between the computation cores and the dedicated cores is done through shared memory. A large memory buffer is created by the dedicated core at start time, with a size chosen by the user. When a client submits new data, it requests the allocation of a segment from this buffer, then copies its data using the returned pointer. Currently a mutex-based allocation algorithm is used in order to allow concurrent atomic allocations of segments from multiple clients. Even though this current allocation solution is already efficient, we are investigating possible algorithms leveraging the *a priori* knowledge provided by the user in order to preallocate segments and avoid locking.

The event-queue is another shared component of the Damaris architecture. It is used by the clients either to inform the server that a write completed (*write-notification*), or to send *user-defined events*. The messages are pulled by an event processing engine (EPE). The configuration file provided by the user contains actions to be performed upon reception of write-notifications and user-defined events. If no action is defined for a given user-defined event, the EPE just ignores it. Actions can be provided by the user through a plugin system. Such actions can prepare data for future analysis, for instance.

All datasets written by the clients are uniquely characterized by a tuple ⟨*name,iteration,source,layout*⟩. *Iteration* gives the current step of the simulation, while *source* uniquely characterizes the client that has written the variable. The *layout* corresponds to a description of the structure of the data, for instance "3D array of 32 bit real values with extents n_x, n_y, n_z". For most simulations, this layout does not vary during runtime and can be provided also by the configuration file. Upon reception of a write-notification, the EPE will add an entry in a metadata structure associating the tuple with the received data (that stay in shared memory until actions are performed on them).

4.3 Features

Behavior management and user-defined actions - The behavior manager can be enriched by plugins provided by the user. A plugin is a function (or a set of functions) embedded in a dynamic library that the behavior manager will load and call in response to events sent by the application. The matching between events and expected reactions is provided by the external configuration file.

Access semantics - The use of a single message queue shared by all the cores and used both for user-defined events and write events ensure that a sequence of events sent by a compute core will be treated in the same order in the dedicated core. Yet, sequences of events coming from different compute cores may interleave. This semantic is important because the purpose of user-defined events is to force a behavior at a given execution point, knowing what has already been written.

Event scope - It can be desirable to force a reaction only when a given subset of computation cores have sent an event (which is a stronger condition than the simple access semantics presented above). This can be done using the notion of event scope, which defines the subset of processes that has to perform a synchronized action, such as statistical data sharing for establishing a global diagnosis of the simulation. This synchronization involves distributed algorithms such as total order broadcast, and efficient scheduling schemes, that are beyond the scope of this paper.

Persistency layer configuration - Damaris acts as an in-memory data staging service. When the local memory is full, some data has to be either thrown away or persistently stored in the file system. The behavior of the system with respect to what should or should not be stored is also defined in the external configuration file. Moreover, this behavior can be changed by the application itself through events or by external applications such as visualization tools connected to the servers. Critical data corresponding to an application's checkpoint can be stored immediately, while other data may reside longer in memory.

4.4 API

Damaris is intended to be a generic, platform-independent, application-independent, easy-to-use tool. The current implementation is developed in C++ and uses the Boost library for interprocess communications, options parsing and configuration. It provides client-side interfaces for C, C++ and Fortran applications and requires only few minor changes in the application's I/O routine, together with an external configuration file that describes the data. The client-side interface is extremely simple and consists in four main functions (here in Fortran):

`dc_init("configuration.xml",core_id,ierr)` initializes the client by providing a configuration file and a client ID (usually the MPI rank). The configuration file will be used to retrieve knowledge about the expected data layouts.

`dc_write("varname",step,data,ierr)` pushes some data into the Damaris' shared buffer and sends a message to Damaris notifying the incoming data. The dataset being sent is characterized by its name and the corresponding iteration of the application. All additional information such as the size of the data its layout and additional descriptions are provided by the configuration file.

`dc_signal("eventname",step,ierr)` sends a custom event. to Damaris in order to force a predefined behavior. This behavior is been defined in the configuration file.

`dc_finalize(ierr)` frees all resources associated with the client. The server is also notified of client's disconnection and will not accept any other incoming data or event from this client.

Additional functions are available to allow direct access to an allocated portion of the shared buffer, avoiding an extra copy from local memory to shared memory. Other functions allow the user to dynamically modify the internal configuration initially provided, which can be useful when writing variable-length arrays (which is the case in particle-based simulations, for example).

Below is an example of a Fortran program that makes use of Damaris to write a 3D array then send an event to the I/O core. The associated configuration file, which follows, describes the data that is expected to be received by the I/O core, and the action to perform upon reception of the event. This action is loaded from a dynamic library provided by the user.

```
program example
  integer :: ierr, rank, step
  real, dimension(64,16,2) :: my_data
  ...
  call dc_initialize("my_config.xml", rank, ierr)
  call dc_write("my_variable", step, my_data, ierr)
  call dc_signal("my_event", step, ierr)
  call dc_finalize(ierr)
  ...
end program example
```

```
myconfig.xml:
<layout name="my_layout" type="real"
        dimensions="64,16,2" language="fortran" />
<variable name="my_variable" layout="my_layout" />
<event name="my_event" action="do_something"
       using="my_plugin.so" scope="local" />
```

Damaris interfaces with an I/O library such as HDF5 by using a custom persistency layer with HPF5 callback routines.

5 Experimental results with CM1

We present in this section the evaluation of our approach based on dedicated I/O cores, as compared with previous approaches with the CM1 atmospheric simulation, using two different platforms.

5.1 The CM1 application

CM1 [3] is used for atmospheric research and is suitable for modeling small-scale atmosphere phenomena such as thunderstorms and tornadoes. It follows a typical behavior of scientific simulations which alternate computation phases and I/O phases. The simulated domain is a fixed 3D array representing part of the atmosphere. The number of points along the x, y and z axes is given by the parameters n_x, n_y and n_z. Each point in this domain is characterized by a set of variables such as local *temperature* or *wind speed*. The user can set which of these variables have to be written to disk for further analysis.

CM1 is written in Fortran 95. Parallelization is done using MPI, by splitting the 3D array along a 2D grid. Each MPI rank is mapped into a pair (i, j) and simulates a $n_{sx} * n_{sy} * n_z$ points subdomain. In the current release (r15), the I/O phase uses HDF5 to write one file per process. Alternatively, the latest development version allows the use of pHDF5. One of the advantages of using a file-per-process approach is that compression can be enabled, which is not the case with pHDF5. Using lossless gzip compression on the 3D arrays, we observed a compression ratio of 187%. When writing data for offline visualization, the floating point precision can also be reduced to 16 bits, leading to nearly 600% compression ratio when coupling with gzip. Therefore, compression can significantly reduce I/O time and storage space. However, enabling compression leads to an additional overhead, together with more variability both from a process to another, and from a write phase to another.

5.2 Platforms and configuration

BluePrint is the Blue Waters interim system running at NCSA. It provides 120 nodes, each featuring 16 1.9GHz POWER5 cpus, and runs the AIX operating system. 64 GB of local memory is available on each node. GPFS is deployed on 2 dedicated nodes. CM1 was run on 64 nodes (1024 cores), with a 960x960x300 points domain. Each core handled a 30x30x300 points subdomain with the standard approach. When dedicating one core out of 16 on each node, computation cores handled a 24*40*300 points subdomain.

Grid'5000 has already been described in Section 3.1.2. We run the development version of CM1 on 28 nodes of the *parapluie* cluster, with PVFS running on 16 nodes of the *parapide* cluster. Thus CM1 runs on 672 cores. The total domain size is 1104x1120x200 points, each core handling a 46x40x200 points subdomain.

5.3 Non-overlapping I/O approaches

Our first investigations on optimizing I/O for CM1 started with standard approaches that do not overlap I/O with computation.

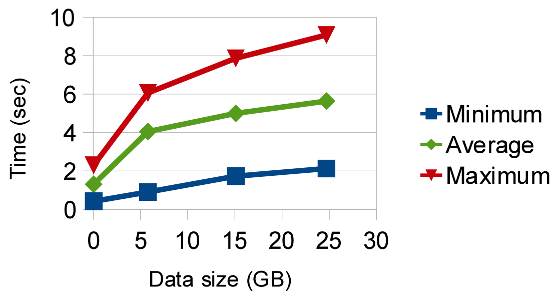

Figure 7: Maximum, minimum an average write time on BluePrint with different total output sizes, using the file-per-process approach, compression enabled.

5.3.1 Standard file-per-process method

CM1 allows as input the ability to set flags to indicate whether a variable should be written or not. Thus, we have evaluated the behavior the CM1's file-per-process method on BluePrint with different output configurations, ranging from a couple of 2D arrays to a full backup of all 3D arrays. Compression was enabled, but the amount of data will be reported as logical data instead of physical bytes stored.

The results are presented in Figure 7. For each of the four experiments, CM1 runs during one hour and writes every 3 minutes. We plot the average values of the time spent by the slowest process in I/O phase, the fastest, and the mean. We observe that the slowest process is almost five times slower than the fastest.

On Grid5000, we run CM1 with an output of 15.8 GB per backup consisting in the set of variables usually required to perform offline visualization. Data were not compressed. CM1 reported spending 4.22% of its time in I/O phases. Yet the fastest processes usually terminate their I/O in less than 1 sec, while the slowest take more than 25 sec, leading to an aggregate throughput of 695 MB/s on average and a standard deviation of 74.5 MB/s. Following the insights of Section 3, we increased the stripe size to 32 MB in PVFS. The variability pattern for one of the I/O phases is shown in Figure 8. This pattern is recurrent and representative of the global behavior.

Note that, since CM1 uses HDF5 or pHDF5, we cannot have complete control over the data layout and on the number of atomic write calls to MPI subroutines. As this layout is far from the ideal cases previously presented with the IOR benchmark, the achieved throughput is much lower than what could be expected from our I/O subsystems. Moreover, the presence of many all-to-all operations within pHDF5 functions forces a high level of synchronization, which does not allow us to have a process-level view of the time spent in I/O.

5.3.2 One step ahead: using collective I/O and grouped I/O

The current development version of CM1 provides a custom pHDF5 backend that can gather MPI processes in groups. Each group creates and writes in a distinct file. This way, the overall number of files can be reduced without the

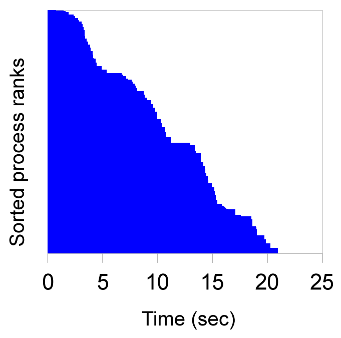

Figure 8: Variability pattern from an output of CM1 using the file-per-process approach on Grid'5000.

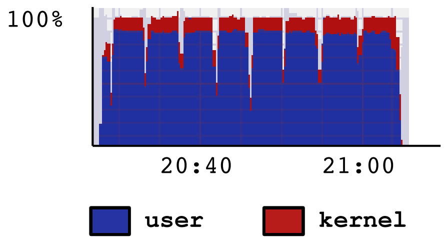

Figure 9: CPU usage measured on a node running CM1 with a collective I/O mechanism.

potential burden of a heavy synchronization of all processes. We thus tested CM1 on Grid'5000, writing either one shared file, or one file per node instead of one per core.

When all processes write to a single shared file, CM1 shows poor results (636 MB/s of aggregate throughput on average) and a relatively high variability, with a 101.8 MB standard deviation. The global shared file approach shows worse performance than the file-per-process approach, both in terms of average performance and performance variability. Figure 9 shows CPU usage on one of the nodes used by CM1 during the execution. Non-optimal use of computing resources can be noticed during I/O.

Writing one file per node using collective I/O leads to a small improvement: 750 MB/s on average. An expected result is the decrease of the associated standard deviation, from 101.8 MB to 36.8 MB. The improvement in standard deviation may not be significant, as the small number of runs does not lead to small enough confidence intervals. The output when using the grouped I/O produces 28 files instead of 672. But the compression capability is still lost due to the use of pHDF5. Thus, we think that such a grouped I/O approach might be advantageous only at extremely large scales, when both the file-per-process approach and the collective I/O approach show their limits.

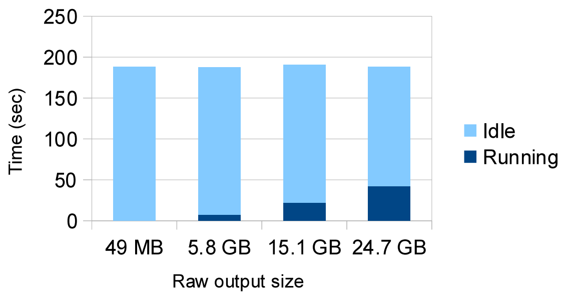

Figure 10: Write time and idle time in the I/O cores on BluePrint. Data size is the raw uncompressed data output from all dedicated cores together

5.4 Our proposal: The Damaris approach: using dedicated I/O cores

In this section, we evaluate the I/O performance and variability when using one dedicated core. Experiments are performed on BluePrint and Grid'5000 and compared to the previous results.

5.4.1 Benefits of the dedicated I/O cores

On BluePrint, we compared the file-per-process approach with the dedicated-core approach, compression enabled in both cases. By mapping contiguous subdomains to cores located in the same node, we have also been able to implement a filter that aggregates datasets prior to actual I/O. The number of files is divided by 16, leading to 64 files created per I/O phase instead of 1024. With a dedicated core, CM1 reports spending less than 0.1 sec in the I/O phase, that is to say only the time required to perform a copy of the data in the shared buffer. We measured the time that dedicated cores spent writing depending on the amount of data. This time is reported on Figure 10 and compared to the time spent by the other cores to perform computations between two I/O phases. We observe that even with the largest amount of data, the I/O core stays idle 75% of the time, thus allowing more data post-processing in order to help further analysis.

We did a similar experiment on Grid'5000, with one core out of 24 dedicated to I/O. CM1 writes every 20 iterations, that is to say about every 3 minutes of wallclock time. Figure 11 shows the CPU usage in a node running CM1. The idle phases corresponding to I/O phases have disappeared and the CPU is constantly fully utilized. The version of Damaris running here does not aggregate datasets, but writes them in a common file per node, thus reducing the number of files from 672 to 28. The I/O cores report to achieve a throughput of 4.32 GB/s, that is to say about 6 times higher than the throughput achieved with the grouped I/O and leading to the same number of files. This throughput presents a standard deviation of 319 MB/s, but this jitter is hidden from the computation cores.

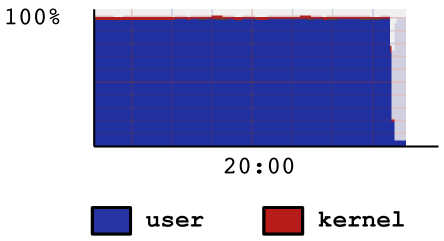

Figure 11: CPU usage measured on a node running CM1 with Damaris.

We thus notice an improvement on both platforms together with a time savings that could be dedicated to data post-processing. CM1 also integrates a communication-intensive phase that periodically computes statistics in order to be able to stop the simulation if some values become physically irrelevant. This phase makes use of several calls to MPI_Gather and MPI_Reduce that cause a high overhead, reported to be more than 7% of walltime on Grid'5000. By moving this diagnosis into the dedicated cores, we could achieve a 7% additional speedup, to be added to the speedup already achieved by removing the I/O part.

5.4.2 Adding compression and scheduling

As an example of the extra degrees of freedom enabled by our approach, we have added two features into our implementation: compression and I/O scheduling. On Grid'5000, with the deployment configuration previously described, a gzip compression level 4 reduced the amount of data from 173.35 GB to only 19.8 GB for the first 200 iterations of the simulation.

The write time is of course higher and more variable, with an average of 21.9 sec and a variance of 58.64 for 308 measures. But this time is still hidden in the I/O core, thus not perceptible from the point of view of the application. In other words, we have an overhead-free compression feature that allows a more than 600% compression factor.

The second feature we added in our implementation is the capacity to schedule data movements. Instead of writing a file as soon as all the required variable are available in the I/O core, the I/O cores compute an estimation of the computation time of an iteration. This time is then divided into as many slots as I/O cores, and each I/O core waits for its slot before writing. Contrary to what we expected, the average time to perform a write was not changed compared to the non-scheduling version of Damaris. Yet this feature may show its benefit when we go to hundreds of thousands of cores. Moreover, we plan to add the possibility for dedicated cores serving different applications to interact in order to have a coupled behavior, at the scale of a whole platform.

5.4.3 Benefits of leaving one core out of computation

Let us call W the time spent writing, C_{std} the computation time of an iteration (without the write phase) with a standard approach, and C_{ded} the computation time when the same workload is divided across one less core. We here assume that the I/O time is completely hidden or at least negligible when using the dedicated core. A theoretical performance benefit of our approach then occurs when

$$W + C_{std} > C_{ded}$$

Assuming an optimal parallelization of the program across N cores per node, we show that this inequality is true when the program spend at least $p\%$ in I/O phase, with $p = \frac{100}{N-1}$. As an example, with 24 cores $p = 4.35\%$, which is already under the 5% usually admitted for the I/O phase of such applications.

However this is a simplified theoretical estimation. In practice, we have run CM1 with the exact same configuration and network topology, dividing the workload across 24 cores per node first, then 23 cores per node. 200 iterations with 24 cores per node where done in 44'17" while only 41'46" were necessary to run 200 iterations with 23 cores per node. The final statistics output by CM1 showed that all the phases that use all-to-all communications benefit from leaving one core out, thus actually reducing the computation time as memory contention is reduced. Such a behavior of multicore architectures is explained in [15].

6 Conclusions

Managing I/O variability in an efficient way can have a substantial impact on the ability to sustain a high performance on Petascale and Post-Petascale infrastructures. The impact of I/O jitter on the overall HPC application performance is already observed at smaller scales, and understanding its behavior and proposing an efficient mechanism to reduce its effects is critical for preparing the advent of Post-Petascale machines. The contributions of this paper can be summarized as follows: 1) We propose a method to visualize some I/O variability patterns and quickly retrieve insights regarding the impact of user-controllable parameters on this variability. Using the IOR benchmark on the Grid'5000 testbed, we show that choosing the appropriate I/O approach and the right value for these parameters can substantially change the I/O throughput by an order of magnitude. 2) After an in-depth discussion of the limitations of standard I/O approaches regarding both I/O throughput and I/O performance variability, we propose a new approach (called Damaris) which leverages dedicated I/O cores in multicore SMP nodes. This solution provides the capability to better schedule data movement, but also to process the data prior to storage. Our implementation, tested with the CM1 atmospheric model on the French Grid'5000 testbed and on BluePrint at NCSA, proved to be effective by completely hiding the I/O costs, achieving a throughput 6 times higher than standard approaches, and allowing overhead-free data compression with up to 600% compression ratio.

Our future work will focus in several directions. We plan to quantify the optimal ratio between I/O cores and computation cores within a node for several classes of HPC simulations. Grid'5000 provides an excellent investigation testbed in this direction, as it provides many-cores SMP nodes. We will also investigate ways to leverage spare time in the I/O cores. A very promising direction is attempting a tight coupling between running simulations and visualization engines, enabling direct access to data by visualization engines (through the I/O cores) while the simulation is running. This could serve to achieve efficient inline visualization without blocking the simulation. The BlobSeer approach [17] of lock-free, concurrency-optimized data management could serve as a basis to make progress on this path. Another direction is the coordination of the I/O cores in order to implement a distributed scheduling of the application's I/O. We also intend to investigate the possibility for multiple independent applications running on the same platform to communicate through their I/O cores in order to make better decisions with respect to scheduling accesses to the file system or to other shared resources.

Acknowledgments

This work was done in the framework of a collaboration between the KerData INRIA - ENS Cachan/Brittany team (Rennes, France) and the NCSA (Urbana-Champaign, USA) within the Joint INRIA-UIUC Laboratory for Petascale Computing. The experiments presented in this paper were carried out using the Grid'5000/ALADDIN-G5K experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see `http://www.grid5000.fr/` for details) and on BluePrint, the IBM Power5 interim machine of the Blue Waters project [1]. The authors also acknowledge the PVFS developers, the HDF5 group and the Grid'5000 users, who helped properly configuring the tools and platforms used for this work. We acknowledge Robert Wilhelmson (NCSA, UIUC) for his insights on the CM1 application and Dave Semeraro (NCSA, UIUC) for our fruitful discussions on visualization/simulation coupling.

References

[1] IBM's BlueWaters, `http://www.ncsa.illinois.edu/BlueWaters/`, viewed on December 2010.

[2] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan. Scalable I/O forwarding framework for high-performance computing systems. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –10, September 2009.

[3] G. H. Bryan and J. M. Fritsch. A benchmark simulation for moist nonhydrostatic numerical models. *Monthly Weather Review*, 130(12):2917–2928, 2002.

[4] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig. Small-file access in parallel file systems. *Parallel and Distributed Processing Symposium, International*, 0:1–11, 2009.

[5] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur. Pvfs: a parallel file system for linux clusters. In *Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*, pages 28–28, Berkeley, CA, USA, 2000. USENIX Association.

[6] C. Chilan, M. Yang, A. Cheng, and L. Arber. Parallel I/O performance study with HDF5, a scientific data package, 2006.

[7] A. Ching, A. Choudhary, W. keng Liao, R. Ross, and W. Gropp. Noncontiguous i/o through pvfs. volume 0, page 405, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[8] P. M. Dickens and R. Thakur. Evaluation of Collective I/O Implementations on Parallel Architectures. *Journal of Parallel and Distributed Computing*, 61(8):1052 – 1076, 2001.

[9] C. Docan, M. Parashar, and S. Klasky. Enabling high-speed asynchronous data extraction and transfer using DART. *Concurrency and Computation: Practice and Experience*, 22(9):1181–1204, 2010.

[10] S. Donovan, G. Huizenga, A. J. Hutton, C. C. Ross, M. K. Petersen, and P. Schwan. Lustre: Building a file system for 1000-node clusters, 2003.

[11] F. Isaila, J. G. Blas, J. Carretero, R. Latham, and R. Ross. Design and evaluation of multiple level data staging for Blue Gene systems. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2010.

[12] J. Li, W. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. page 39. IEEE, 2006.

[13] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf. Managing variability in the IO performance of petascale storage systems. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[14] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, CLADE '08, pages 15–24, New York, NY, USA, 2008. ACM.

[15] S. Moore. Multicore is bad news for supercomputers. *Spectrum, IEEE*, 45(11):15–15, 2008.

[16] MPICH2. `http://www.mcs.anl.gov/research/projects/mpich2/`.

[17] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie. BlobSeer: Next Generation Data Management for Large Scale Infrastructures. *Journal of Parallel and Distributed Computing*, 71(2):168–184, Feb. 2011.

[18] A. Nisar, W. keng Liao, and A. Choudhary. Scaling parallel I/O performance through I/O delegate and caching system. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1 –12, November 2008.

[19] C. M. Patrick, S. Son, and M. Kandemir. Comparative evaluation of overlap strategies with study of I/O overlap in MPI-IO. *SIGOPS Oper. Syst. Rev.*, 42:43–49, October 2008.

[20] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges. MPI-IO/GPFS, an Optimized Implementation of MPI-IO on Top of GPFS. volume 0, page 58, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[21] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis. Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, page 17, november 2006.

[22] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, pages 231–244. Citeseer, 2002.

[23] H. Shan and J. Shalf. Using IOR to Analyze the I/O performance for HPC Platforms. In *Cray User Group Conference 2007*, Seattle, WA, USA, 2007.

[24] D. Skinner and W. Kramer. Understanding the causes of performance variability in HPC workloads. volume 0, pages 137–149, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[25] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. *Frontiers of Massively Parallel Processing, Symposium on the*, 0:182, 1999.

[26] A. Uselton, M. Howison, N. Wright, D. Skinner, N. Keen, J. Shalf, K. Karavanic, and L. Oliker. Parallel i/o performance: From events to ensembles. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –11, april 2010.

[27] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. PreDatA – preparatory data analytics on peta-scale machines. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, April 2010.

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399