Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O

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

Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O

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

***Abstract*—With exascale computing on the horizon, the performance variability of I/O systems represents a key challenge in sustaining high performance. In many HPC applications, I/O is concurrently performed by all processes, which leads to I/O bursts. This causes resource contention and substantial variability of I/O performance, which significantly impacts the overall application performance and, most importantly, its predictability over time. In this paper, we propose a new approach to I/O, called Damaris, which leverages dedicated I/O cores on each multicore SMP node, along with the use of shared-memory, to efficiently perform asynchronous data processing and I/O in order to hide this variability. We evaluate our approach on three different platforms including the Kraken Cray XT5 supercomputer (ranked 11th in Top500), with the CM1 atmospheric model, one of the target HPC applications for the Blue Waters postpetascale supercomputer project. By overlapping I/O with computation and by gathering data into large files while avoiding synchronization between cores, our solution brings several benefits: 1) it fully hides jitter as well as all I/O-related costs, which makes simulation performance predictable; 2) it increases the sustained write throughput by a factor of 15 compared to standard approaches; 3) it allows almost perfect scalability of the simulation up to over 9,000 cores, as opposed to state-of-the-art approaches which fail to scale; 4) it enables a 600% compression ratio without any additional overhead, leading to a major reduction of storage requirements.**

***Index Terms*— Exascale Computing; Multicore Architectures; I/O; Variability; Dedicated Cores**

I. 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 input/output (I/O) operations. A typical behavior in large-scale simulations consists of alternating computation phases and write phases. As a rule of thumb, it is commonly accepted that a simulation spends at most 5% of its run time in I/O phases. Often due to explicit barriers or communication phases, all processes perform I/O at the same time, causing network and file system contention. It is commonly observed that some processes exploit a large fraction of the available bandwidth and quickly terminate their I/O, then remain idle (typically from several seconds to several minutes) waiting for slower processes to complete their I/O. This jitter can even be observed at relatively small scale, where measured I/O performance can vary by several orders of magnitude between the fastest and slowest processes [30]. With multicore architectures, this variability becomes even more of a problem, as multiple cores in a same node compete for the network access. This phenomenon is exacerbated by the fact that HPC resources are typically used by many concurrent I/O intensive jobs. This creates file system contention between jobs, further increases the variability from one I/O phase to another and leads to 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 ranging 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 [30] 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 substantially differ due to multiple factors, including the platform, the underlying file system, the network, and the I/O pattern of the application. For instance, using a single metadata server in the Lustre file system [11] causes a bottleneck when following the *file-per-process* approach (described in Section II-B), a problem that PVFS [5] or GPFS [27] are less likely to 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 [25]. Yet these optimization usually rely on all-to-all communications that impact their scalability.

The main contribution of this paper is precisely to propose an approach that completely hides the I/O jitter exhibited by most widely used approaches to I/O management in HPC simulations: the *file-per-process* and *collective-I/O* approaches (described in Section II). Based on the observation that a first level of contention occurs when all cores of a multicore SMP node try to access the network for intensive I/O at the same time, our new approach to I/O, called Damaris (Dedicated Adaptable Middleware for Application Resources Inline Steering), leverages *a dedicated I/O core in each multicore SMP node along with shared memory to perform asynchronous data processing and I/O*. These key design choices build on the observation that 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 not only help reducing jitter but also increase overall performance. Besides, most data written by

HPC applications are only eventually read by analysis tasks but not used by the simulation itself. Thus write operations can be delayed without consistency issues. Damaris takes into account user-provided information related to the application behavior and the intended use of the output in order to perform "smart" I/O and data processing within SMP nodes. Some of these ideas have been partially been explored in other efforts parallel to ours: a detailed positioning of Damaris with respect to related work is given in Section V-B.

We evaluate the Damaris approach with the CM1 application (one of the target applications for the Blue Waters [22] project) on three platforms, each featuring a different file system: Grid'5000 with PVFS, Kraken with Lustre and BluePrint, a Power5 cluster with GPFS. We compare Damaris to the classical file-per-process and collective-I/O approaches that have shown to greatly limit the scalability of the CM1 application and motivated our investigations. By using shared memory and by gathering data into large files while avoiding synchronization between cores, our solution achieves its main goal of fully hiding the I/O jitter. It thus allows the application to have a predictable run time and a perfect scalability. It also increases the I/O throughput by a factor of 6 compared to standard approaches on Grid'5000, and by a factor of 15 on Kraken, hides all I/O-related costs and enables a 600% compression ratio without any additional overhead. To the best of our knowledge, no concurrent approach has achieved such improvements.

This paper is organized as follows: Section II presents the background and motivations of our study. Damaris is presented in Section III together with an overview of its design and its API. In Section IV we evaluate this approach with the CM1 atmospheric simulation running on 672 cores of the *parapluie* cluster on Grid'5000, 1024 cores of a Power5 cluster and on up to 9216 cores on Kraken. Section V presents a study of the theoretical and practical benefits of the approach. This section also explains the originality of our Damaris approach with respect to related work. Finally, Section VI summarizes the contribution and discusses future directions.

II. Background and motivations

A. Understanding I/O jitter

Over the past several years, chip manufacturers have increasingly focused on multicore architectures, as the increase of clock frequencies for individual processors has leveled off, primarily due to substantial increases in power consumption. High performance variability across individual components of these more complex systems 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 runtime of applications. In [28], 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, causing synchronization between processes that run within the 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, issues 1 and 2 can be better handled by tuning large-scale applications in such a 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 [30]. While the performance of computation phases of HPC applications is usually stable and only suffers 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 [17], variability is expressed in terms of *interferences*, with the distinction between *internal interferences* caused by access contention between the processes of an application (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 [17] to limit access contentions and allow a higher and less variable I/O throughput.

B. Approaches to I/O management in HPC simulations

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

*a) **The** **file-per-process** **approach**:* This approach consists of 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 when scaling to hundreds of thousands of files: special optimizations are then necessary [4]. File systems using a single metadata server, such as Lustre, suffer from a bottleneck: simultaneous creations of so many files are serialized, which leads to immense I/O variability. Moreover, reading such a huge number of files for post-processing and visualization becomes intractable.

*b) **Using** **collective I/O**:* In MPI applications utilizing collective I/O, 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 [25]. Algorithms termed as "two-phase I/O" [9], [29] enable efficient collective I/O implementations by aggregating requests and by adapting the write pattern to the file layout across multiple data servers [7]. Collective I/O avoids metadata redundancy as opposed to the file-per-process approach. However, it imposes additional process synchronization, leading to potential loss of efficiency in I/O operations. In addition, none of today's data formats offers compression features using this approach. Intuitively and experimentally [12], any approach that uses synchronization between processes as in collective-I/O, is more likely to reduce I/O variability from one process to another in a single write

phase, but at the price of additional synchronizations that can limit the global I/O throughput and introduce variability from a write phase to another.

It is usually possible to switch between these approaches when a scientific format is used on top of MPI; going from HDF5 to pHDF5 [6] is a matter of adding a couple of lines of code, or simply changing the content of an XML file with ADIOS [18]. 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. In addition, the aforementioned approaches create periodic peak loads in the file system and suffer from contention at several levels. This first happens at the level of each multicore SMP node, as concurrent I/O requires all cores to access remote resources (networks, I/O servers) at the same time. Optimizations in collective-I/O implementations are provided to avoid this first level of contention; e.g. in ROMIO [8], data aggregation is performed to gather outputs from multiple processes to a subset of processes that interact with the file system by performing larger, contiguous writes. Yet with the ever growing number of cores, these "two phases" I/O optimizations based on communications are unlikely to scale well, inviting to consider ways for small groups of processes to concurrently and efficiently perform I/O, without the need for synchronization.

III. The Damaris approach

To sustain a high throughput and a lower variability, it is preferable to avoid as much as possible access contentions at the level of the network interface and of the file system, for example by reducing the number of writers (which reduces the network overhead and allows data servers to optimize disk accesses and caching mechanisms) and the number of generated files (which reduces the overhead on metadata servers). As the first level of contention occurs when several cores in a single SMP node try to access the same network interface, it becomes natural to work at the level of a node.

We propose to gather the I/O operations into one single core that will perform writes of larger data in each SMP node. In addition, this core is dedicated to I/O (i.e. will not run the simulation code) in order to overlap writes with computation and avoid contention for accesses to the file system. The communication between cores running the simulation and dedicated cores is done through shared-memory, to make a write as fast as a `memcpy`. We call this approach Damaris. Its design, implementation and API are described below.

A. Principle

Damaris consists of a set of MPI processes running on a set of dedicated cores (typically one) in every SMP node used by the simulation. Each dedicated process keeps data in a shared memory segment and performs post-processing, filtering, indexing and finally I/O in response to user-defined events sent either by the simulation or by external tools.

The buffering system running on these dedicated cores includes metadata information about incoming variables. In other words, clients do not write raw data but enriched datasets in a way similar to scientific data formats such as HDF5 or NetCDF. Thus dedicated cores have full knowledge of incoming datasets and can perform "smart actions" on these data, such as writing important datasets in priority, performing

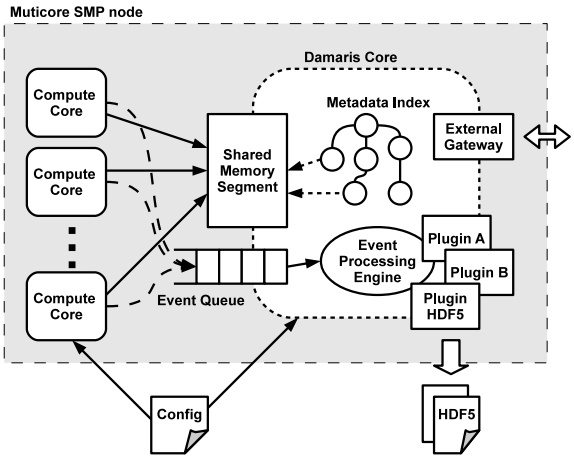


Fig. 1. Design of the Damaris approach.

compression, statistical studies, indexing, or any user-provided transformation. These transformations are provided by the user through a plugin system, which makes the system fully adaptable to the particular requirements of an application. By analyzing the data, the notion of an "important dataset" can be based on the scientific content of the data and thus dynamically computed, a task that low-level I/O schedulers could not perform.

B. Architecture

Figure 1 presents the architecture of Damaris, by representing a multicore SMP node in which one core is dedicated to Damaris. The other cores (only three represented here) are used by the simulation. As the number of cores per node increases, dedicating one core has a diminishing impact. Thus, our approach primarily targets SMP nodes featuring a large number of cores per node (12 to 24 in our experiments).

Shared-memory: 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. Thus the user has a full control over the resources allocated to Damaris. When a compute core submits new data, it reserves a segment of this buffer, then copies its data using the returned pointer, so the local buffer can be reused. Damaris uses the default mutex-based allocation algorithm of the Boost library to allow concurrent atomic reservation of segments by multiple clients. We also implemented another lock-free reservation algorithm: when all clients are expected to write the same amount of data, the shared-memory buffer is split in as many parts as clients and each client uses its own region.

Configuration file: To avoid using the shared memory to transfer too much metadata information, Damaris uses an external configuration file to provide static information about the data (such as names, description, unit, dimensions...). This design principle is directly inspired by ADIOS [18] and also present in many other tools such as EPSN [13]. The goals of this external configuration are 1) to keep a high-level description of the datasets within the server, allowing higher-level data manipulations, 2) to avoid static layout descriptions to be sent by clients through the shared memory (only data is sent together with the minimal descriptor that lets the server retrieve the full description in its metadata system). Additionally, it helps defining the behavior of dedicated cores

through the configuration of actions to be performed on data prior to storage.

Event queue: The event-queue is another shared component of the Damaris architecture. It is used by 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) on the server side. The configuration file also includes information about the actions to perform upon reception of an event. Such actions can prepare data for future analysis, or simply write it using any I/O library.

Metadata management: All variables written by the clients are characterized by a tuple ⟨*name,iteration,source,layout*⟩. *Iteration* gives the current step of the simulation, while *source* uniquely characterizes the sender (e.g. its MPI rank). The *layout* is a description of the structure of the data: type, number of dimensions and extents. For most simulations, this layout does not vary at 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. The data stay in shared memory until actions are performed on them.

C. Key design choices

Behavior management and user-defined actions: The EPE can be enriched by plugins provided by the user. A plugin is a function embedded in the simulation, in a dynamic library or in a Python script, that the EPE 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. Thus, it is easy for the user to define a precise behavior for Damaris by simply changing the configuration file. Damaris was designed with the intent to provide a very simple way for users to extend it and adapt it to the particular needs of their simulations.

Minimum-copy overhead: The efficiency of interactions between clients and dedicated cores is another strength of Damaris. At most a single copy from a local variable to the shared memory buffer is required to send data to the dedicated core. Damaris also includes the possibility to "write" data without actually making a copy: the simulation directly allocates its variables in the shared memory buffer. When the simulation finishes working on an array, it simply informs the dedicated core that the data can be considered as ready. In a context of a shrinking memory/FLOP ratio, offering this optimization can be crucial for some applications. This is a strong point of Damaris that distinguishes it from other dedicated-process-based approaches [19], [16], further described in Section V-B.

Persistency layer: Finally our implementation of Damaris interfaces with HDF5 by using a custom persistency layer embedded in a plugin, as shown as an example in Figure 1.

D. Client-side 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, and Xerces-C for XML configuration. It provides client-side interfaces for C, C++ and Fortran applications which can be summarized by four main functions (here in C):

- `df_initialize` and `df_finalize` initialize and free the resources used by Damaris.
- `df_write("varname",step,data)` copies the data in shared memory along with minimal information and notifies the server. All additional information such as the size of the data and its layout (including its datatype) are provided by the configuration file.
- `df_signal("eventname",step)` sends a custom event to the server in order to force a behavior predefined in the configuration file.

Additional functions are available to allow direct access to an allocated portion of the shared buffer (`dc_alloc` and `dc_commit`), avoiding an extra copy from local memory to shared memory. Other functions let the user write arrays that don't have a static shape (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.

```
program example
  integer :: rank, step
  real, dimension(64,16,2) :: my_data
  call df_initialize("my_config.xml", rank)
  ...
  call df_write("my_variable", step, my_data)
  call df_signal("my_event", step)
  ...
  call df_finalize()
end program example
```

Associated XML configuration file:

```
<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 has been released as an open-source software [1] and is now used by several research groups, willing in improve the I/O performance of their applications and to provide them post-processing capabilities. As seen above, Damaris requires a few modifications in existing applications. We made this choice (as opposed to hiding everything under the MPI layer or in the operating system) as it offers more flexibility and can leverage the scientific semantics of data. Feedbacks from several independent users confirmed the simplicity of our API.

IV. Experimental evaluation

We evaluate our approach based on dedicated I/O cores against standard approaches (file-per-process and collective-I/O) with the CM1 atmospheric simulation, using three different platforms: Kraken, Grid'5000 and a Power5 cluster.

A. 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. Each point in this domain

is characterized by a set of variables such as local *temperature* or *wind speed*. CM1 is written in Fortran 95. Parallelization is done using MPI, by splitting the 3D array along a 2D grid of equally-sized subdomains that are handled by each process. The I/O phase uses HDF5 to write one file per process, or pHDF5 to write in a collective manner. One of the advantages of using a file-per-process approach is that compression can be enabled, which is not the case with pHDF5. However at large process counts, the file-per-process approach generates a huge number of files, making all subsequent analysis tasks intractable.

B. Platforms and configuration

- **Kraken** is a supercomputer at NICS, currently ranked 11th in the Top500. It features 9408 Cray XT5 compute nodes connected through a Cray SeaStar2+ interconnect. Each node has 12 cores and 16 GB of local memory. We study the scalability of different approaches, including Damaris. Thus, the problem size varies from an experiment to another. When all cores are used by the simulation, each process handles a $44 \times 44 \times 200$ points subdomain. Using Damaris, each non-dedicated core (11 out of 12) handles a $48 \times 44 \times 200$ points subdomain, thus making the total problem size equivalent.
- **Grid'5000** is a French grid testbed. We used its *parapluie* cluster (featuring 40 nodes of 2 AMD 1.7 GHz CPUs, 12 cores/CPU, 48 GB RAM) to run the CM1 simulation on 28 nodes (672 cores) and 38 nodes (912 cores). The *parapide* cluster (2 Intel 2.93 GHz CPUs, 4 cores/CPU, 24 GB RAM, 434 GB local disk) was used to deploy a PVFS file system on 15 nodes, used both as I/O server and metadata server. All nodes communicate through a 20G InfiniBand 4x QDR link connected to a common Voltaire switch. We use MPICH [21] with ROMIO [29] compiled against the PVFS library. The total domain size in CM1 is $1104 \times 1120 \times 200$ points, so each core handles a $46 \times 40 \times 200$ points subdomain with a standard approach, and a $48 \times 40 \times 200$ points subdomain when one core out of 24 is used by Damaris.
- **BluePrint** provides 120 Power5 nodes. Each node consists in 16 cores and features 64 GB of memory. The GPFS file system is deployed on 2 separate nodes. CM1 was run on 64 nodes (1024 cores), with a $960 \times 960 \times 300$ points domain. Each core handled a $30 \times 30 \times 300$ points subdomain with the standard approach. When dedicating one core out of 16 on each node, computation cores handled a $24 \times 40 \times 300$ points subdomain. On this platform we vary the size of the output by enabling or disabling variables. We enabled the compression feature of HDF5 for all the experiments done on this platform.

We ran CM1 with an output configuration and frequency corresponding to what can be expected by atmospheric scientists from such a simulation.

C. Experimental results

In this section, we present the results achieved in terms of I/O jitter, I/O performance and resulting scalability of the application. We also provide two improvements to Damaris.

1) Effect on I/O jitter: Figure 2 shows the average and maximum duration of an I/O phase on Kraken from the point of view of the simulation. It corresponds to the time between

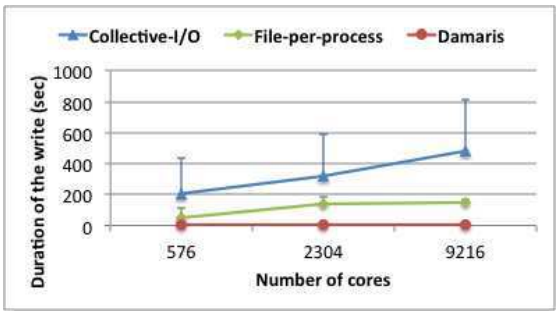


Fig. 2. duration of a write phase on Kraken (average and maximum). For readability reasons we don't plot the minimum write time. Damaris shows to completely remove the I/O jitter while file-per-process and collective-I/O have a big impact on the runtime predictability.

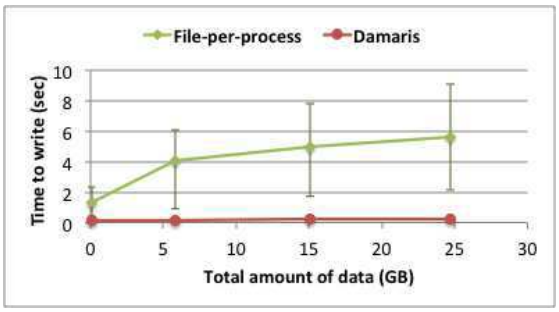
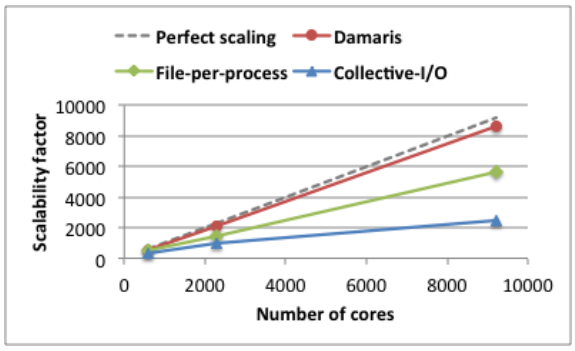


Fig. 3. duration of a write phase (average, maximum and minimum) using file-per-process and Damaris on BluePrint (1024 cores). The amount of data is given in total per write phase.

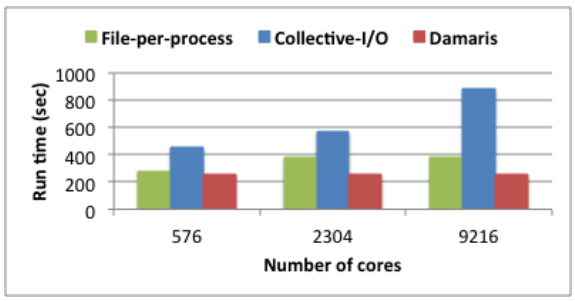
the two barriers delimiting the I/O phase. As we can see, this time is extremely high and variable with Collective-I/O, achieving up to 800 sec on 9216 processes. The average of 481 sec represents about 70% of the overall simulation's run time, which is simply unacceptable. By setting the stripe size to 32 MB instead of 1 MB in Lustre, the write time went up to 1600 sec with Collective-I/O, exemplifying the fact that bad choices of file system's configuration can lead to extremely poor I/O performance. Unexpectedly, the file-per-process approach seemed to lead to a lower variability, especially at large process count. Yet it still represents an unpredictability (difference between the fastest and the slowest phase) of about $\pm 17 sec$. For a one month run, writing every 2 minutes would lead to an uncertainty of several hours to several days of run time. When using Damaris, we dedicate one core out of 12 on each SMP node, thus potentially reducing the computation power for the benefit of I/O efficiency (the impact on overall application performance is discussed in the next section). As a means to reduce I/O jitter, this approach is clearly effective: the time to write from the point of view of the simulation is cut down to the time required to perform a series of copies in shared memory. It leads to a write time of 0.2 sec and does not depend anymore on the number of processes. The variability is in order of 0.1 sec (too small to be represented here).

On BluePrint, we vary the amount of data using the file-per-process approach. The results are presented in Figure 3. As we increase the amount of data, we increase the variability of the I/O time with the file-per-process approach. With Damaris however, the write time remains in the order of 0.2 sec for the largest amount of data and the variability in the order of 0.1 sec again.

Similar experiments have been carried out on Grid5000.

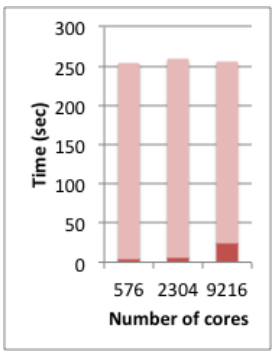


(a) Scalability factor on Kraken

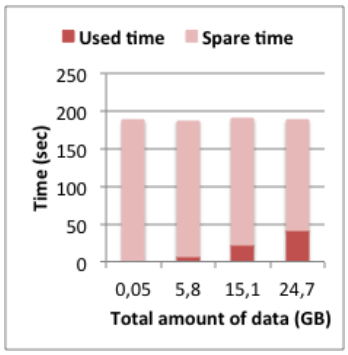


(b) Run time on Kraken

Fig. 4. (a) Scalability factor and (b) overall run time of the CM1 simulation for 50 iterations and 1 write phase, on Kraken.



(a) Write / Spare time on Kraken



(b) Write / Spare time on BluePrint

Fig. 5. Time spent by the dedicated cores writing data for each iteration, and time spared. The spare time is the time dedicated cores are not performing any task.

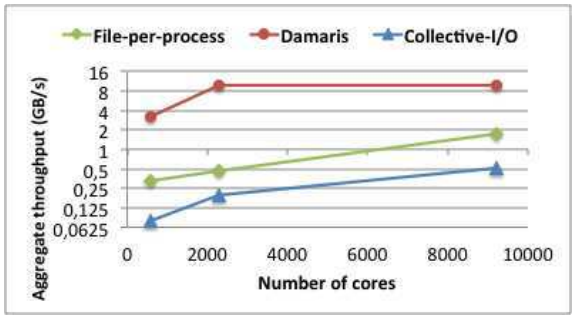


Fig. 6. Average aggregate throughput achieved on Kraken with the different approaches. Damaris shows a 6 times improvement over the file-per-process approach and 15 times over collective-I/O on 9216 cores.

We ran CM1 using 672 cores, writing a total of 15.8 GB uncompressed data (about 24 MB per process) every 20 iterations. With the file-per-process approach, 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.

These experiments show that by replacing write phases by simple copies in shared memory and by leaving the task of performing actual I/O to dedicated cores, Damaris is able to completely hide the I/O jitter from the point of view of the simulation, making the application runtime more predictable.

2) Application's scalability and I/O overlap: CM1 exhibits very good weak scalability and very stable performance when no I/O is performed. Thus as we increase the number of cores, the scalability becomes mainly driven by the I/O phase. To measure the scalability of an approach, we define the scalability factor S of an approach as $S = N * \frac{C_{576}}{T_N}$ where N is the number of cores considered. We take as a baseline the time C_{576} of 50 iterations of CM1 on 576 processes without dedicated core and without any I/O. T_N is the time CM1 takes to perform 50 iterations plus one I/O phase, on N cores. A perfect scalability factor on N cores should be N. The scalability factor on Kraken for the three approaches is given in Figure 4 (a). Figure 4 (b) provides the associated application run time for 50 iterations plus one write phase. As we can see, Damaris shows a nearly perfect scalability where other approaches fail to scale.

Since the scalability of our approach comes from the fact that I/O overlap with computation, we have to show that the dedicated cores have enough time to perform the actual I/O while computation goes on. Figure 5 shows the time used by the dedicated cores to perform the I/O on Kraken and BluePrint, as well as the time they spare for other usage. As the amount of data on each node is equal, the only explanation for the dedicated cores to take more time at larger process count on Kraken is the access contention for the network and the file system. On BluePrint the number of processes here is constant for each experiments, thus the differences in the times to write come from the different amounts of data.

Similar results (not presented because of space constraints) have been achieved on Grid'5000. On all platforms, Damaris shows that it can fully overlap writes with computation and still remain idle 75% to 99% of time. Thus without impacting the application, we could further increase the frequency of outputs or perform inline data analysis, as mentioned in Section III. These use cases will be subject to a future paper.

3) Effective I/O performance: Figure 6 presents the aggregate throughput obtained with the different approaches on Kraken. Note that in the case of Damaris, this throughput is only seen by the dedicated cores. Damaris achieves an aggregate throughput about 6 times higher than the file-per-process approach, and 15 times higher than Collective I/O. This is due to the fact that Damaris avoids process synchronization and access contentions at the level of a node. Also by gathering data into bigger files, it reduces the pressure on metadata servers and issues bigger operations that can be more efficiently handled by storage servers.

The results obtained on 672 cores of Grid'5000 are presented in Table I. The throughput achieved with Damaris is more than 6 times higher than standard approaches. Since compression was enabled on BluePrint, we don't provide the resulting throughputs, as it would depend on the overhead of the compression algorithm and the resulting size of the data.

	Average aggregate throughput
File-per-process	695 MB/s
Collective-I/O	636 MB/s
Damaris	4.32 GB/s

TABLE I
AVERAGE AGGREGATE THROUGHPUT ON GRID'5000, WITH CM1 RUNNING ON 672 CORES.

In conclusion, reducing the number of writers while gathering data into bigger files also has an impact on the throughput that the simulation can achieve. On every platform, Damaris substantially increases throughput, thus making a more efficient use of the file system.

D. Potential use of spare time

In order to leverage spare time in the dedicated cores, we implemented several improvements.

Compression: 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. We achieved an apparent throughput of 4.1 GB/s from the point of view of the dedicated cores. Contrary to enabling compression in the file-per-process approach, the overhead and jitter induced by this compression is completely hidden within the dedicated cores, and do not impact the running simulation. In addition, by aggregating the data into bigger files, we reduce its total entropy, thus increasing the achievable compression ratio.

Data transfer scheduling: Additionally, we implemented in Damaris the capability to schedule data movements. The algorithm is very simple and does not involve any communication between processes: each dedicated core computes an estimation of the computation time of an iteration from a first run of the simulation (about 230 sec on Kraken). This time is then divided into as many slots as dedicated cores. Each dedicated core then waits for its slot before writing. This avoids access contention at the level of the file system. Evaluating this strategy on 2304 cores on Kraken, the aggregate throughput achieves 13.1 GB/s on average, instead of 9.7 GB/s when this algorithm is not used.

These strategies have also been evaluated on 912 cores of Grid'5000. All these results are presented in Figure 7 in terms of write time in the dedicated cores. As we can see, the scheduling strategy reduces the write time in both platforms. Compression however introduces an overhead on Kraken, thus we are facing a tradeoff between reducing the required storage space and the spare time. A potential optimization would be to enable or disable compression at run time depending on the need to reduce write time or storage space.

V. DISCUSSION AND RELATED WORK

In this section, we evaluate the benefits of our approach by computing mathematical bounds of effectiveness. We then position our approach with respect to other related research.

A. Are all cores really needed for computation?

Let us call W_{std} the time spent writing and C_{std} the computation time between with a standard approach, C_{ded} the computation time when the same workload is divided across one less core in each node. We here assume that the I/O time is null or negligible when using the dedicated core

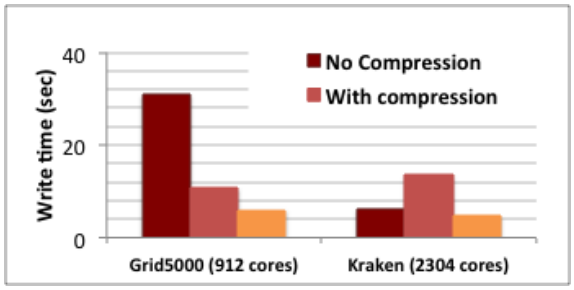


Fig. 7. Write time in the dedicated cores when enabling the compression feature and the scheduling strategy.

(which is experimentally verified) from the point of view of the simulation, and we call W_{ded} the time that the dedicated core spends writing. A theoretical performance benefit of our approach then occurs when

$$W_{std} + C_{std} > max(C_{ded}, W_{ded})$$

Assuming an optimal parallelization of the program across N cores per node, and the worst case for Damaris where $W_{ded} = N * W_{std}$ (even though this has been shown not to be true in Section IV-C3), we show that this inequality is true when the program spends 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. Thus assuming that the application effectively spends 5% of the time writing data, on a machine featuring more than 24 cores per node, it is more efficient to dedicate one core per node to hide the I/O phase. However, many HPC simulations do not exhibit a linear scalability: this enlarges the spectrum of applications that could benefit from Damaris.

In this work, we have used only one dedicated core per node, as it turned out to be an optimal choice. However, Damaris can be deployed on several cores per node. Two different interaction semantics are then available: dedicated cores may have a symmetrical role but are attached to different clients of the node (e.g. they all perform I/O on behalf of different groups of client cores), or they can have an asymmetric behavior, (e.g. one dedicated core receives data from clients and writes it to files, while another one performs visualization or data-analysis). Due to space constraints, deeper insights on semantics is left outside the scope of this paper.

B. Positioning Damaris in the "I/O landscape"

Through its capability of gathering data into larger buffers and files, Damaris can be compared to the ROMIO data aggregation feature [29]. Yet, data aggregation is performed synchronously in ROMIO: all cores that do not perform actual writes in the file system must wait for the aggregator processes to complete their operations. Through space-partitioning, Damaris can perform data aggregation and potential transformations in an asynchronous manner and still use the idle time remaining in the dedicated cores.

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 [24], using asynchronous communications. Yet non-blocking file operation primitives are not part of the current MPI-2 standard. Potential benefits of overlapping

communication and computation are explored in [26]. Our Damaris approach aims to exploit such potential benefits.

Other approaches leverage data-staging and caching mechanisms [23], [15], or forwarding approaches [2] to achieve better I/O performance. Forwarding architectures 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 cross-applications access contention and thus to jitter. However, the trend towards I/O delegate systems underscores the need for new I/O approaches. Our approach relies on dedicated I/O cores at the application level rather than hardware I/O-dedicated or forwarding nodes, with the advantage of letting the user configure its dedicated resources to best fit its needs.

The use of local memory to alleviate the load on the file system is not new. The Scalable Checkpoint/Restart by Moody et al. [20] already makes use of node-level storages to avoid the heavy load caused by periodic global checkpoints. Yet their work does not use dedicated resources or threads to handle or process data, and the checkpoints are not asynchronous.

Some research efforts have focused on reserving 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 [31] 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 [10] RDMA-based transport method, hidden behind the ADIOS interface which allows the system to adapt to any simulation that has an ADIOS I/O backend. However, given the high ratio between the number of nodes used by the simulation and the number of PreDatA nodes, the PreDatA middleware is forced to perform streaming data processing, while our approach using dedicated cores in the simulation nodes permits keeping the data longer in memory or any local storage devices, and to smartly schedule all data operations and movements. Clearly some simulations would benefit from one approach or the other, depending on their needs in terms of memory, I/O throughput and computation performance, but also the two approaches – dedicating cores or dedicating nodes – are complementary and we could imagine a framework that make use of the two ideas.

Space-partitioning at the level of multicore SMP nodes, along with shared memory, has also successfully been used to optimize communications between coupled simulations [14]. In contrast to this work which does not focus on I/O, our goal is precisely to optimize I/O to remove the performance bottleneck usually created by massively concurrent I/O and the resulting jitter.

Closest to our work are the approaches described in [16] and [19]. While the general goals of these approaches are similar (leveraging service-dedicated cores for non-computational tasks), their design is different, and so is the focus and the (much lower) scale of their evaluation. [16] is an effort parallel to ours. It mainly explores the idea of using dedicated cores in conjunction with the use of SSDs to improve the overall I/O throughput. Architecturally, it relies on a FUSE interface, which introduces useless copies through the Kernel and reduces the degree of coupling between cores. Using small benchmarks we noticed that such a FUSE interface is about 10 times slower in transferring data than using shared memory. In [19], active buffers are handled by dedicated processes that can run on any node and interact with cores running the simulation through network. In contrast to both approaches, Damaris makes a much more efficient design choice using the shared intra-node memory, thereby avoiding costly copies and buffering. The approach defended by [16] is demonstrated on a small 32-node cluster (160 cores), where the maximum scale used in [19] is 512 cores on a POWER3 machine, for which the overall improvement achieved for the global run time is marginal. Our experimental analysis is much more extensive and more relevant for today's scales of HPC supercomputers: we demonstrate the excellent scalability of Damaris on a real supercomputer (Kraken, ranked 11th in the Top500 supercomputer list) up to almost 10,000 cores, with the CM1 tornado simulation, one of the target applications of the Blue Waters postpetascale supercomputer project. We demonstrate not only a speedup in I/O throughput by a factor of 15 (never achieved by previous approaches), but we also demonstrate that Damaris totally hides jitter and substantially cuts down the application run time at such high scales (see Figure 4): execution time is cut by 35% compared to the file-per-process approach with 9,216 cores, whereas the largest experiment in [19] (512 cores) with a real-life application only shows a very marginal improvement in execution time. With Damaris, the execution time for CM1 at this scale is even divided by 3.5 compared to approaches based on collective-I/O! Moreover, we further explore how to leverage the spare time of the dedicated cores (e.g. we demonstrate that it can be used to compress data by a factor of 6).

VI. Conclusions

Efficient management of I/O variability (*aka* jitter) on today's Petascale and Post-Petascale HPC infrastructures is a key challenge, as jitter has a huge impact on the ability to sustain high performance at the scale of such machines (hundreds of thousands of cores). Understanding its behavior and proposing efficient solutions to reduce its effects is critical for preparing the advent of Exascale machines and their efficient usage by applications at the full machine scale. This is precisely the challenge addressed by this paper. Given the limited scalability of existing approaches to I/O in terms of I/O throughput and also given their high I/O performance variability, we propose a new approach (called Damaris) which originally leverages dedicated I/O cores on multicore SMP nodes in conjunction with an efficient usage of shared intra-node memory. This solution provides the capability not only to better schedule data movement through asynchronous I/O, but also to leverage the dedicated I/O cores to do extra useful data pre-processing prior to storage or visualization (such as compression or formatting).

Results obtained with one of the challenging target applications of the Blue Waters postpetascale supercomputer (now being delivered at NCSA), clearly demonstrate the benefits of Damaris in experiments with up to 9216 cores performed on the Kraken supercomputer (ranked 11th in the Top500 list). Damaris completely hides I/O jitter and all I/O-related

costs and achieves a throughput 15 times higher than standard existing approaches. Besides, it reduces application execution time by 35% compared to the conventional file-per-process approach. Execution time is divided by 3.5 compared to approaches based on collective-I/O! Moreover, it substantially reduces storage requirements, as the dedicated I/O cores enable overhead-free data compression with up to 600% compression ratio. To the best of our knowledge, no concurrent approach demonstrated such improvements in all these directions at such scales. The high *practical* impact of this promising approach has recently been recognized by application communities expected to benefit from the Blue Waters supercomputer and, for these reasons, Damaris was formally validated to be used by these applications on Blue Waters.

Our future work will focus on 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. We are also investigating other ways to leverage spare time of I/O cores. A very promising direction is to attempt 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 enable efficient inline visualization without blocking the simulation, thereby reducing the pressure on the file systems. Finally, we plan to explore coordination strategies of I/O cores in order to implement distributed I/O scheduling.

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. Some experiments were carried out using the Grid'5000/ALADDIN-G5K experimental testbed (see http://www.grid5000.fr/) and Kraken at NICS (see http://www.nics.tennessee.edu/). The authors also acknowledge the PVFS developers, the HDF5 group, Kraken administrators and the Grid'5000 users, who helped properly configuring the tools and platforms used for this work. We thank Robert Wilhelmson and for his insights on the CM1 application and Dave Semeraro (NCSA, UIUC) for our fruitful discussions on visualization/simulation coupling.

REFERENCES

[1] Damaris open-source implementation: http://damaris.gforge.inria.fr.

[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. *International Parallel and Distributed Processing Symposium*, pages 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*, 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. page 405, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[8] P. Dickens and J. Logan. Towards a high performance implementation of MPI-I/O on the Lustre file system. *On the Move to Meaningful Internet Systems OTM 2008*, 2008.

[9] 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.

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

[11] 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.

[12] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf. Damaris: Leveraging Multicore Parallelism to Mask I/O Jitter. Rapport de recherche RR-7706, INRIA, Nov 2011.

[13] A. Esnard, N. Richart, and O. Coulaud. A steering environment for online parallel visualization of legacy parallel simulations. In *Distributed Simulation and Real-Time Applications, 2006. DS-RT'06. Tenth IEEE International Symposium on*, pages 7–14. IEEE, 2006.

[14] Fan Zhang and Manish Parashar and Ciprian Docan and Scott Klasky and Norbert Podhorszki and Hasan Abbasi. Enabling In-situ Execution of Coupled Scientific Workflow on Multi-core Platform. 2012.

[15] 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.

[16] M. Li, S. Vazhkudai, A. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman. Functional partitioning to optimize end-to-end performance on many-core architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE Computer Society, 2010.

[17] 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.

[18] 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.

[19] X. Ma, J. Lee, and M. Winslett. High-level buffering for hiding periodic output cost in scientific simulations. *IEEE Transactions on Parallel and Distributed Systems*, 17:193–204, 2006.

[20] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. *SC Conference*, pages 1–11, 2010.

[21] Mpich2. http://www.mcs.anl.gov/research/projects/mpich2/.

[22] NCSA. BlueWaters project, http://www.ncsa.illinois.edu/BlueWaters/.

[23] 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*, 2008.

[24] 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.

[25] 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. page 58, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[26] 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, 2006.

[27] 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*. Citeseer, 2002.

[28] D. Skinner and W. Kramer. Understanding the causes of performance variability in HPC workloads. In *IEEE Workload Characterization Symposium*, pages 137–149. IEEE Computer Society, 2005.

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

[30] 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.

[31] 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*, 2010.