On the benefit of dedicating cores to mask I/O jitter in HPC simulations

Matthieu Dorier

On the benefit of dedicating cores to mask I/O jitter in HPC simulations

Master Thesis

Matthieu Dorier
matthieu.dorier@irisa.fr

Supervisors: **Gabriel Antoniu, Luc Bougé**
{Gabriel.Antoniu,Luc.Bouge}@irisa.fr

ENS Cachan, Brittany extension, IRISA, KerData Team

June 2, 2011

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. We here utilize the IOR benchmark to explore the influence of user-configurable parameters of the I/O stack on performance 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.

Keywords: Exascale Computing, Multicore Architectures, I/O, Performance Variability, Dedicated Cores

Contents

1 Introduction

As HPC[1] 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 (Input/Output). This variability, hard to characterize, comes from the many levels that go from the computation nodes to dedicated I/O nodes running the parallel file system. As storing data is the only way to allow post-processing and retrieving of scientific results, dealing with this performance variability is more of an issue.

I/O in large-scale simulations A typical I/O approach in large-scale simulations consists of alternating computation phases and I/O phases. Often due to explicit synchronization 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 small scale (1000 processes) 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 [42]. 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.

I/O performance evaluations 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 parallel file system to the simulation-side I/O library, few efforts have been made in addressing I/O jitter. Yet it has been shown [42] 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 parallel file system [13] causes a bottleneck when each process writes its own file (file-per-process approach), a problem that PVFS [6] or GPFS [35] do not exhibit. In contrast, byte-range locking in GPFS or equivalent mechanisms in Lustre cause lock contentions when writing to shared files, while leaving the resolution of access semantics to the upper MPI-IO (the IO part of the Message Passing Interface [15] standard) layer in PVFS avoids this kind of overhead but forces more optimizations in MPI-IO [21]. To address this issue, elaborate algorithms at the MPI-IO level are used in order to maintain a high throughput [31].

[1] High Performance Computing

Contribution In this report, 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 [36]. These experiments are carried out on the French Grid'5000 testbed with PVFS as the underlying parallel file system. Studying the variability of a system poses an immediate problem of interpretability of results. While pure performance studies can be done by running the application once or a few times in order to provide confidence intervals for a performance metrics, variability is by definition associated to the characterization of not only one, but a consistently large set of experiments. As we will see in this report, using the statistical variance as a metric for studying a system's variability is not relevant. Thus we propose a new graphical method to study this variability. This method allow rapid insights of the influence of a parameter over global performance and performance variability. As an illustration, we show the impact of some user-controllable parameters of the considered I/O system on this variability.

Based on the observation that a first level of contention occurs when all the cores of a multicore SMP[2] 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 or a few cores 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) [4] (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]Symmetric Multi-Processing

2 Background and related work

2.1 On large-scale simulations

HPC is at the interface between computer science and many other fields such as biology, engineering, chemistry, astronomy,... Supercomputers are used by scientists to evaluate mathematical models of our real world through simulations. The higher the performance of the HPC platform used, the more accurate results they can retrieve from large-scale simulations. Thus machine vendors together with computer science researchers are pushing computation power always farther, now achieving about 4.5 petaflops ($4.5E^{15}$ floating point operations per second) with the Tianhe-1A supercomputer at the National Supercomputing Center in Tianjin (China) [40], expecting more than 11 petaflops with IBM's Blue Waters, and targeting exascale machines by 2018 [10].

Over the past several years, chip manufacturers have increasingly focused on multicore architectures, as the increasing clock frequencies for individual processors have leveled off, primarily due to substantial increases in power consumption [17]. These more and more complex systems present new challenges to programmers, as approaches which work efficiently on simpler architectures often do not work well on these new systems.

Large-scale simulations usually make a symmetrical use of resources [16]; HPC platforms are providing an homogeneous set of nodes that are connected through an efficient network topology, such as 2D or 3D torus, to match the most common data partitioning patterns. The communication between processes is handled using the message-passing interface (MPI) [3] implementation provided on the platform. MPI is a standard that defines the semantics of a set of communication primitives such as *send* or *receive*, either synchronous or asynchronous, as well as more complex operations such as efficient broadcasting of data, or distributed processing, allowing remote processes to communicate and exchange data. Yet the scalability of such an approach to millions of processes is uncertain [18, 17] and finding new designs for distributed applications is a major challenge [16] if one wants to leverage all the capabilities of the underlying machine.

In this context, a key challenge consists in dealing with the high performance variability across individual components, which becomes more of an issue, as it can be very difficult to track the origin of performance weaknesses and bottlenecks. Moreover, performance variability lead to unpredictable run time. In particular the jitter of I/O systems is a major concern, as it causes processes to waste time idle in write phases, waiting for the slowest process to complete its output. As the performance of storage systems fade behind an ever increasing computation power, global performance of applications starts indeed being driven by the performance of I/O systems. Thus any I/O jitter can cause a drastic loss of overall efficiency.

2.2 Understanding I/O variability

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 [37], 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 [42]. 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 [23], 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 [23] to allow a more efficient and less variable access to the file system.

2.3 The I/O stack: approaches to I/O management in HPC

The notion of I/O stack designates a set of layers between the application level that writes the data and the file system level that stores it. The typical I/O stack of HPC applications is presented in Figure 1. A scientific data format such as HDF5 [27] or NetCDF [41] is commonly used for writing the data together with some metadata information, in order for these data to be understandable by other applications potentially running on a different platform. These scientific formats are usually well tuned to interact with the underlying MPI-IO layer, which provides a unified way to send data to a parallel file system. Tuning the I/O stack such that it exhibits the smallest possible jitter while sustaining high

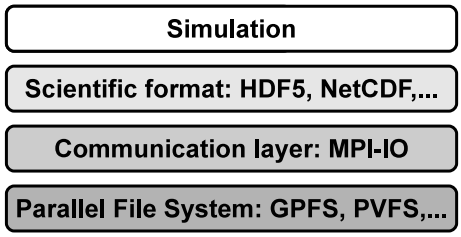

Figure 1: The I/O stack.

throughput requires a deep understanding of the I/O system's architecture and the possible ways to write data from large-scale simulations. In our study, we chose to focus on two of these layers, which will be further presented in this section : the parallel file system, and the MPI-IO layer. Thereby this section introduces these two layers.

2.3.1 Parallel file systems

The design of parallel file systems for HPC infrastructures is based on the assumption that the computing area (set of computing processors) is separated from the storage area. This storage area, made of storage devices accessed through I/O servers, is shared by all the users of the supercomputer, and can even be shared between several supercomputers at the same time making it a crucial component to optimize. Thus the file system has to provide efficient support for concurrent accesses issued by multiple HPC applications running at the same time, and it has to be able to cope with the ever increasing number of cores used by each of these applications. Most parallel filesystems, including PVFS [6] or GPFS [35], have the same kind of architecture featuring metadata servers that are responsible for keeping the informations about files (name, group, authorizations, placement of chunks among the storage devices...), and multiple I/O servers responsible for handling read and write requests. Yet some file systems such as Lustre [13] have only one metadata server, which can be the origin of a bottleneck when a lot of processes perform metadata-related operations (creating files, listing a directory,...). High degree of parallelism is achieved by stripping big files in small chunks (several MB) that are distributed across I/O servers (thus across disks) usually in a round-robin manner [9]. The size of the stripes and their distribution across I/O servers can usually be set by the end-user through extended directory operations that leverage tools introduced by SGI's XFS [43].

Such a parallel file system has to expose a precise consistency semantics that defines the result of conflicting operations such as concurrent writes to the same file or interleaved reads and writes from different processes. The most common such a semantics is the POSIX consistency semantics, which states for example that a process reading a particular file will see either all or none of the modifications produced by a write performed by another process. This is a strong consistency model that requires either process synchronization or file locking when accessing data and metadata, as it has been initially designed for local file systems. On GPFS [35] this consistency is provided through a distributed

byte-range locking algorithm, which issues access authorization to processes for sets of intervals in a file. Even though this distributed algorithm performs better than centralized locking while allowing non-conflicting writes to be issued concurrently, it introduces more overhead in small accesses and still locks parts of files, which should be avoided. The solution proposed by PVFS however consists in providing a weaker semantics, without any locking mechanism. As a consequence if a process issues a write of some data and that data must be split in several chunks, it might append that another process reading meanwhile the same file see some chunks modified and some other not. Using this semantics requires processes to be synchronized at another level in order to remain consistent. This is done by implementing the semantics inside the upper-layer, at the MPI-IO level.

Non-contiguous I/O is a major problem in this type of file system. Yet it is a very common access pattern in scientific simulations and must be handled properly. In PVFS, these non-contiguous accesses are performed through the *list I/O* [9] interface, an extension of the PVFS interface that aims at handling atomically a list of operations. Like the access semantics, this kind of extension is leveraged in MPI-IO implementations.

As the number of concurrent processes increases, the current design of parallel file systems does not fit the requirements of the HPC workloads, and leads to a jitter. As an example we can consider the case of the next IBM's supercomputer Blue Waters, which will provide 100 I/O nodes to handle the load of more than 300.000 computing cores. The presence of locking mechanisms immediately lead to the serialization of some requests, and thus a non-optimal use of bandwidth. The first idea to better handle the load of large-scale simulations is to leverage their MPI-IO interface and to make all processes cooperate in writing only one big file. Doing so, processes can agree on the operations to perform, exchange data and avoid the lock mechanisms of the file system. This can be done through the use of collective I/O [11], a general technique leveraging a high degree of interaction between MPI-IO and the file system, which will be treated in the next section.

2.3.2 I/O strategies using MPI-IO

Two main approaches are typically used for performing I/O in large-scale HPC simulations: either we consider that, as a set of processes, the application should make each process writes its own file, or as a single program it should write a single file. Since MPI-2 [15], the message passing interface standard includes an I/O part called MPI-IO, aiming at unifying an improving the efficiency of file accesses. As a response to these two I/O strategies, MPI-IO thus provides functions for both *independant* and *collective* I/O.

The *file-per-process* approach, or *independant I/O*, consists in having each process write in a separate, relatively small file. This approach is popular first because it is easy to implement, then because when writing data for checkpointing purpose, each process writes its own file from which it will be able to eventually restart without carrying about other processes. Whereas the file-per-process approach avoids synchronization between processes, parallel file systems are not well suited for this type of load at a large scale, in particular because of the overhead introduced by metadata accesses.

Special optimizations are then necessary [5] 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: it requires additional post-processing phases that gather the multiple datasets into single files, as the number of processes used for visualizing and analyzing the results usually do not match the number of processes used by the simulation that has generated the data.

Using *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 [31]. Algorithms termed as "two-phase I/O" [11, 38] enable efficient collective I/O implementations by aggregating requests and by adapting the write pattern to the file layout across multiple data servers [9]. Processes first exchange data in order to issue bigger and contiguous writes, then each process sends the data it is responsible for to the file system. Scientific data formats such as HDF5 and NetCDF provide a parallel interface: pHDF5 [7] and pNetCDF [22] that leverage collective I/O. When using one of these formats, collective I/O avoids metadata redundancy (only one process effectively writes the additional information), 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 parallel interface of scientific data formats to perform data compression, which is a desirable feature considering the very large amounts of data generated.

It becomes easier today to switch between these approaches when a scientific format is used on top of MPI; if the data is split in a regular way across processes, going from HDF5 to pHDF5 is a matter of adding a couple of lines of code that adapt the local representation of data to the global layout. It can be even simply a matter of changing the content of an XML file when using an adaptable layer such as ADIOS [24]. ADIOS is indeed a first step in the path to new I/O approaches. It proposes to move all the I/O configuration into an external file that can be modified from a run to another without recompiling the code, thus accelerating the tuning of the I/O part of large-scale simulations. 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. Thus, new I/O approaches have to be found in order to improve the efficiency and reduce the jitter.

2.4 Motivation: toward new I/O approaches, requirements

In the previous sections we have seen the limitation of the current I/O stack, which has been designed for the most common I/O patterns: file-per-process or collective-I/O. When scaling to hundreds of thousands of processes, these I/O strategies together with the parallel file systems do not scale, suffering from a lake of performance and an increasing variability. It is usually accepted that a simulation takes a maximum of 5% of its time in performing I/O or I/O-related operations. This threshold cannot be sustained and new I/O approaches have to be found. The first attempts in optimizing the I/O stack can be done when setting up the HPC platform, by providing more resources to I/O than just a couple of data servers. Following this direction, some approaches leverage data-staging and caching mechanisms [29, 19], 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, the fact that these resources are shared by all users still leads to multi-application access contention. Thus, our work focuses on approaches that can be tightly coupled with the application, and over which end-users can have a complete control. Several other works can be considered to draw the requirements of such an approach.

Overlapping I/O with computation - In order to avoid I/O bursts with the approaches previously mentioned, other efforts are focused on overlapping computation with I/O. This reduces the impact of I/O latency on overall performance. Overlap techniques can be implemented directly within simulations [30], 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 [34], 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.

Dedicating resources to I/O - Some research efforts have focused on using the resources in an heterogeneous manner, by reserving some computation nodes 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 [44] 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 [12] RDMA[3]-based transport method. However, given the high ratio between the number of nodes used by the simulation and the number of nodes

[3]Remote Direct Memory Access

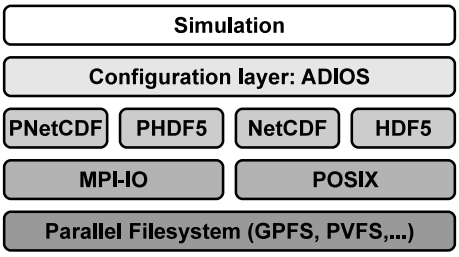

Figure 2: An I/O stack featuring an intermediate configuration layer.

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.

Making I/O configuration transparent - The differences between supercomputing platforms together with the configuration of particular experiments force HPC users to implement several I/O back-ends for their simulations, and to spend time tuning these back-ends in order to achieve optimal performance. When taking a look at a simulation code, we usually observe several transport layers, enabled or disabled and configured at compile time with `#ifdef` and `#define` precompilation statements. Finding the best configuration, besides the fact that it involves code recompilation when changing the parameters, is time and resource consuming. We have already mentioned ADIOS, as a solution that aims at unifying transport methods behind a common API proposing a set of functions as simple as POSIX I/O functions. Thanks to this library, after having written the I/O part of their simulation using ADIOS calls, users are free to choose the output methods, from raw POSIX to richer formats like pHDF5 or pNetCDF, as well as custom backends, using a simple XML configuration file. Figure 2 shows the new I/O stack enriched by this configuration layer. Even though ADIOS allows the users to reconfigure their I/O backends without code recompilation, we can expect more from this kind of configuration layer. Our approach follows this idea of externally-configurable backend, but allows I/O reconfiguration at run time and even automatic I/O reconfiguration depending on the contents of the data being written (the backend can wait for relevant data to appear before starting to write something, or some data can be thrown away if they do not match some predefined criteria of relevance).

Gathering data and allowing data compression - As we said previously, one of the disadvantages of the file-per-process approach is the fact that data are hardly readable afterward for analysis purpose, while on the other hand collective I/O imposes synchronization across processes and does not allow data compression. Thus, new approaches should provide simple ways to gather and compress data without synchronization overhead. In [14] the authors propose an approach where a large number

of small files is actually stored in one big file at the file system level, thus avoiding metadata contention. We think that a better approach should allow partially or fully distributed data compression and consistent aggregation of datasets. Our approach leverages multicore SMP nodes and allows to reduce the number of files to one file per node instead of one per core.

In this section we have presented the I/O systems and the usual approaches to I/O management in HPC simulations. We have seen the limitations of these approaches and the main requirements for new ones. The next section will present in more details some origins of the I/O jitter when using the common I/O strategies. As the number of layers and the number of parameters to configure each of these layers is huge, we will only focus on some of these parameters. Nevertheless, our methodology for studying the variability is generic and can even be applied to jitter analysis in a wider context than just I/O.

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 file-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 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 and tools

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

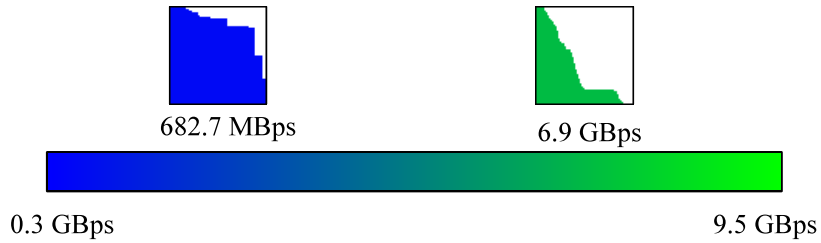

Figure 3: Visual representation of runs variability. The left run has a smaller throughput but less overall idle time than the right one.

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 3. 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: a linear shape may indicate that requests are serialized and considered one at a time, like in a waiting queue. Staircase patterns indicate that groups of requests can be treated simultaneously. We sometimes noticed that the slowdown of one single process caused a huge waste of time. 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. Grid'5000 allows users to deploy their environment and to have a complete control over a set of reserved nodes. Contrary to a real HPC platform, where end-users can only submit jobs, we have a root access on all the reserved resources during the time of this reservation, which in particular allows us to deploy and configure our own parallel file system and our own implementation of MPI. In the following experiments, 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-2 [26], with ROMIO [38] as implementation of MPI-IO, compiled against the PVFS library.

IOR [36] 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 4 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 outputs 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 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.

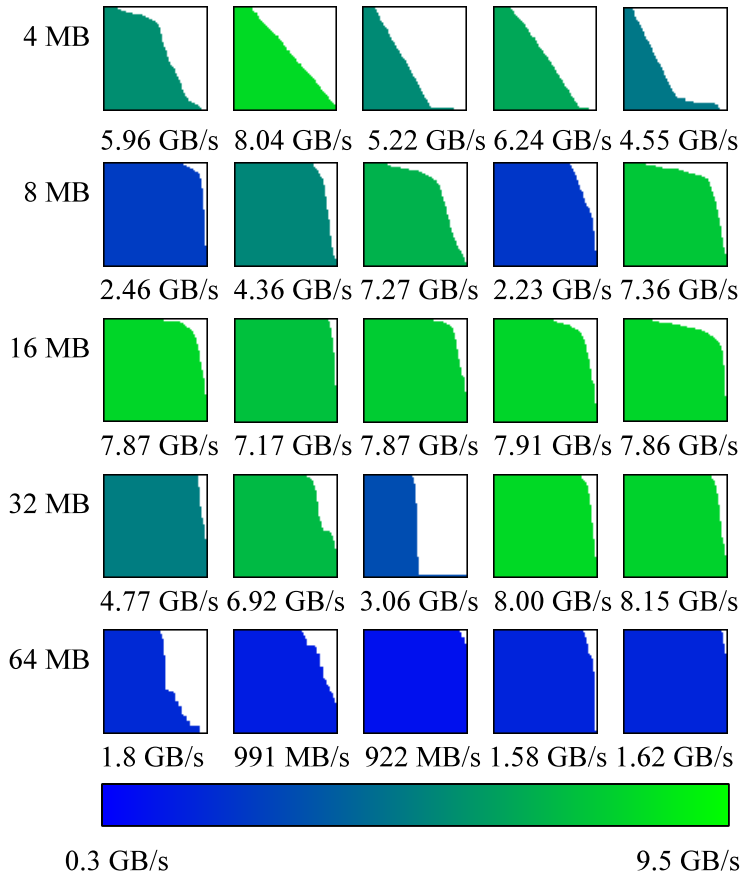

Figure 4: Influence of the data size on I/O performance variability.

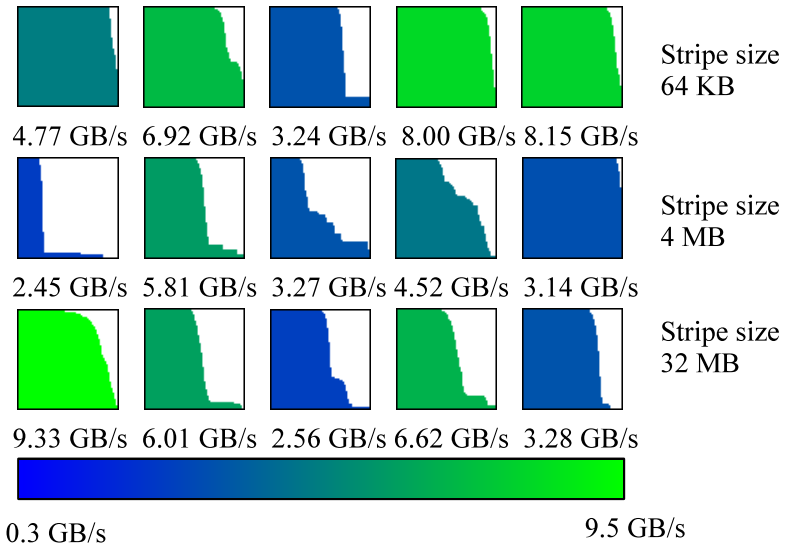

Figure 5: Throughput variability for three experiments with different stripe size.

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 GB/s 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 5, 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-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

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.

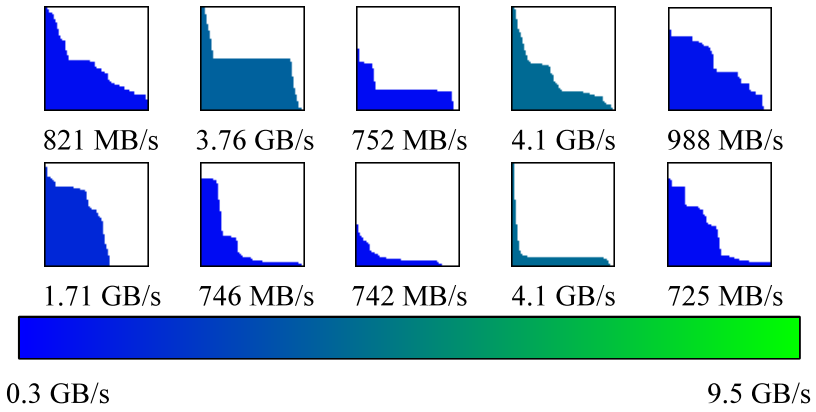

Figure 6: Throughput variability with the file-per-process approach.

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 6 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/s. Thus Figure 7 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 and peak throughput achieved, highly depends on the data size and on the I/O strategy. 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

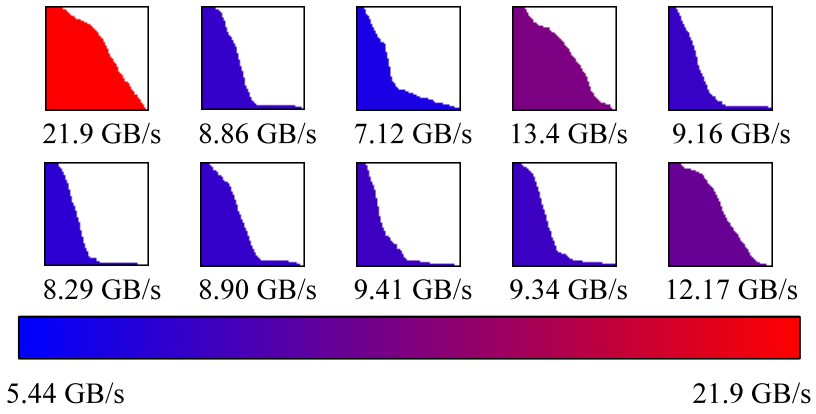

Figure 7: Throughput variability with the file-per-process approach, writing 32 MB per process with a 32 MB stripe size.

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 Damaris: 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 strategy 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 together with all the opportunities it offers to rethink the standard computing paradigms.

4.1 Main 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. Thus the normal symmetric computational pattern, presented in Figure 8, is changed into the one presented in Figure 9.

Contrary to the software that we are commonly using on our personal machines, which depend on interactions with a user through a graphical interface, a scientific simulation is somehow a black box that performs computation and output files following an extremely predictable behavior. Yet to our knowledge, no attempt have been made so far to leverage this *a priori* predictability in order to offer a better quality of service in parallel file systems or other I/O systems. Our buffering system running on dedicated cores is configured to match the expected I/O pattern, which requires the user to describe it in a configuration file before starting the simulation. Based on this knowledge, the dedicated core 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. Performing a copy of data in shared memory is generally several orders of magnitude faster than writing that data into a file, thus the write time in the simulation becomes negligible, and so does the I/O jitter.

4.2 Architecture of the Damaris middleware

The architecture of Damaris is shown in Figure 10, 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 [4].

External configuration - 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 model of the data generated by the application. It follows an idea introduced by the ADIOS data format in [24], 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. Moreover, the simulation is also initialized with this configuration file, allowing the client-side library to efficiently prepare intermediate buffers.

Intra-node 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. 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

[4] By the term "core" we also include hardware threads that allow to have a larger number of virtual cores.

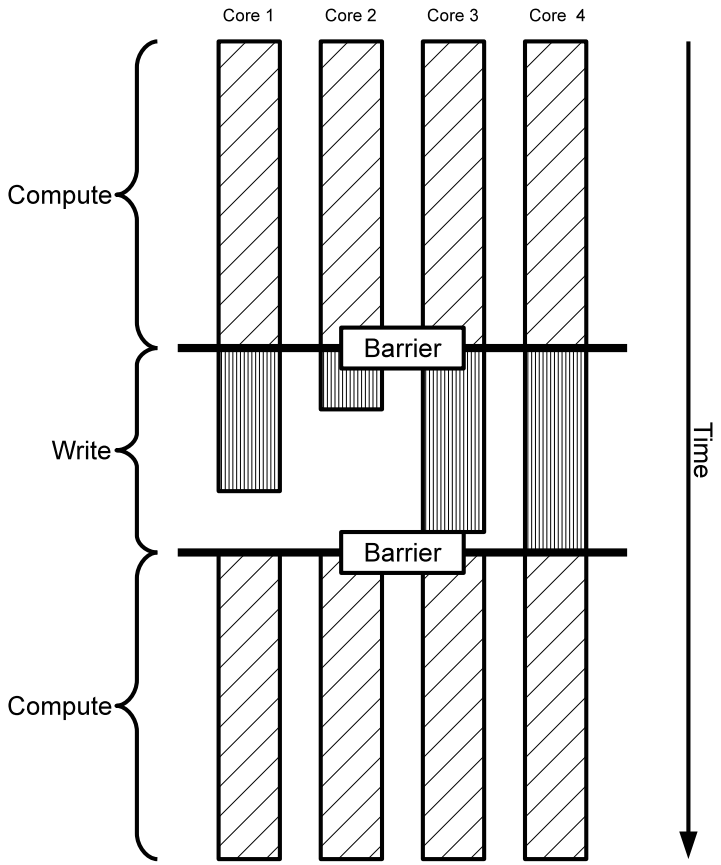

Figure 8: Temporal behavior using a standard I/O approach: because of explicit barriers or communication phases, all the processes have to wait for the slowest before going to the next computation step, causing a huge waste of time.

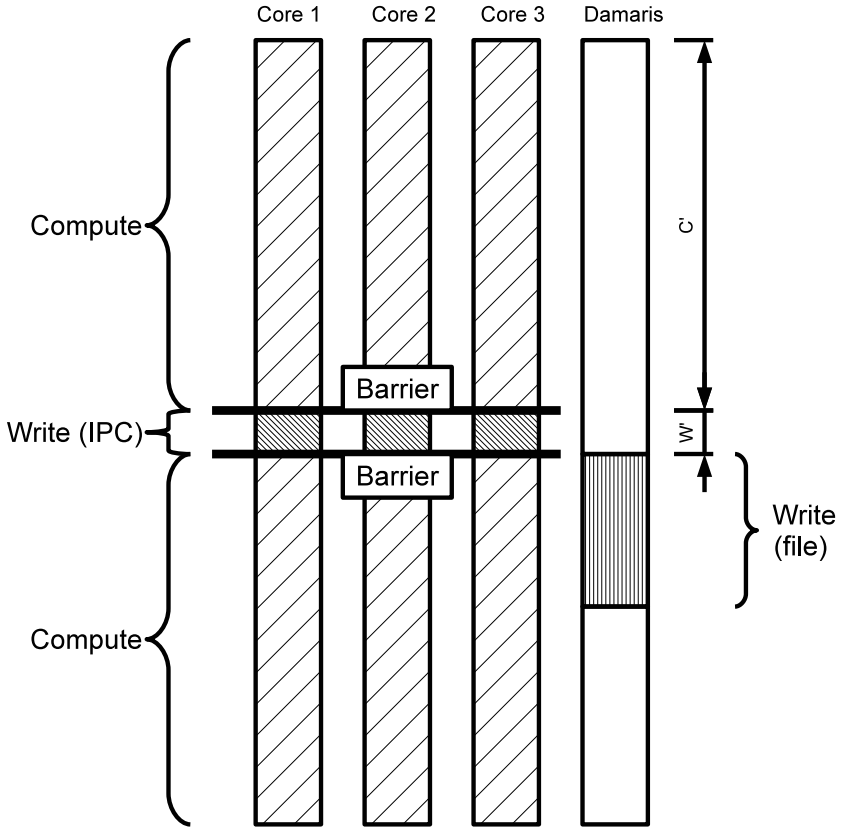

Figure 9: Temporal behavior using Damaris: each core writes in shared memory using interprocess communications (IPC), then the I/O core takes care of the effective write asynchronously.

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.

Metadata management - All datasets written by the clients are uniquely characterized by a tuple $\langle$*name,iteration,source,layout*$\rangle$. *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 bits 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 system 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). This metadata index can be used to navigate through the stored variables, filter them by name, iteration or other conditions. We plan to extend this metadata structure into a distributed metadata service shared by all the dedicated cores, thus allowing an efficient navigation through the complete output of the simulation.

Plugin system and events - 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.

4.3 Deeper insights on particular 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

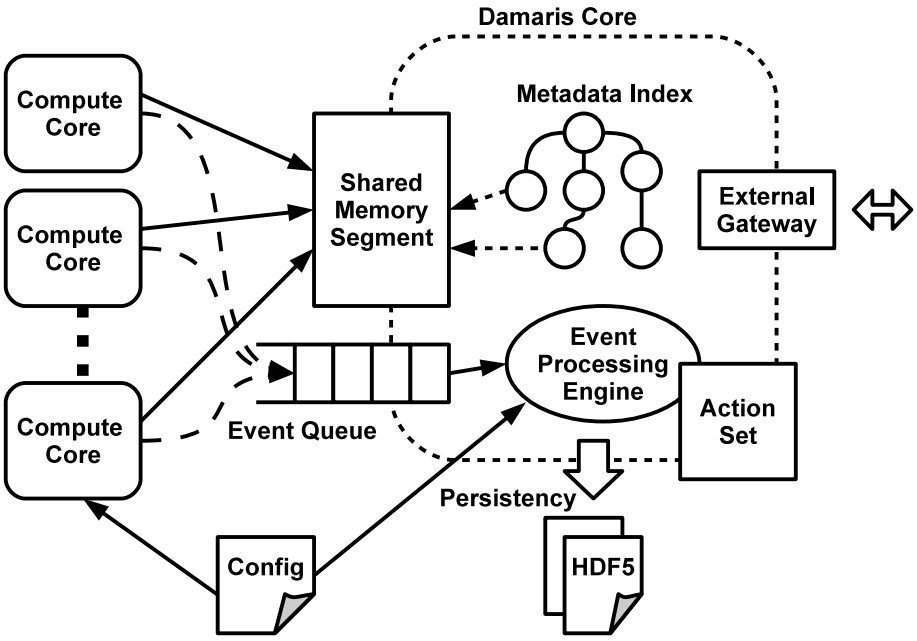

Figure 10: Design of the Damaris approach.

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 and integration in simulations

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 [39] 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. Considering the event-queue semantics, the notification of disconnection arrives after all events potentially sent before.

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, where the number of particles handled by a process can vary, for example). Finally it is possible to embed the server within the simulation so it can be deployed and undeployed with the simulation.

Listing 1 provides 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, in Listing 2, 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.

Listing 1: ***Fortran "hello world" using Damaris***

```
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
```

Listing 2: ***my_config.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 HDF5 callback routines. Since HDF5 calls are hidden in the dedicated cores, the integration in existing simulations is extremely simple. To illustrate this simplicity, a comparison is provided in Annexe A between two programs written in C: the first one uses HDF5 to open a file, open related structures and write some data, while the second one uses Damaris to perform the same operations. As an example, it took less than an hour to create an I/O part for the CM1 atmospheric simulation to use Damaris. The following section presents the results that have been achieved with this particular simulation.

5 Experimental results with the CM1 atmospheric model

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

5.1 The CM1 application

CM1 [4] 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 across processes, 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 1.9 GHz 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. The interconnect between nodes and between data servers is 1 Gb Ethernet. 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 24x40x300 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.

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 11. 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 12. This pattern is recurrent and representative of the global behavior.

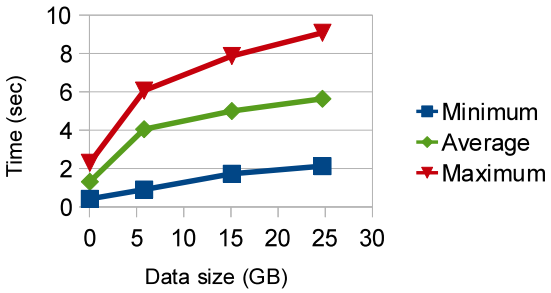

Figure 11: Maximum, minimum an average write time on BluePrint with different total output sizes, using the file-per-process approach, compression enabled.

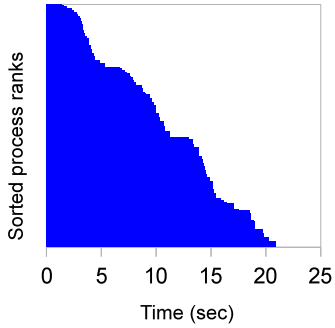

Figure 12: Variability pattern from an output of CM1 using the file-per-process approach on Grid'5000.

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 potential burden of a heavy

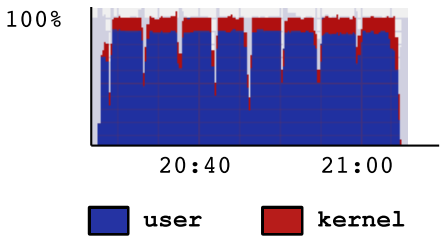

Figure 13: CPU usage measured on a node running CM1 with a collective I/O mechanism.

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 13 shows CPU usage on one of the nodes used by CM1 during the execution. This CPU usage measurement is retrieved from the Ganglia monitoring system provided by the Grid'5000 platform. 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.

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

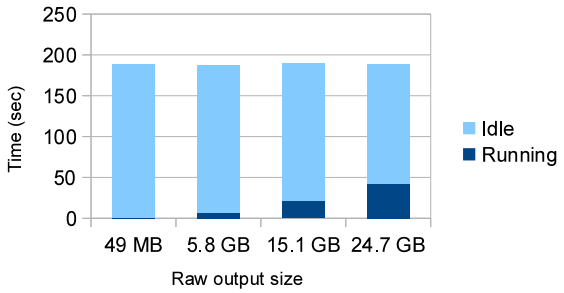

Figure 14: Write time and idle time in the I/O cores on BluePrint. Data size is the raw uncompressed data output from all dedicated cores together

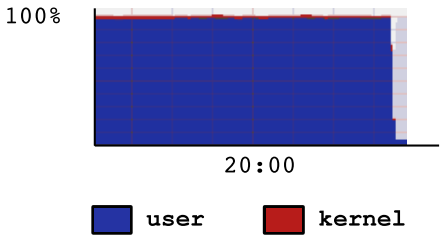

Figure 15: CPU usage measured on a node running CM1 with Damaris.

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 14 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 15 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.

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 [25].

As we have seen, Damaris presents very good results in hiding the I/O jitter. Moreover, it spares time to perform data-processing or other tasks. Damaris can serve as a basis for several research directions that we present in the next section.

6 Prospects: looking toward exascale with Damaris

In this section, we study the new opportunity offered by Damaris now that a good basis for dedicating cores has been developed and tested.

6.1 Coupling simulation with visualization through Damaris

A first direction that can be explored is the opportunity provided by Damaris in efficiently coupling simulations with visualization tools. Current ways of retrieving scientific insights from large simulation's outputs are based on offline methods. After the simulation terminates, the output is processed from a distributed visualization pipeline [8, 33]. Using dedicated cores, we can potentially connect visualization tools directly to large-scale simulations and perform visualization while the simulation is running (inline visualization). This provides an efficient way to better interact with the running simulation and to control it without introducing any jitter.

We have previously explained that large-scale simulation have a predictable time-pattern and data layout. We can go deeper in this direction and make a parallel between scientific simulations and websites. Websites usually separate the presentation of contents from the contents itself. The same content can be displayed in different manner as soon as its layout is properly described. In large-scale simulations, visualization tools can be seen as web browsers that navigate throughout the datasets and display them. The dedicated cores would act as a distributed database for contents while visualization tools could also leverage Damaris' configuration files to produce different views of this content. With a direct interaction between the visualization engine and the dedicated cores, Damaris could prepare the data in an efficient way for the expected views.

6.2 From concurrency to cooperation and code-coupling

In Section 5.4.2 we have implemented a scheduling algorithm to avoid access contention in the file system. This very simple technique was possible because no user-defined event or data-processing plugins were used in our case. In the presence of more elaborated behaviors, the dedicated cores used by Damaris should be able to cooperate in order to

make an efficient use of resources and set priorities to actions. This implies finding efficient distributed scheduling strategies.

Considering several simulations running on the same HPC platform, we could imagine extensions of this distributed scheduling in order for the different applications to communicate an agree on efficient resource sharing. This could avoid cross-applications interference effects.

Finally, it is also common for several large-scale simulation to be coupled. As an example, the Community Climate System Model (CCSM) features 4 different codes that respectively simulate atmosphere, ocean, ice and vegetation [32]. The objectives of code-coupling consists in sharing data between several simulations in order to simulate more complex systems. As for visualization, a same data can have a different layout in the different code, thus different views of the same data could be provided by dedicated cores that would efficiently adapt these views together with the underlying contents.

6.3 Dedicating more than one core

One can wonder if it could be interesting to dedicate more than one core to I/O, and if this is possible with the current implementation of Damaris. Indeed, we have seen in the experiments that leaving cores out can lead to better performance. The answer is yes. Yet in the current version of Damaris, each dedicated core will feature its own shared buffer, and a client will have to choose one single dedicated core to communicate with. Moreover, several configuration files should be provided in order for the dedicated cores not to interfere. Since no communication is currently provided between dedicated cores, there would be no way to gather datasets without implementing some complex user plugins. A 8 cores SMP node with 2 dedicated I/O cores would thus be seen as 2 separate 4 cores SMP nodes.

We plan to investigate how the number of cores influence the scalability of the simulation, and how to build a multithreaded version of Damaris that could be used on several dedicated cores. The main issue of this research direction is the definition a consistent semantics, because a multithreaded version of Damaris would be likely to process user-defined events in parallel and would not be consistent with the semantics we have provided here.

As a conclusion of this section, Damaris provides a very good basis for starting research in several directions targeting new challenges at Exascale.

7 Conclusions

7.1 Leave a core, go faster

We have seen throughout this paper that standard I/O approaches introduce a jitter in large-scale simulations. This jitter was presented in the context of small 1000-way scales experiments on two platforms, and the effect of this jitter is becoming a real problem at

larger scales. Current experiments conducted on the Kraken [20] machine at NICS[5] with Lustre as underlying parallel file system show that classical I/O approaches do not scale anymore, leading to extremely poor performance and an overhead much higher than the commonly accepted 5% of wallclock time. Thus new I/O approaches are needed.

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.

7.2 Future works

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, as exposed in Section 6. 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 [28] 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.

[5]National Institute for Computational Sciences

Acknowledgments

I would like to acknowledge all the members of the KerData team: Alexandra Carpen-Amarie, Alexandru Costan, Diana Moise, Gabriel Antoniu, Houssem Chihoub, Luc Bougé, Radu Tudoran, Viet-Trung Tran, together with Bunjamin Memishi and Izabella Moise, with a particular regard to Gabriel who pushed me in submitting a poster (which has been accepted) to the ICS'11 conference and to submit a paper for the extremely selective SC'11 conference.

I also acknowledge Franck Cappello, and Marc Snir (co-directors of the Joint INRIA/UIUC Laboratory for Petascale Computing) for co-authoring this last paper and providing very good insights on this work, and Leigh Orf for co-authoring the paper, providing insights and acting as an editor by correcting it.

Finally I also 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, the PVFS developers, the HDF5 group and the Grid'5000 users, who helped properly configuring the tools and platforms used for this work.

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].

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] Y. Aoyama and J. Nakano. *Rs/6000 sp: Practical MPI programming*. Citeseer, 1999.

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

[5] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. Small-file access in parallel file systems. *Parallel and Distributed Processing Symposium, International*, 0:1–11, 2009.

[6] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev 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.

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

[8] H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, et al. Extreme Scaling of Production Visualization Software on Diverse Architectures. *IEEE Computer Graphics and Applications*, pages 22–31, 2010.

[9] Avery Ching, Alok Choudhary, Wei keng Liao, Rob Ross, and William Gropp. Noncontiguous i/o through pvfs. volume 0, page 405, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[10] Computerworld. Scientists IT Community Await Exascale Computers, http://www.computerworld.com/s/article/345800/, viewed on December 2010.

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

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

[13] Stephanie Donovan, Gerrit Huizenga, Andrew J. Hutton, C. Craig Ross, Martin K. Petersen, and Philip Schwan. Lustre: Building a file system for 1000-node clusters, 2003.

[14] Wolfgang Frings, Felix Wolf, and Ventsislav Petkov. Scalable massively parallel I/O to task-local files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 17:1–17:11, New York, NY, USA, 2009. ACM.

[15] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir. MPI-2: Extending the message-passing interface. In *Euro-Par'96 Parallel Processing*, pages 128–135. Springer, 1996.

[16] A. Geist and R. Lucas. Major computer science challenges at exascale. *International Journal of High Performance Computing Applications*, 23(4):427, 2009.

[17] R. Gioiosa. Towards sustainable exascale computing. In *VLSI System on Chip Conference (VLSI-SoC), 2010 18th IEEE/IFIP*, pages 270–275. IEEE.

[18] A. Hoisie and V. Getov. Extreme-scale computing–where'just more of the same'does not work. *Computer*, 42(11):24–26, 2009.

[19] Florin Isaila, Javier Garcia Blas, Jesus Carretero, Robert Latham, and Robert Ross. Design and evaluation of multiple level data staging for Blue Gene systems. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2010.

[20] Kraken Cray supercomputer. http://www.nics.tennessee.edu/computing-resources/kraken.

[21] R. Latham, R. Ross, and R. Thakur. The impact of file systems on MPI-IO scalability. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 146–198, 2004.

[22] 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.

[23] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew 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.

[24] Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen 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.

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

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

[27] NCSA. Hierarchical Data Format HDF5, http://www.hdfgroup.org/HDF5/, viewed in December 2010.

[28] Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise, and Alexandra Carpen-Amarie. BlobSeer: Next Generation Data Management for Large Scale Infrastructures. *Journal of Parallel and Distributed Computing*, 71(2):168–184, February 2011.

[29] A. Nisar, Wei 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.

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

[31] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia, and Alice 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.

[32] Nicolas Richart. *Conception et mise en œvre d'une plate-forme de pilotage de simulations numériques parallèles et distribuées*. PhD thesis, Université de Bordeaux I, 2010.

[33] O. Rubel et al. High performance multivariate visual data exploration for extremely large data. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE Press, 2008.

[34] Jose Carlos Sancho, Kevin J. Barker, Darren J. Kerbyson, and Kei 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.

[35] 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.

[36] 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.

[37] 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.

[38] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. *Frontiers of Massively Parallel Processing, Symposium on the*, 0:182, 1999.

[39] The Boost C++ Library. http://www.boost.org/.

[40] Top500 supercomputer. http://www.top500.org, viewed in May 2011.

[41] UCAR. NetCDF format, http://www.unidata.ucar.edu/software/netcdf/, viewed in May 2011.

[42] A. Uselton, M. Howison, N.J. Wright, D. Skinner, N. Keen, J. Shalf, K.L. 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.

[43] XFS. http://xfs.org/, viewed in May 2011.

[44] Fang Zheng, H. Abbasi, C. Docan, J. Lofstead, Qing 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.

A Damaris integration in existing simulations

Listing 3 presents the way a program can write a simple 2D array using HDF5. It first creates the file, then creates a space by providing the dimensions and extents, creates a dataset hierarchy, and finally writes the data. Then all the associated identifiers are closed. As a comparison, Listing 4 presents the same write operation through Damaris, which consists of only one single function call. All required information such as the data dimensions and extents are provided in the configuration file (Listing 5). Setting the variable persistence to "immediate" indicates that the data must be stored upon reception, thus Damaris will write it using HDF5 as soon at it receives it. The persistence layer of Damaris uses HDF5 and look like the Listing 3, but it is made generic and is hidden to the user.

Listing 3: ***Writing a dataset using HDF5***

```
#include "hdf5.h"
#define FILE "dset.h5"

int main() {

   hid_t file_id, dataset_id, dataspace_id; /* identifiers */
   herr_t status;
   int i, j, dset_data[4][6];
   hsize_t dims[2];

   /* Initialize the dataset. */
   for (i = 0; i < 4; i++)
      for (j = 0; j < 6; j++)
         dset_data[i][j] = i * 6 + j + 1;

   /* Create a file. */
   file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

   /* Create the data space for the dataset. */
   dims[0] = 4;
   dims[1] = 6;
   dataspace_id = H5Screate_simple(2, dims, NULL);

   /* Create the dataset. */
   dataset_id = H5Dcreate(file_id, "/dset", H5T_STD_I32BE, dataspace_id,
                          H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);

   /* Write the dataset. */
   status = H5Dwrite(dataset_id, H5T_NATIVE_INT, H5S_ALL,
                     H5S_ALL, H5P_DEFAULT, dset_data);
```

```
    /* Terminate access to the data space. */
    status = H5Sclose(dataspace_id);

    /* Close the dataset. */
    status = H5Dclose(dataset_id);

    /* Close the file. */
    status = H5Fclose(file_id);
}
```

Listing 4: ***Writing a dataset using Damaris***

```
#include "damaris.h"

int main() {

    int status, core_id = 0;
    int i, j, dset_data[4][6];
    /* Initialize Damaris client. */
    DC_initialize("config.xml",core_id);

    /* Initialize the dataset. */
    for (i = 0; i < 4; i++)
        for (j = 0; j < 6; j++)
            dset_data[i][j] = i * 6 + j + 1;

    /* Write the dataset. */
    status = DC_write("dset", dset_data, 0);

    /* Finalize Damaris. */
    DC_finalize();
}
```

Listing 5: ***Associated XML configuration***

```
<layout name="dset_layout" type="int"
        dimensions="4,6" language="C" />
<variable name="dset" layout="dset_layout" persistency="immediate" />
```