Towards a storage backend optimized for atomic MPI-I/O for parallel scientific applications

Viet-Trung Tran

Towards a storage backend optimized for atomic MPI-I/O for parallel scientific applications

Viet-Trung Tran*
ENS Cachan, Brittany
IRISA
Rennes, France
Email: viet-trung.tran@irisa.fr

I. Introduction

Scientific applications are becoming increasingly data-intensive: high-resolution simulations of natural phenomenas, climate modeling, large-scale image analysis, etc. Such applications currently manipulate data volumes in the petabyte scale and with the growing trend of data sizes we are rapidly advancing towards the exabyte scale. In this context, I/O performance has been repeatedly pointed out as a source of bottleneck that negatively impacts the performance of the applications. One of problems that is causing the I/O bottleneck are the I/O access patterns generated by such scientific applications that do not match the I/O access interfaces exposed by the file systems used as underlying storage back-ends.

One particularly difficult challenge in this context is the need to efficiently address the I/O needs of scientific applications [1], [2] that partition multi-dimensional domains into overlapping subdomains. The subdomains need to be processed in parallel and then stored in a globally shared file. Since the file is a flat sequence of bytes, subdomains map to non-contiguous regions in the file. Because the subdomains overlap, under concurrent accesses such non-contiguous regions may interleave in an inconsistent fashion if they are not grouped together as a single atomic transaction. Therefore, *atomicity* of *non-contiguous, overlapping* reads and writes of data from a shared file is a crucial issue.

The purpose of this Phd research aims at addressing this shortcoming of existing approaches by optimizing the storage back-end specifically for the access patterns described above.

II. Problem description

In a large class of scientific applications, especially large-scale simulations, input and output data represents huge spatial domains made of billions of cells associated with a set of parameters (e.g., temperature, pressure, etc.). These spatial domains represent the state of the simulated system at a specific moment in time. They are iteratively refined by the simulation in order to obtain an insight into how the system evolves in time. The large size of the spatial domains triggers the need to parallelize the simulation, by splitting them into subdomains that are distributed and processed by a large number of compute elements, typically MPI processes.

In many such simulations, the contents of a cell depends on the contents of the neighboring cells. Thus, the cells at the border of a subdomain (called "ghost cells") need to be shared by more than one MPI process. In order to avoid repeated exchanges of border cells between MPI processes during the simulation, a large class of applications [1], [2], [3], [4] partition the spatial domain in such way that the resulting subdomains overlap at their borders.

At each iteration, the MPI processes typically dump their subdomains in parallel in a globally shared file, which is then later used to interpret and/or visualize the results. Since the spatial domain is a multidimensional structure which is stored as a single, flat sequence of bytes, the data corresponding to each subdomain maps to a set of non-contiguous regions within the file. This in turn translates to a write-intensive access pattern where the MPI processes *concurrently write a set of non-contiguous regions in the same file*. Moreover, because some subdomains overlap, the non-contiguous regions in the file belonging to such subdomains may overlap too.

In order to obtain a globally shared file that represents the whole spatial domain in a consistent way, it is important to write all non-contiguous regions belonging to the same subdomain in an atomic fashion. This atomicity requirement is defined in MPI standardization as a guarantee that in concurrent, overlapping MPI-I/O write operations (which can possibly involve sets of non-contigous regions), the overlapped regions shall contain data only from one of the MPI processes that participates in the concurrent writes.

However, this work is a non-trivial issue, and that has been difficult to be implemented. Currently, MPI-I/O implementations like ROMIO [5] poorly support MPI atomic mode because of the poor implementations of storage back-ends.

* Advisers: Gabriel Antoniu, INRIA/IRISA, Rennes, France, gabriel.antoniu@inria.fr; Luc Bougé, ENS Cachan/Brittany, IRISA, Rennes, France, Luc.Bouge@bretagne.ens-cachan.fr

Either they do not offer guarantees with respect to atomicity, which means concurrent writes need to be coordinated at application level. Or they offer limited guarantees, such as the POSIX atomicity semantics [6], where a write of a contiguous region is guaranteed to end up as a single sequence of bytes, without interleaving with other writes. However, in our context, the POSIX atomicity guarantees are not sufficient, because a single write operation involves a whole set of non-contiguous regions. Under concurrency, this can lead to an interleaved overlapping that generates inconsistent states.

III. Related work

Previous work has shown that providing MPI atomicity efficiently enough is not a trivial task in practice, especially when dealing with concurrent, non-contiguous I/O, most of which rely on locking-based techniques. Several approaches have been proposed at various levels: at the level of the MPI-I/O layer, at the level of the file system.

A first series of approaches assumes no specific support at the level of the parallel file system This is typically the case of PVFS [7], a widely-used parallel file system which makes the choice of enabling high-performance data access for both contiguous and non-contiguous I/O operations without guaranteeing atomicity at all for I/O operations. For applications where MPI atomicity requirement needs to be satisfied, a solution (e.g. illustrated in [8]) consists in guaranteeing MPI atomicity portably at the level of the MPI-I/O layer. The lack of guarantees on the semantics of I/O operations provided by the file system comes however at a high cost introduced by the use of coarse-grain locking at a higher level. Typically, the whole file is locked for each I/O request and thus concurrent accesses are serialized, which is an obvious source of overhead. To avoid this bottleneck in the case of concurrent non-overlapping accesses to the same shared file, an alternative approach [9] proposes to introduce a mechanism for automatic detection of non-overlapping I/O and thus avoid locking in this case. However, as acknowledged by the authors of this approach, an unnecessary overhead due to the detection mechanism is then introduced for non-overlapping concurrent I/O.

Another class of approches addresses the case where the underlying parallel file system supports POSIX atomicity. Atomic contiguous I/O can then seamlessly be mapped to atomic read/write primitives provided by the POSIX interface. However, POSIX atomicity alone is not enough to provide the necessary atomicity guarantees for applications that exhibit concurrent, non-contiguous I/O operations. It is important to note that, to enable MPI atomicity, both contiguous and non-contiguous I/O requests need to be considered.

Parallel file systems such as GPFS [10] and Lustre [11] provide POSIX atomicity semantics using a distributed locking approach: locks are stored and managed on the storage servers hosting the objects they control. Whereas POSIX atomicity can simply and directly be leveraged for contiguous I/O operations using *byte range locking*, enabling atomic *non-contiguous* I/O based on POSIX atomicity is not efficient. In the default scheme, if we consider a set of non-contiguous byte ranges to be atomically accessed by an individual I/O request, it is then necessary to lock the smallest contiguous byte range that covers all elements of the set of ranges to be accessed. This leads to unnecessary synchronization and thus to a potential bottleneck, since this contiguous byte range also covers unaccessed data that would not need to be locked.

IV. Our approach

We propose a general approach to solve the issue of enabling a high throughput under concurrency for writes of non-contiguous, overlapped regions under MPI atomicity guarantees. This approach relies on three key design principles:

Dedicated API at the level of the storage back-end: Traditional approaches address the issue of providing support for MPI atomic, non-contiguous writes by implementing the MPI-I/O access interface as a layer on top of highly standardized consistency semantics models (e.g. POSIX). The rationale behind this approach is to be able to easily plug in a different storage back-end without the need to rewrite the MPI-I/O layer. However, this advantage comes at a high price: the MPI-I/O layer needs to translate the MPI atomicity into a different consistency model, which greatly limits the potential to optimize for the access patterns that are needed in our context. By contrast, we propose to extend the storage back-end with a data access interface that provides native support for non-contiguous, MPI-atomic data accesses. Using this approach circumvents the need to translate to a different consistency model and enables introducing optimizations directly at the level of the storage back-end, enhancing the potential to implement a a better concurrency control scheme.

Data striping: Data striping a well-known technique to increase the throughput of data accesses, by splitting the huge file where the spatial domain is stored into *chunks* that distributed among multiple storage elements. Using a load-balancing allocation strategy that redirects write operations to different storage elements in a round robin fashion, the I/O workload is distributed by itself, which effectively increases the overall throughput that can be achieved.

Versioning as a key to enhance data access under concurrency: Most storage back-ends manipulate a single version of the file at a time under concurrency by providing locking-based mechanisms that guarantee exclusive access to overlapped regions, in order to eliminate inconsistencies. However, in our context, such an approach does not scale, even if the storage back-end is able to deliver high throughputs. The problem comes from the fact that a locking-based

scheme is expensive, as it enables only one writer at a time to gain exclusive access to a region. Since there are many overlapped regions, this leads to a situation where many writers are idle while waiting for their turn to lock, which greatly limits the potential to achieve a globally high aggregated throughput. In order to avoid this issue, we propose a versioning-based access scheme that avoids the need to lock, effectively eliminating the need of writers to wait for each other.

Our approach is based on *shadowing* techniques [12], which offers the illusion of creating a new standalone snapshot of the file for each update to it but to physically store only the differences and manipulate metadata in such way that the aforementioned illusion is upheld. Starting from the principles introduced in [13], we propose to enable concurrent MPI processes to write their non-contiguous regions in complete isolation, without having to care about overlappings and synchronization. This is made possible by keeping data immutable: new differences are added instead of modifying an existing snapshot. It is at the metadata level where the ordering is done and the overlappings are resolved in such way as to expose a snapshot of the file that looks as if all differences were applied in an arbitrary sequential order.

V. Implementation

Following the approach proposed in Section IV, we have chosen to build the storage back-end on top of *BlobSeer* [13], a versioning-oriented data sharing service specifically designed to meet the requirements of data-intensive applications that are distributed at large scale: *scalable aggregation of storage space* from a large number of participating machines with minimal overhead, support to store *huge data objects*, *efficient fine-grain access* to data subsets and ability to sustain a *high throughput under heavy access concurrency*.

The choice of building the storage back-end on top of BlobSeer was motivated by two factors. First, BlobSeer supports transparent striping of BLOBs into chunks and enables fine-grain access to them, which enables to store each spatial domain directly as a BLOB. This in turn avoids the need to perform data striping explicitly. Second, BlobSeer offers out-of-the-box support for shadowing by generating a new BLOB snapshot for each fine-grain update while physically storing only the differences. This provides a solid foundation to introduce versioning as a key principle to support MPI atomicity.

Since BlobSeer supports atomic reads and writes of contiguous regions only, we can not leverage its versioning-oriented access interface directly to support MPI atomicity which consists of both atomic contiguous and non-contiguous accesses. As mentioned in Section IV, we want to avoid locking and introduce optimizations directly at the level of the storage back-end. These following tunings were done inside BlobSeer in order to support MPI atomicity:

- We extended the access interface of BlobSeer such that it can describe complex non-contiguous data access in a single call. We introduce a series of versioning-oriented primitives that facilitate non-contiguous accesses. These primitives are closely matched the List I/O interface proposal introduced in [14].
- We added support for atomicity of non-contiguous I/O under concurrency by hacking inside BlobSeer's shadowing technique. We built the metadata corresponding to update made by non-contiguous writes such that only consistent snapshots that obey MPI atomicity are published.

A detail implementation of our prototype is described in [15]. To leverage our storage back-end at the level of the MPI-I/O layer, we integrated our prototype into ROMIO [5], a popular MPI-I/O implementation. Since our prototype provides native support for both contiguous and non-contiguous writes under MPI atomicity, ROMIO thus has no need to translate between consistency model, which was source of overhead that impacts the performance of the applications.

VI. Preliminary results

We targeted to build a storage back-end that explicitly optimizes for non-contiguous, overlapped I/O accesses that obey MPI atomicity guarantees. To evaluate our prototype, we compare our approach to the locking-based approach that leverages a POSIX-compatible file system at the level of the MPI-I/O layer, which is the traditional way of addressing MPI atomicity. The Lustre parallel file system [11] was chosen against our prototype. Both Lustre and our prototype are plugged into ROMIO by using their default ROMIO ADIO modules.

Two series of experiments have been realized. Our first experiment aims at evaluating the scalability of our approach when increasing the number of clients that concurrently write non-contigous regions into the same file. In this scenario, we consider the extreme case where each of the clients writes a large set of non-contiguous regions that are intentionally selected in such way as to generate a large number of overlapping that need to obey MPI atomicity. In the second experiment, we performed an evaluation of the performance of our approach using a standard benchmark, *MPI-tile-IO*, that closely simulates the access patterns of real scientific applications that split the input data into overlapped subdomains that need to be concurrently written in the same file under MPI atomicity guarantees.

The preliminary experiments are performed on the Grid'5000 [16] testbed, a reconfigurable, controllable and monitor-able experimental Grid platform gathering 9 sites geographically distributed in France. The results have been reported in [15]. Our approach demonstrated excellent scalability under concurrency when compared to the Lustre-based

approach. It achieved an aggregated throughput ranging from 3.5 times to 10 times higher in several experimental setups, including highly standardized MPI benchmarks specifically designed to measure the performance of MPI-I/O for non-contiguous overlapped writes that need to obey MPI-atomicity semantics.

VII. Conclusions

We have proposed an original versioning-based mechanism that can be leveraged to efficiently address the I/O needs of data-intensive MPI applications involving data-partitioning schemes that exhibit overlapping non-contiguous I/O where MPI-I/O atomicity needs to be guaranteed under concurrency.

Unlike traditional approaches that leverage POSIX-compliant parallel file systems as storage back-ends and employ locking schemes at the level of the MPI-I/O layer, we propose to use versioning techniques as a key principle to achieve high throughput under concurrency while guaranteeing MPI atomicity. We implemented this idea in practice by extending BlobSeer, an existing versioning-oriented, distributed data sharing service, with a non-contiguous data access interface that we directly integrated with ROMIO, a standard MPI-I/O implementation. A number of experiments were performed which demonstrated our approach having excellent scalability under concurrency when compared to locking-based approach.

Such promising results encouraged us to purse interesting future work directions. In particular, one advantage of BlobSeer that we have not developed is to expose its versioning interface directly at application level. The ability to make use of versioning at application level brings several potential benefits, such as in the case of producer-consumer workloads where for example the output of simulations is concurrently used as the input of visualizations. Using versioning at application level could avoid expensive synchronization schemes, which is an acknowledged problem of current approaches.

References

[1] E. Smirni, R. Aydt, A. Chien, and D. Reed, "I/O requirements of scientific applications: An evolutionary view," in *HPDC '02: Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing*. IEEE, 2002, pp. 49–59.

[2] E. Smirni and D. A. Reed, "Lessons from characterizing the I/O behavior of parallel scientific applications," *Performance Evaluation*, vol. 33, no. 1, pp. 27–44, 1998.

[3] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed, "Input/Output characteristics of scalable parallel applications," in *SC '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '95. New York, NY, USA: ACM, 1995.

[4] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Sclatter Ellis, and M. Best, "File-access characteristics of parallel scientific workloads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 10, pp. 1075–1089, October 1996.

[5] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-I/O portably and with high performance," in *IOPADS '99: Proceedings of the 6th Workshop on I/O in parallel and distributed systems*. New York, NY, USA: ACM, 1999, pp. 23–32.

[6] *Information technology - Portable Operating System Interface (POSIX)*. Institute of Electrical & Electronics Engineers, 2009.

[7] I. F. Haddad, "PVFS: A parallel virtual file system for Linux clusters," *Linux Journal*, vol. 2000, November 2000.

[8] R. Ross, R. Latham, W. Gropp, R. Thakur, and B. Toonen, "Implementing MPI-I/O atomic mode without file system support," in *CCGRID '05: Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid*, ser. CCGRID '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 1135–1142.

[9] S. Sehrish, J. Wang, and R. Thakur, "Conflict detection algorithm to minimize locking for MPI-I/O atomicity," in *Euro PVM/MPI '09: Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 143–153.

[10] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, ser. FAST '02. Berkeley, CA, USA: USENIX Association, 2002.

[11] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the Linux Symposium*, 2003.

[12] O. Rodeh, "B-trees, shadowing, and clones," *ACM Transactions on Storage*, vol. 3, no. 4, pp. 1–27, 2008.

[13] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "BlobSeer: Next-generation data management for large scale infrastructures," *Journal of Parallel and Distributed Computing*, vol. 71, pp. 169–184, February 2011.

[14] A. Ching, A. Choudhary, W.-k. Liao, R. Ross, and W. Gropp, "Noncontiguous I/O through PVFS," in *CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing*, ser. CLUSTER '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 405–.

[15] V.-T. Tran, B. Nicolae, G. Antoniu, and L. Bougé, "Efficient support for MPI-IO atomicity based on versioning," INRIA, Research Report RR-7487, 12 2010.

[16] Y. Jégou, S. Lantéri, J. Leduc, M. Noredine, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and T. Iréa, "Grid'5000: a large scale and highly reconfigurable experimental grid testbed," *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, November 2006.