Orthrus: a Framework for Implementing High-performance Collective I/O in the Multicore Clusters

Xuechen Zhang
School of Computer Science
Georgia Institute of Technology
Atlanta, GA, 30332
xczhang@cc.gatech.edu

Jianqiang Ou
ECE Department
Wayne State University
5050 Anthony Wayne Drive
Detroit, MI, 48202, USA
jianqiang.ou@wayne.edu

Kei Davis
CCS Division
Los Alamos National Laboratory
Los Alamos, NM 87545, USA
kei.davis@lanl.gov

Song Jiang
ECE Department
Wayne State University
5050 Anthony Wayne Drive
Detroit, MI, 48202, USA
sjiang@eng.wayne.edu

ABSTRACT

This paper presents a framework, *Orthrus*, that can accommodate multiple collective-I/O implementations, each optimized for some performance aspects, and dynamically select the best performing one accordingly to current workload and system performance bottleneck. We have implemented *Orthrus* in the ROMIO library. Our experimental results with representative MPI-IO benchmarks show that *Orthrus* can significantly improve the performance of collective I/O under various workloads and system scenarios.

Categories and Subject Descriptors

B.4.3 [**Interconnections(Subsystems)**]: [Parallel I/O]

Keywords

Parallel I/O; parallel file systems; collective I/O.

1. INTRODUCTION

Collective I/O is a technique commonly employed in MPI programs that coordinates and reorganizes the requests from the multiple processes of a program before sending them to data nodes. There are two factors that may compromise its performance. One factor is the potentially high cost for the data exchange operation required in collective I/O. The overhead of data exchange can be significant if a large amount of data needs to be exchanged, especially when these aggregators and the processes communicating with them are on different compute nodes. If there are multiple cores on each node—the common case in today's HPC systems—the ideal scenario in terms of minimizing the cost of data exchange is to have each aggregator only be responsible for accessing data for processes on the same node and so have all data exchange occur within individual nodes. However, such a method for assigning file realms to aggregators may compromise the efficiency of I/O operations on the data nodes—the second factor that compromises collective-I/O's performance goal.

Copyright is held by the author/owner(s).
HPDC'13, June 17–21, 2013, New York, NY, USA.
ACM 978-1-4503-1910-2/13/06.

This paper describes a general framework *Orthrus* that allows multiple collective-I/O implementations, each optimized for one (or more) performance aspect(s), such as I/O pattern or relative hardware performance characteristics, to co-exist in a library. Based on a prediction of which would perform best for the current I/O pattern and system load, the framework essentially provides MPI programmers a collective-I/O library that adapts to access pattern changes and other dynamic system characteristics. A major challenge for the framework to achieve effectiveness is in the dynamic prediction of the performance of the various candidate implementations in a given scenario. To solve it, we propose a simple, efficient, accurate, and portable method for selecting the best performer for a collective-I/O operation. It does not take any workload or system information as input and does not involve any complicated modeling or simulation for performance prediction. The key technique is to use performance examination, rather than performance modeling, in the prediction. Each candidate collective-I/O implementation is given opportunities to demonstrate its performance, which is recorded for comparison and selection. By doing so, all of the hard-to-capture information is reflected in the actual performance of a candidate's examination run, which provides an accurate prediction of the performance that would be exhibited should the candidate be selected for executing the collective I/O in the near future.

2. THE DESIGN OF *ORTHRUS*

The crux of the *Orthrus* framework is to know how each candidate implementation would perform if it were used to execute a collective-I/O function call. The strategy of *Orthrus* is to apply each candidate implementation on the real execution of the function call to examine its actual performance. Certainly in the library one application-level call cannot be executed more than once: the overhead would be excessively large and the caching effect would invalidate the performance results from all but the first call. However, with different candidates executing different calls we would need to ensure that their performance results were comparable by invoking these candidates within the same workload, i.e. have the data requested by the calls have the same pattern.

In addition, the performance examination period must constitute only a small fraction of the total collective-I/O time because many candidates might not provide efficient I/O service. To characterize iteration's data access pattern *Orthrus* uses a number of factors, including name of accessed file, number of processes issuing I/O requests, number of requests, and request sizes, and their relative offsets in the file. We refer to these factors collectively as the call's *signature*. Two collective I/O calls are considered to have the same pattern and comparable performance results only when their signatures are the same.

Orthrus carries out its operations as follows. When a collective-I/O call is made, first its signature is compared to the signature of the previous call. If they are not the same, a new candidate examination period is started. If they are the same, then either the system is in a candidate examination period, in which case *Orthrus* keeps testing a candidate implementation, or the system uses the currently selected candidate implementation to execute the call. When a new candidate examination period is started, calls are serviced by the candidates in rotation, each for a fixed number of times (three by default), as long as the signature does not change. The throughput of each candidate is computed as an average to minimize the effect of transient changes in the execution environment. When the examination period ends without a change in signature, the candidate with the highest throughout is selected to execute the calls until the signature changes. During a regular execution period, even if the collective-I/O calls do not change their access pattern, the dynamic system environment, such as communication and I/O request traffic initiated by other programs, could change, and accordingly the best performing candidate for this program might change. To detect such system variations we monitor the throughput of the selected candidate. If the deviation of the throughout exceeds a certain threshold (15% by default) of the average recorded in the candidate examination period the system returns to the examination mode.

To test our idea we implemented two strategies. One, *core-first*, constrains data exchange to be intra-node. In the current implementation of *core-first* there is one aggregator per compute node to represent all the processes running on that node. We choose as the aggregator the process in that node requesting the largest amount of data. The processes' I/O requests are collected by the aggregator, which then sorts them in the ascending order according to the offsets of their requested data. The second strategy, *disk-first*, is designed to optimize for disk efficiency [3]. It sets up the same number of aggregators as the number of data nodes, each collecting and sending requests to one data node. Before that it sorts the requests as *core-first* does. However, *disk-first* ensures that each data node receives requests in one well-ordered sequence in the execution of a collective I/O call, while in *core-first* each data node receives requests from multiple (possibly all) aggregators, and the order of requests from different aggregators is essentially random. We also plug the ROMIO collective [2] implementation into the *Orthrus* framework.

3. PERFORMANCE EVALUATION

We evaluate *Orthrus* when a program using collective I/O runs on a PVFS2 cluster with 11 compute nodes (8-core Xeon) and 6 data nodes. The program (*Matrix*) simulates the I/O pattern exhibited with access of a large matrix file by multiple processes in a block-by-block fashion. To reveal *Orthrus*'s adaptability to the changes in the run-time environment, from the 60th second of *Matrix*'s execution we ran the NPB/FT program [1] to generate inter-node all-to-all communication. Figure 1 shows the instantaneous I/O throughput of *Matrix*, at each multiple of 250 iterations, using each of the four strategies. Initially *Orthrus* selects *disk-first*. After the injection of the external communication traffic the inter-node bandwidth available to *Matrix* is reduced and its data exchange becomes more expensive. Accordingly the throughputs of *ROMIO* and *disk-first* are reduced by up to 46% and 60%, respectively, and *core-first* shows its performance advantage. As shown in Table 1, during the execution of the benchmark *core-first* generates a much smaller number of IP segments as reported by /proc/net/snmp. When *Orthrus* detects the throughout degradation with *disk_first* it re-evaluates the candidates and responsively switches to *core-first*, significantly reducing inter-node data exchange.

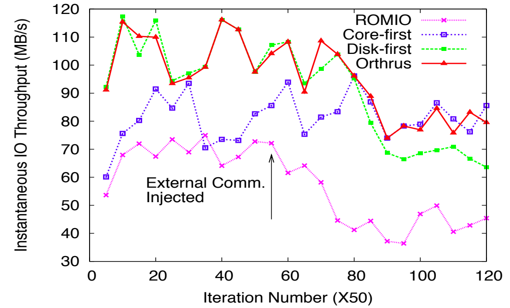

Figure 1: Instantaneous throughput measured for the four schemes after execution of every 250 iterations. The NPB/FT program starts executing at the 60th second.

Schemes	*ROMIO*	*core-first*	*disk-first*
# of In Segs.	701,600	337,257	701,056
# of Out Segs.	1,385,008	322,884	1,412,066

Table 1: Average number of incoming and outgoing IP segments transmitted at each compute node

4. CONCLUSION

We have presented the design and implementation of *Orthrus*, a framework that can host multiple collective-I/O implementations and adaptively select the one that provides the highest I/O throughout according to current workload and system dynamics. We have implemented *Orthrus* in the ROMIO library and tested it with multiple sample collective-I/O implementations. Our results show that *Orthrus* can significantly improve the throughput of collective I/O for representative MPI-IO benchmarks.

5. REFERENCES

[1] "FT: Discrete 3D Fast Fourier Transform", *URL: http://www.nas.nasa.gov/publications/npb.html*

[2] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO", In *Frontiers'99*.

[3] X. Zhang, S. Jiang, and K. Davis, "Making Resonance a Common Case: A High-Performance Implementation of Collective I/O on Parallel File Systems", In *IPDPS'09*.