Portable and Scalable MPI Shared File Pointers

Jason Cope[1], Kamil Iskra[1], Dries Kimpe[2], and Robert Ross[1]

[1] Argonne National Laboratory, Argonne, IL 60439
[2] University of Chicago, Chicago, IL 60637
{copej,iskra,dkimpe,rross}@mcs.anl.gov

Abstract

While the I/O functions described in the MPI standard included shared file pointer support from the beginning, the performance and portability of these functions have been subpar at best. ROMIO [1], which provides the MPI-IO functionality for most MPI libraries, to this day uses a separate file to manage the shared file pointer. This file provides the shared location that holds the current value of the shared file pointer. Unfortunately, each access to the shared file pointer involves file lock management and updates to the file contents. Furthermore, support for shared file pointers is not universally available because few file systems support native shared file pointers [5] and a few file systems do not support file locks [3].

Application developers rarely use shared file pointers, even though many applications can benefit from this file I/O capability. These applications are typically loosely coupled and rarely exhibit application-wide synchronization. Examples include application tracing toolkits [8,4] and many-task computing applications [10]. Other approaches to the shared file pointer I/O models frequently used by these application classes include file-per-process, file-per-thread, and file-per-rank approaches. While these approaches work relatively well at smaller scales, they fail to scale to leadership-class computing systems because of the intense metadata loads generated they generate. Recent research identified significant improvements from using shared-file I/O instead of multifile I/O patterns on leadership-class systems [6].

In this paper, we propose integrating shared file support into the I/O forwarding layer commonly found on leadership-class computing systems. I/O forwarding middleware, such as the I/O Forwarding Scalability Layer (IOFSL) [9,2], bridges the compute and I/O subsystems of leadership-class computing systems. This middleware layer captures all file I/O requests generated by applications executing on compute nodes and forwards them to dedicated I/O nodes. These I/O nodes, a common hardware feature of leadership-class computing systems, execute the I/O requests on behalf of the application. The I/O forwarding layer on these system is best suited to provide and manage shared file pointers because it has access to all application I/O requests and can provide enhanced file I/O capabilities independent of the system and I/O software stack. By embedding this capability into the I/O forwarding layer, applications developers can utilize shared file pointers for a variety of file I/O APIs (MPI-IO, POSIX, and ZOIDFS),

synchronization levels (collective and independent I/O), and computing systems (IBM Blue Gene and Cray XT systems).

We are adding several features to IOFSL and ROMIO to enable portable MPI-IO shared file pointer access. In prior work, we extended the ZOIDFS API [2] to provide a distributed atomic append capability. Our current work extends and generalizes this capability to provide shared file pointers as defined by the MPI standard. First, we created a per file shared (key,value) storage space. This capability allows users of the API to instantiate an instance of a ZOIDFS file handle and associate file state with the handle (such as the current position of a file pointer). Since a ZOIDFS file handle is a persistent, globally unique identifier linked to a specific file, this does not result in extra state for the client. To limit the amount of state stored within the I/O node and to enable recovery from faults, we are integrating purge policies for the key value store. Example policies include flushing data to other IOFSL servers or persistently storing this data in extended attribute fields of the target file.

In prior work, we implemented a distributed atomic append by essentially implementing a per file, system wide shared file pointer. In our current work, we instead require a shared file pointer per MPI file handle. This is easily implemented by storing the current value of the shared file pointer in a key uniquely derived from the MPI file handle. We modified ROMIO to generate this unique key. When a file is first opened, a sufficiently large, random identifier is generated. This identifier is subsequently used to retrieve or update the current value of the shared file pointer. To avoid collisions, we rely on the fact that the key space provided by IOFSL supports an exclusive create operation. In the unlikely event that the generated identifier already exists for the file, ROMIO simply generates another one.

By providing set, get, and atomic increment operations, the IOFSL server is responsible for shared file pointer synchronization. This precludes the need for explicit file lock management for shared file pointer support. Overall, few modifications to ROMIO were required. Before executing a shared read or write, ROMIO uses the key store to atomically increment and retrieve the shared file pointer. It then subsequently accesses the file using an ordinary independent I/O operation. To simplify fault tolerance, we plan to combine the I/O access and the key update into one operation. ROMIO's `MPI_File_close` method removes the shared file pointer key in order to limit the amount of state held by the I/O nodes. For systems such as the Cray XT series, where I/O nodes are shared among multiple jobs, we automatically purge any keys left by applications that failed to clean up the shared file pointer, for example because of unclean application termination. On systems employing a dedicated I/O node, no cleanup is necessary, since the I/O node (and the IOFSL server) is restarted between jobs.

These modifications provide a low-overhead, file-system-independent, shared file pointer implementation for MPI-IO on those systems supported by IOFSL. Unlike other solutions, our implementation does not require a progress thread or hardware-supported remote memory access functionality [7].

Acknowledgments

This work was supported by the U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

1. ROMIO: A high-performance, portable MPI-IO implementation. http://www.mcs.anl.gov/romio/
2. Ali, N., Carns, P., Iskra, K., Kimpe, D., Lang, S., Latham, R., Ross, R., Ward, L., Sadayappan, P.: Scalable I/O Forwarding Framework for High–Performance Computing Systems. In: IEEE International Conference on Cluster Computing 2009 (2009)
3. Carns, P., Ligon, W., Ross, R., Thakur, R.: PVFS: A parallel file system for linux clusters. Proceedings of the 4th Annual Linux Showcase and Conference (2000)
4. Chan, A., Gropp, W., Lusk, E.: An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. Scientific Programming 16(2-3), 155–165 (2008)
5. Freedman, C., Burger, J., DeWitt, D.: SPIFFI-A Scalable Parallel File System for the Intel Paragon. IEEE Transactions on Paralllel and Distributed Systems 7(11), 1185–1200 (1996)
6. Lang, S., Carns, P., Latham, R., Ross, R., Harms, K., Allcock, W.: I/O performance challenges at leadership scale. In: Proceedings of Supercomputing (November 2009)
7. Latham, R., Ross, R., Thakur, R.: Implementing MPI-IO Atomic Mode and Shared File Pointers Using MPI One-Sided Communication. International Journal of High Performance Computing Applications 21(2), 132–143 (2007), http://hpc.sagepub.com/cgi/content/abstract/21/2/132
8. Müller, M., Knüpfer, A., Jurenz, M., Lieber, M., Brunst, H., Mix, H., Nagel, W.: Developing scalable applications with vampir, vampirserver and vampirtrace. In: PARCO 2007 (2007)
9. Ohta, K., Kimpe, D., Cope, J., Iskra, K., Ross, R., Ishikawa, Y.: Optimization Techniques at the I/O Forwarding Layer. In: IEEE International Conference on Cluster Computing 2010 (2010)
10. Raicu, I., Foster, I., Wilde, M., Zhang, Z., Iskra, K., Beckman, P., Zhao, Y., Szalay, A., Choudhary, A., Little, P., Moretti, C., Chaudhary, A., Thain, D.: Middleware support for many-task computing. Cluster Computing 13, 291–314 (September 2010), http://dx.doi.org/10.1007/s10586-010-0132-9