# OMPIO: A Modular Software Architecture for MPI I/O

Mohamad Chaarawi[1], Edgar Gabriel[1], Rainer Keller[2], Richard L. Graham[3], George Bosilca[4], and Jack J. Dongarra[4]

[1] Department of Computer Science, University of Houston, Houston, TX, USA
{mschaara,gabriel}@cs.uh.edu
[2] High Performance Computing Center Stuttgart (HLRS), Stuttgart, Germany
keller@hlrs.de
[3] Oak Ridge National Laboratory (ORNL), Oak Ridge, TN, USA
rlgraham@ornl.gov
[4] Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA
{bosilca,dongarra}@eecs.utk.edu

**Abstract.** I/O is probably the most limiting factor on high-end machines for large scale parallel applications as of today. This paper introduces OMPIO, a new parallel I/O architecture for Open MPI. OMPIO provides a highly modular approach to parallel I/O by separating I/O functionality into smaller units (frameworks) and an arbitrary number of modules in each framework. Furthermore, each framework has a customized selection criteria that determines which module to use depending on the functionality of the framework as well as external parameters.

## 1 Introduction and Motivation

Amdahl's law stipulates that the scalability of a parallel application is limited by its least scalable section. For many scientific applications, the scalability limitation comes from the performance of I/O operations. MPI [12], the most popular parallel programming paradigm on clusters today introduced the notion of parallel I/O in version two of the specification. Although its adoption by the end-users has been modest, it has been shown, that in combination with parallel file systems, MPI I/O can significantly improve the performance of I/O operations [7, 14] compared to sequential I/O.

Switching from the sequential Fortran or C I/O routines to MPI I/O potentially requires significant work by application developers, due to the fact that many MPI I/O features do not have counterparts in other I/O specifications. However, application developers are more willing to make drastic investment in rewriting substantial part of the application if the direct effect of the investment is a significant reduction in the application execution time, or a more robust scalability. This is however not always the case. The reasons for the limited performance often observed with MPI I/O is the diversity of existing I/O solutions which make each I/O environment (almost) unique. The performance of parallel I/O operations is influenced by the file system utilized, as well as by the number

Y. Cotronis et al. (Eds.): EuroMPI 2011, LNCS 6960, pp. 81–89, 2011.
© Springer-Verlag Berlin Heidelberg 2011

of storage servers, the I/O bandwidth of each storage server, the network connectivity in-between the storage servers as well as between the storage servers and compute nodes, and the network interconnect and its OS-level parameters used for the MPI level communication. Additionally, application characteristics such as frequency and volume of I/O operations as well as the algorithm utilized to implement the functionality (e.g., the collective I/O operations), will greatly contribute towards the I/O performance observed by the end-user.

In this paper, we present a new parallel I/O architecture for Open MPI called OMPIO. The goal of OMPIO is to provide the infrastructure that allows to deal with the challenges of parallel I/O in a flexible manner, and consequently allows to optimize the performance of I/O operation for different applications and hardware configurations. At the core of the architecture is the separation of parallel I/O functionality into frameworks. This allows to encapsulate various aspects of parallel I/O into smaller functional units, such as dealing with file system specific operations, individual I/O, collective I/O, or shared file pointer operations. Each framework has typically multiple modules providing the required functionality, each module being designed for different scenarios. We argue, that the selection criteria that determines which module is being used is highly dependent on the functionality provided by a framework and on external parameters such as the file system utilized, hardware configuration, process placement by the batch scheduler or application characteristics.

The remainder of the paper is organized as follows: Section 2 discusses the related work in the area and makes the case why currently existing approaches, provided by most popular MPI I/O libraries, do not offer the required flexibility to deal with the diversity of the available I/O subsystems. Section 3 describes the design of the new OMPIO module and its associated set of frameworks. In section 4 we present a case study where we evaluate two different benchmarks on two different platforms using a PVFS2 and a Lustre file system. The results demonstrates a the available functionality in OMPIO and exposes some of the advantages of the new architecture for collective I/O operations. Finally, section 5 summarizes the paper and presents the ongoing work in this area.

## 2 Related Work

The most widely used implementation of MPI I/O as of today is ROMIO [16]. ROMIO is part of the MPICH [8] distribution and is the basis for many I/O libraries used in other public domain MPI libraries such as Open MPI and commercial MPI implementations. ROMIO abstracts file systems specific operations using the Abstract-Device Interface for Parallel I/O, called ADIO [15], which reduces the number of routines that have to be implemented in order to support a new file system. ROMIO also has the ability to support multiple file systems simultaneously, e.g., in case an application opens a file on two different file systems. However, the selection criteria which ADIO module shall be used as of today is based on the file system only. Krimpe et al. [9] allowed for non-file system specific selection of some modules by prepending a keyword to the name of

the file. The solution presented in this paper has two main advantages compared to ROMIO. First, the usage of different frameworks allows a more fine grained separation of functionality than the approach used in ROMIO. Second, OMPIO introduces the ability to make non-file system specific module selection that do not require any modifications of the end-user application.

In [4], the authors introduced the ability to easily modify parameters of collective I/O operations. However, the work focused entirely on collective I/O, leaving other aspects of parallel I/O unmodified. Furthermore, it is our understanding that the framework described in this paper does not allow for easy deployment of new collective I/O algorithms, but is restricted to modifying parameters of the provided collective read/write operations.

The Adaptable IO System (ADIOS) [11] is an I/O library designed to allow end users to select the best I/O method based on the application's access pattern and the underlying file system and hardware at hand. The access pattern of the application is described in a separate input file, providing some of the functionality that the file view provides in MPI I/O. Thus, the ADIOS library has the ability to utilize POSIX style I/O operations, MPI I/O or any other supported API without having to change the application itself. ADIOS also introduces a file format called BP, which serves as an intermediate format that is easily converted to other standard file formats such as HDF5.

## 3 The OMPIO Set of Frameworks

The Open MPI Project [5] is an open source implementation of the MPI specification that is developed and maintained by a consortium of academic, research, and industry partners. The internal architecture of Open MPI is built around the Modular Component Architecture (MCA) [1], which allows for compile or run time selection of the components used by the MPI library. A component framework in Open MPI is dedicated to a single task, such as providing parallel job control or performing MPI collective operations. Modules are self-contained software units that can configure, build, and install themselves. Modules adhere to the interface prescribed by the component framework that they belong to, and provide requested services to higher-level tiers and other parts of MPI. This mechanism allows a single Open MPI installation to simultaneously support various network interconnects. The new OMPIO module is a module of the IO framework of Open MPI, and is designed to co-exist with ROMIO, the parallel I/O library used in all released versions of Open MPI. Generally speaking, when a file is being opened, both OMPIO and ROMIO are being queried, and the module returning the higher priority value is used to for the subsequent I/O operations.

The main goals of OMPIO are three fold. First, it increases the modularity of the parallel I/O library by separating functionality into distinct sub-frameworks. Second, it allows frameworks to utilize different run-time decision algorithms to determine which module to use in a particular scenario, enabling non-file system specific decisions. Third, it improves the integration of parallel I/O functions

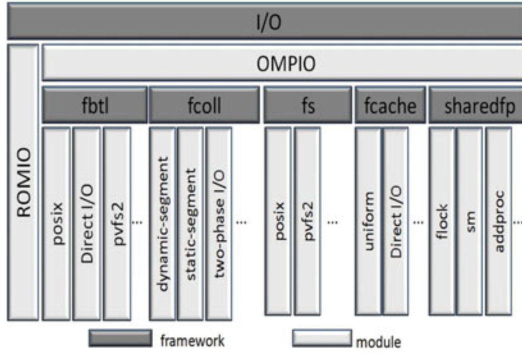


**Fig. 1.** Overview of the OMPIO component and its frameworks

with other components of Open MPI, most notably the derived data type engine and the progress engine. The integration with the Open MPI progress engine allows for seamless progress of non-blocking I/O operations. The integration with the derived data type engine has multiple advantages, most notably faster decoding of derived data types and the usage of optimized data type to data type copy operations. Furthermore, OMPIO has the ability to use the data conversion functionality of the data type engine, without having to provide the according (fairly complex) functions.

Similarly to the selection logic in other Open MPI frameworks, each sub-framework of the OMPIO component determines in `MPI_Init` the list of available modules and opens them. Upon opening a file using `MPI_File_open`, the OMPIO module initializes each sub-framework for that particular file. A framework will query each available module which in return responds with a priority value indicating its readiness to be used for the given file. As an example, a module providing a POSIX style interface might return a low priority value for most files, indicating that it could be used for the according operations. However, a specific module optimized for the given file system or installation will typically return a higher priority and will be chosen for the subsequent I/O operations. Each sub-framework or module will typically have different rules on when to return a high priority. Conditions include the file systems type, location of participating processes, network parameters or user specified settings. In the following we present briefly each sub-framework and the currently available modules.

### 3.1 The *file system* Framework (fs)

The *fs* framework abstracts out file manipulation operations such as opening, closing, and deleting a file. The semantics of most of the operations are collective. Furthermore, file system specific info objects have to be interpreted and applied within this module. The *fs* framework has as of today a module providing generic POSIX interface, and separate modules for Lustre and a PVFS2 which allow to modify stripe size and stripe depth when creating a new file.

### 3.2 The *file byte-transfer layer* Framework (fbtl)

The *fbtl* framework provides the abstraction for all individual read and write operations. A module implementing the *fbtl* interfaces has to provide, as of today, blocking and non-blocking read and write operations, as well as a progress function that will be registered with the Open MPI progress engine in order to enforce the progress of pending I/O calls. The interfaces of the read and write operations currently take a list containing tuples of <*memory address*, *length in bytes*, *file offset*>. Currently available are *fbtl* modules which provides POSIX semantics, and a module utilizing native PVFS2 read/write operations. Note however, that the current OMPIO implementation only supports blocking operations for both *fbtl*. Support for non-blocking individual operations are expected to be available in the near future.

### 3.3 The *collective I/O* Framework (fcoll)

This framework provides interfaces for collective file I/O operations. In contrary to the other frameworks which are part of the OMPIO set, the *fcoll* framework triggers the selection logic not upon opening a file, but every time the file view is being set.

Collective I/O operations are a very good example for the necessity to have non-file system specific selection logic. As an example, the Lustre file system serving the Jaguar system at Oak Ridge National Laboratory and the Lustre file system at our development cluster at the University of Houston have fundamentally different characteristics, such as number of Object Storage Targets (OSTs), bandwidth of each OST, and network characteristics between compute nodes and OSTs. Despite the fact that both installations utilize the same file system, different algorithms for collective I/O operations have to be used on these two installations in order to maximize the I/O performance of an application, since some optimizations only make sense for certain hardware configurations.

The *fcoll* framework has five different modules to choose from, one module for each of the following algorithms: two-phase I/O, static segmentation, dynamic segmentation, individual algorithm and an algorithm where each I/O node is only receiving requests by a single aggregator process. The first three algorithms have been extended to include a heuristic which automatically determines the number of aggregator processes to be used [2].

The current selection logic is based on an extensive set of tests that has been executed on various platforms and file systems. Among the factors that influence which module is being used is the average contiguous data chunk accessed by each process, gaps size in the file view between processes, and file system characteristics, such as the stripe size and the minimal data required to saturate the read/write bandwidth of one process. We omit here details of this selection logic due to space limitations, more details may be found in [2].

### 3.4 The *file cache* Framework (fcache)

The *fcache* framework provides the ability to set and retrieve information related to the file layout, such as the number of storage servers used, list of storage

servers, and stripe depth for each file separately. The main functionality of the *fcache* is to provide a mapping of <*offset into file*, *length in bytes*> to a list of <*storage server id*, *local offset on that storage server*, *local length*>. This allows for various optimizations for example for collective I/O operations. As of today, only a trivial module is available for UFS style file systems which provides only basic information.

### 3.5 The *shared file pointer* Framework (sharedfp)

The *sharedfp* framework provides the functionality required to manage the shared file pointer, allowing for generic and architecture specific optimizations. Although shared file pointer operations have been sparingly used in the community, due to the fact that in the most general case an implementation of shared file pointer operations will be slow, it is well understood that, for particular architectures or settings, efficient implementations do exist. As an example, if the shared file pointer is utilized by processes in a communicator that spans a single physical node, the shared file pointer can be efficiently implemented using a small shared memory segment. Alternatively, some of the strict requirements of a shared file pointer can be relaxed for certain usage scenarios, allowing the utilization of individual files per process. In doing so, the consolidation to a single output file may be delayed to the post-processing step [10].

We have explored a number of shared file pointer algorithms in [10], which are currently being converted into modules in the near future along with the selection logic, which will include process placements as one of the key criteria to determine which module to use.

## 4 Experimental Results

Two application benchmarks are used for evaluation on two different platforms. The Shark cluster at the University of Houston consists all-in-all of 29 nodes, with a PVFS2 file system consisting of 22 server nodes where each server uses its local disk space as the back-end storage. The stripe size of the file system is 64 kB. The file system uses GE as the network interconnect.

The Deimos PC Farm at TU Dresden has 724 compute nodes with a Lustre file system exported by 11 I/O servers via a separate 4x SDR InfiniBand network. The file system is organized in 48 OSTs with a stripe size of 1 MB.

The first benchmark used is MPI-TILE-IO [13], a test application that implements tile access to a two dimensional dense dataset. This type of workload is seen in tiled displays (for small numbers of tiles) and in some numerical applications. Several parameters that control the file access and 2D distribution of the processes can be modified at runtime. The results shown report two tile sizes of 64 Bytes (2048 x 1600 elements) and 1 MB (20 x 15 elements), which represents a non-contiguous and contiguous access respectively. We report the maximum bandwidth achieved across five executions of every test case.

**Table 1.** Performance comparison between OMPIO's and ROMIO's default setting using MPI-TILE-IO

| Platform/Number of Processes/Tile Size | ROMIO | OMPIO |
|---|---|---|
| Shark/81(9x9)/64B | 303.8 MB/s | 591.1 MB/s |
| Shark/81(9x9)/1MB | 290.1 MB/s | 625.4 MB/s |
| Deimos/256(16x16)/64B | 411.6 MB/s | 2167.1 MB/s |
| Deimos/256(16x16)/1MB | 517.7 MB/s | 2491.2 MB/s |

The results shown in table 1 show the results using the MPI-TILE-IO benchmark over Shark with PVFS2 and Deimos with Lustre. ROMIO has been executed with default parameters, i.e. without passing any additional hints or parameters to the library in order to have a base-line number from the performance perspective. In OMPIO we set the optimal cycle buffer size determined for the according file system. OMPIO chooses the two-phase I/O module for the 64 Byte tile size and the dynamic segmentation module for the 1 MB tile size. The heuristic determining the number of aggregators automatically leads to 81 aggregator processes on Shark and 256 aggregators on Deimos in these test cases. Collective I/O operations in ROMIO use the two-phase I/O algorithm with one aggregator per node as the default setting. The results show that OMPIO leads to a performance benefit in these test cases which can be attributed mostly to the different number of aggregators used by OMPIO and the different algorithm used in the first case. However, the main message of this result is not the performance benefit observed due to the different number of aggregators, instead the flexibility to switch seamlessly between different collective I/O module for the same application due to the component architecture of OMPIO.

In the second scenario, we demonstrate the flexibility and modularity of the OMPIO architecture by using the Open Tool for Parameter Optimizations (OTPO) [3] to tune collective I/O operations and parameters for a given test case. OTPO is a tool which can be used to optimize runtime parameters of Open MPI. The tool takes in an input file which contains the names of parameters to be explored along with the according rules on how to modify the parameters, and the name of the benchmark/application to be executed when exploring the parameter space. After the optimization, OTPO reports the set of parameter combination(s) which lead to the lowest execution time. In this particular scenario, we used the Latency-IO micro-benchmark developed as part of the latency test suite [6].

The parameter file that is passed to OTPO contains the different collective I/O algorithms that are available in OMPIO and a some parameters of these modules. Thus, the parameters to be optimized and according values are:

- *fcoll module:* static, dynamic, individual, two-phase
- *number of aggregators:* 5, 10, 20, 40
- *cycle buffer size:* 2 MB, 20 MB, 32 MB, 64 MB, 128 MB

In this case 65 different parameter combinations were generated from the input file, the winning combination was (dynamic, 20, 32 MB). While there were other parameter combinations that provided performance close to the wining combination, only two out of 65 parameter combinations were within 10% of the best performance value. Those combinations were (dynamic, 20, 20 MB) and (static, 20, 32 MB). Overall, 23 out of 65 parameter combinations were within 25% of the best performance.

This type of tuning of collective I/O parameters is possible because of the OMPIO architecture and allows end-users and system administrators to pre-tune a module for a particular application or scenario without having to recompile the MPI library.

## 5 Conclusion

This paper introduces OMPIO, a newly developed parallel I/O architecture designed for Open MPI. OMPIO introduces a modular architecture for parallel I/O that separates functionality into different sub-frameworks and allows for a highly flexible composition of modules in order to provide MPI I/O functionality, and reduces the barriers to develop new, site-specific modules and configurations. We demonstrate the usability of OMPIO by executing various benchmarks on a PVFS2 and Lustre file system on two different clusters. OMPIO is currently being evaluated by the Open MPI group and should be publicly available by the end of the summer, with the initial intent of serving as a research vehicle into parallel I/O.

The ongoing work includes multiple areas. First and foremost, we are working on implementing the non-blocking I/O operations within the OMPIO framework. This will support most of the operations defined in the MPI-2.2 specification and will open the door for further optimizations for collective I/O operations. Second, we are continuing to improve our collective I/O algorithms, most notably by exploring new grouping strategies for the dynamic and static segmentation algorithms.

## References

1. Barrett, B., Squyres, J.M., Lumsdaine, A., Graham, R.L., Bosilca, G.: Analysis of the component architecture overhead in Open MPI. In: Proc. of the 12th European PVM/MPI Users' Group Meeting, Sorrento, Italy, pp. 175–182 (September 2005)
2. Chaarawi, M.: Optimizing Parallel I/O Operations for High Performance Computing. Ph.D. thesis, Department of Computer Science, University of Houston (2011)
3. Chaarawi, M., Squyres, J.M., Gabriel, E., Feki, S.: A tool for optimizing runtime parameters of open MPI. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 210–217. Springer, Heidelberg (2008)
4. Coloma, K., Ching, A., Choudhary, A., Liao, W., Ross, R., Thakur, R., Ward, L.: A New Flexible MPI Collective I/O Implementation. In: Proceedings of the 2006 IEEE International Conference on Cluster Computing, pp. 1–10 (2006)

5. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B.W., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 97–104. Springer, Heidelberg (2004)
6. Gabriel, E., Fagg, G.E., Dongarra, J.J.: Evaluating dynamic communicators and one-sided operations for current MPI libraries. International Journal of High Performance Computing Applications 19(1), 67–79 (2005)
7. Gabriel, E., Venkatesan, V., Shah, S.: Towards high performance cell segmentation in multispectral fine needle aspiration cytology of thyroid lesions. Computational Methods and Programs in Biomedicine 98(3), 231–240 (2009)
8. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing 22(6), 789–828 (1996)
9. Kimpe, D., Ross, R., Vandewalle, S., Poedts, S.: Transparent log-based data storage in MPI-IO applications. In: Cappello, F., Herault, T., Dongarra, J. (eds.) PVM/MPI 2007. LNCS, vol. 4757, pp. 233–241. Springer, Heidelberg (2007)
10. Kulkarni, K., Gabriel, E.: Evaluating Algorithms for Shared File Pointer Operations in MPI I/O. In: Allen, G., Nabrzyski, J., Seidel, E., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2009. LNCS, vol. 5544, pp. 280–289. Springer, Heidelberg (2009)
11. Lofstead, J., Zheng, F., Klasky, S., Schwan, K.: Adaptable, metadata rich IO methods for portable high performance IO. In: Proc. of IPDPS 2009, Rome, Italy, May 25-29 (2009)
12. Message Passing Interface Forum: MPI-2.2: Extensions to the Message Passing Interface (September 2009), `http://www.mpi-forum.org`
13. Ross, R.: Parallel I/O Benchmarking Consortium, `http://www.mcs.anl.gov/research/projects/pio-benchmark`
14. Ross, R., Nurmi, D., Cheng, A., Zingale, M.: A Case Study in Application I/O on Linux Clusters. In: ACM/IEEE Supercomputing Conference, Denver, CO, USA (2001)
15. Thakur, R., Gropp, W., Lusk, E.: An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In: Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computation, pp. 180–187. IEEE Computer Society Press, Los Alamitos (1996)
16. Thakur, R., Gropp, W., Lusk, E.: On implementing MPI-IO portably and with high performance. In: Proc. of the 6th Workshop on I/O in Parallel and Distributed Systems, pp. 23–32 (1999)