Design and Evaluation of Nonblocking Collective I/O Operations

Vishwanath Venkatesan[1], Mohamad Chaarawi[1], Edgar Gabriel[1], and Torsten Hoefler[2]

[1] Department of Computer Science, University of Houston,
{venkates, mschaara, gabriel}@cs.uh.edu
[2] Blue Waters Directorate, University of Illinois
htor@illinois.edu

Abstract. Nonblocking operations have successfully been used to hide network latencies in large scale parallel applications. This paper presents the challenges associated with developing nonblocking collective I/O operations, in order to help hiding the costs of I/O operations. We also present an implementation based on the libNBC library, and evaluate the benefits of nonblocking collective I/O over a PVFS2 file system for a micro-benchmark and a parallel image processing application. Our results indicate the potential benefit of our approach, but also highlight the challenges to achieve appropriate overlap between I/O and compute operations.

1 Introduction

Overlapping computation and communication is a standard technique to optimize the performance of parallel applications. This technique allows to hide latencies and improve bandwidth of data transfers to remote processes. This functionality is offered to the user through a special nonblocking interface, which allows to start operations and check for completions later. Benefits of nonblocking operations have been demonstrated for point-to-point [1, 2] and nonblocking collective [3, 4] operations. The Message Passing Interface (MPI) standard specifies so called "immediate" versions of some operations. MPI-2.2 offers immediate versions of all point-to-point communication calls and MPI-3.0 will add immediate versions of all collective communication functions. Those special functions return with a handle before the operation is completed. The handle can be used to test and wait for completion of the associated operations.

With the advent of data-intensive computing [5], the input/output from/to disk (I/O) of application data can become a significant bottleneck. This does not only include reading the dataset initially and saving it at the end but also periodic application-level checkpoints and out-of-core processing. In addition to this, while the compute and network power of parallel HPC systems is growing steadily, the performance of the I/O subsystem can often not keep up with this growth. Thus, nonblocking I/O interfaces are important to improve application performance.

In this work, we propose a new interface that is similar to the newly introduced nonblocking collective communication operations and show an optimized implementation of this interface. In particular, the contributions of this paper are as follows:

1. We propose a simple extension to the MPI-2.2 standard to enable the user to specify overlap of I/O operations with other computation and communication operations conveniently.
2. We describe a framework to efficiently implement nonblocking collective I/O routines.
3. We demonstrate an implementation of this framework and performance results on a parallel file system.

The outline of the paper is as follows: section 2 presents the technical challenges associated with nonblocking collective I/O operations. Section 3 evaluates the benefits of nonblocking collective I/O operations, followed by a general discussion on nonblocking collective I/O interfaces in section 4. Finally, section 5 summarizes the contributions of the paper and presents the ongoing work in this domain.

2 Challenges of Nonblocking Collective I/O Operations

In the following, we detail the challenges of providing nonblocking collective I/O operations. For this, we describe first the collective I/O algorithm used and then elaborate the extensions introduced in LibNBC.

2.1 Collective I/O Algorithm

The collective I/O algorithm used for the prototype implementation of nonblocking collective I/O operations is based on the dynamic segmentation algorithm [6]. This algorithm is an extension of the classical two-phase collective I/O algorithm. Similar to two-phase I/O, the main goal of this algorithm is to combine data from multiple processes in order to minimize the number of discontiguous I/O requests. In contrast to two-phase I/O however, the dynamic segmentation algorithm does not create a globally sorted data array based on the offsets in the file. Instead, each aggregator is assigned a group of processes and performs the data gathering/scattering and sorting only within its group. This allows to execute the shuffle step including the sorting and data gathering/scattering more efficiently, since the all-to-all(v) type communication in the two-phase I/O algorithm is replaced by a number of independent all-gather(v) operations in the dynamic segmentation algorithm.

For very large collective operations, the dynamic segmentation algorithm is split into multiple cycles. This allows to keep the amount of temporary buffer required on the aggregator processes within constant, reasonable limits. Note, that depending on the offsets into the file a process might have to contribute different amounts of data to its aggregator in each cycle.

2.2 A Framework for Nonblocking Collective I/O Operations

A similar problem, the implementation of nonblocking collective operations, has been discussed in [4]. The framework for nonblocking collectives is implemented in the open-source library libNBC. We utilize and extend libNBC in conjunction with Open MPI's OMPIO framework [7] to handle nonblocking collective I/O operations. The central concept in libNBC's design is the collective operation schedule. During initialization of the operation, each process records its part of the collective operation in a local schedule. A schedule contains, among others, send and recv operations and a so called "barrier" which acts as local synchronization object. A barrier in a schedule has the semantics that all operations before the barrier have to be finished before any of the operations after the barrier can be started. The execution of a schedule is nonblocking and the state of the operation is simply kept as a pointer to a position in the schedule. With send, recv, and barrier, one can express any collective communication algorithm; see [4] for further details.

A major difference between collective communication and collective I/O operations stems from the fact, that each process is allowed to provide different volumes of data to a collective read or write operation, without having knowledge on the data volumes provided by other processes. This is not the case for collective communication operations, where either each process provides exactly the same amount of data (e.g. Bcast, Reduce, Allreduce, Gather, Scatter, Allgather, Alltoall etc.) or in case of the vector version of the operations a process knows the communication volumes of all processes communicating with him (Gatherv, Scatterv, Allgatherv, Alltoallv, Alltoallw). This information is however essential to determine how much data a process has to contribute within a cycle of the collective I/O operation.

Thus, the first step in most collective I/O algorithms is an All-gather(v) step which determines the overall amount of data each process is contributing along with the according offsets into the file. In the case of the dynamic segmentation algorithm, this communication operation is within each group of an aggregator. This allows every process to determine how much data it has to contribute in every cycle of the algorithm. For nonblocking operations, the challenge is, that upon calling `MPI_File_iwrite_all` the according All-gather(v) operations can not be finished, since this would result in a blocking communication operation when initiating the nonblocking write-all. This is however not possible, since it could lead to a deadlocks.

Thus, the solution developed here consists of a two-step approach: while initiating the nonblocking collective read/write operation, we generate first a schedule which executes the nonblocking All-gather(v) communication step [3] The last step of the All-gather(v) schedule will be executed when the All-gather(v) operation is finished, and creates a new schedule which executes the actual collective

[3] Note, that the operation is not exactly an `MPI_Allgatherv`, but consists of multiple All-gather(v) operations executed on disjoint groups of processes in the same communicator.

I/O operations. This second schedule contains the data gathering at the aggregator processes, the sorting based on the offsets into the file, and the asynchronous writing to the file.

Associated with that are two further challenges: first, no temporary buffers used within the collective I/O algorithm can be allocated upfront when posting the operation, since the overall amount of data and many of the according buffers are only known at the end of the All-gather(v) step. Therefore, we extended the set of operations supported by the progress engine of LibNBC in addition to nonblocking read and write by dynamic memory management functions, which allow to allocate and free buffers as part of the libNBC schedule. Second, due to the fact that the asynchronous I/O operation are implemented using `aio_read` and `aio_write` operations which have their own data structure to identify pending operations, the libNBC progress engine has been extended with the ability to progress multiple, different handles simultaneously, e.g. `MPI_Requests` for communication operations and the internal aio-handles for asynchronous I/O operations.

2.3 Schedule Caching

One of the distinct features of libNBC is its ability to cache a schedule of a collective operation. This allows to speed up execution of operations which are posted repeatedly by an application. I/O operations generally fit the repetitive pattern required for caching a schedule, e.g. in case an application writes periodic checkpoint files. In this scenario an application has two options. The first option is to append the most recent data that has to be written to the end of an existing file. The second option would use a different file for every checkpoint. Both approaches post unique challenges for caching a schedule.

For the first option, the challenge comes from the fact that every collective I/O operation which appends data to an already existing file will lead to new offset values into the file. Moreover, the MPI standard also allows for a process to mix individual and collective I/O calls, which makes predicting the current position of the file pointer of a process impossible. Since the order in which data has to be written to the file depends on the file view and the current position of the individual file pointer, the actual amount of data that a process has to contribute to a particular cycle of the collective I/O is not necessarily repetitive, even if the arguments passed to the MPI function are identical to the previous instance. Thus, caching the schedule would not help in this scenario.

The second scenario where a separate file is used for every checkpoint is equally challenging, due to the fact the schedules would be cached on a per file handle basis. This is in equivalence to the collective communication operations, where the caching is being done on a per communicator basis, although the MPI specification does not providing attribute caching functions on files as of today. Transferring a schedule from one file handle to another file handle can theoretically be done, the challenge being however how to keep a file handle around once a file has been closed, without creating an unnecessary memory overhead.

3 Performance Evaluation

In the following section we evaluate the impact of the nonblocking collective I/O operations. We first describe the execution environment followed by the results obtained with a micro-benchmark and a parallel image processing application.

3.1 Experimental Setting

The system used in these tests is the *crill-cluster* at the University of Houston, which consists of 16 nodes with four 12-core AMD Opteron (Magny Cours) processor cores each (48 cores per node, 768 cores total), 64 GB of main memory and two dual-port InfiniBand HCAs per node. The parallel file system used is PVFS2 with 16 I/O servers and a stripe size of 64 KB. The file system is mounted onto the compute nodes over the Gigabit Ethernet network interconnect of the cluster. The current implementation of nonblocking collective operations is tied to Open MPI and its new parallel I/O framework (OMPIO), mostly for retrieving and maintaining file handle related aspects and for decoding derived data types and the file view. The version of Open MPI executed is equivalent to the Open MPI trunk revision 24640. In the following analysis we focus for the sake of simplicity on write operations.

The first test executed is using the Latency-IO micro-benchmark developed as part of the latency test suite [8]. Initially, we compare the performance obtained with the blocking version of the dynamic segmentation algorithm vs. a sequence of `NBC_File_iwrite_all` followed by `NBC_Wait`. Table 1 presents the bandwidth achieved in both scenarios for 64 and 128 MPI processes when using 32 aggregator processes and a 4 MB cycle buffer size. The overall file size written were 63 GB and 125 GB respectively (1000MB per process). All tests have been executed three times, and we present the average bandwidth obtained over all three runs. The results indicate a small overhead for the 64 processes test case of the nonblocking implementation, which achieved 94% of the bandwidth obtained with the blocking version. For 128 processes the nonblocking version slightly outperformed the blocking version, which we attribute however to measurement jitter. All-in-all, the conclusion drawn from this analysis is that nonblocking implementation does not impose a significant, fundamental overhead compared to the blocking version.

Table 1. Performance comparison of blocking vs. nonblocking collective I/O algorithm.

No. of processes	Blocking Bandwidth	Nonblocking Bandwidth
64	703 MB/s	660 MB/s
128	574 MB/s	577 MB/s

In the second test we evaluate the ability to overlap collective I/O operations with compute operations. For this, the same benchmark is executing a compute function after posting the nonblocking collective write operation. The compute operation is configured to take the equal amount of time as the I/O operation. Thus, we expect to observe an overall execution time equal or larger than the time required to perform the I/O operation only for the according scenario, with the upper bound being twice the amount of time required for the same test without the compute operation in case I/O and computation cannot be overlapped. Table 2 presents the results achieved for the same test cases as outlined above, the first column being the time spent in the I/O test without overlap, the second column representing the time spent in writing the same amount of data and performing an equally expensive compute operation, and the third column showing the time spent in the compute operation for the overlap test.

Table 2. Evaluating the overlap potential of nonblocking collective I/O operations.

No. of processes	I/O only time	Overlapping time	Time spent in computation
64	85.69 sec	85.80 sec	85.69 sec
128	205.39 sec	205.91 sec	205.39 sec

The results in this section indicate the ability to entirely hide the I/O operation under optimal circumstances. These optimal circumstances are represented by the ability of libNBC to progress the operation either through a progress thread or through inserting regularly `NBC_Test` function calls into the compute operation. Within the context of this analysis, we choose the second approach. Moreover, we also identified that the frequency and number of calls to `NBC_Test` have a tremendous influence on the overlap performance: calling it too often will introduce an additional overhead, if there are to few calls to this function, the library will not be able to progress the function. In our experimental results we identified the time required to execute one cycle in the dynamic segmentation algorithm as the optimal interval between two subsequent calls to `NBC_Test`.

3.2 An Application Scenario

Further tests have been executed with a parallel image processing application. This application is used to analyze smear sample from fine needle aspiration cytology, with the overall goal being to assist medical doctors in identifying cancer cells [9]. The challenge imposed by this application is due to the high resolution of the microscopes and the fact that images are captured at various wave-length to identify different chemical properties of the cells. For a $1cm \times 1cm$ sample with 31 spectral channels the image can contain overall up to 50GB of raw data. The MPI version of the code has furthermore the option to write the texture data into output files to facilitate future processing steps in realizing a complete computer aided diagnosis (CAD) solution. This makes the application compute and I/O intensive.

For the following tests, we focus on the code section which writes the texture data into files. This code sequence contains a loop in which texture data for each of the twelve Gabor filters is calculated and then written to a separate file. The computational part within this loop consists of two parallel fast-fourier transforms (FFTs), which are implemented using the FFTW library [10] version 2.1.5, and a convolution operation. For the version using the non-blocking collective I/O functions, writing the texture data in one iteration is overlapped with the execution of the FFTs and the convolution of the next iteration. Progress of the non-blocking collective I/O operation is implemented in two ways. The first one uses `NBC_Test` function calls in-between each FFT and the convolution operation. The second code version uses a patched version of the FFTW library which contains further function calls to `NBC_Test`. Note, that the initial reading of the image and final writing of the cluster assignments have not been modified and still use the blocking collective MPI I/O version.

For evaluation purposes we used two separate images. The first image has 8192 × 8192 pixels and 21 spectral channels, writing 12 times 256 MB of texture data (3 GB total) . The second image has 12281 × 12281 pixels and also 21 spectral channels, writing 12 times 576 MB of data (6.75 GB total). Tests have been executed with 64 and 96 processes on the same cluster and file system as in the previous section. Figure 1 show the times spent in I/O operations for each test case. We present again the average obtained over three separate runs. Note, that we ensured that both blocking and non-blocking collective I/O operation use the same algorithm, with the same number of aggregator processes and the same cycle buffer size.

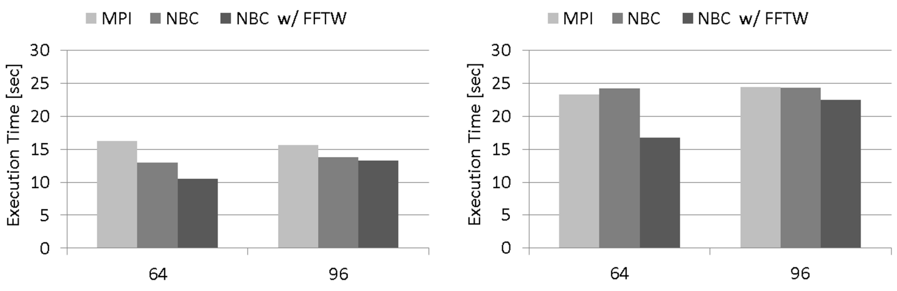

Fig. 1. Comparison of I/O times for 8k × 8k image (left) and 12k × 12k image (right) for 64 and 96 MPI processes.

The results indicate that the version of the code which uses the FFTW library as a 'black box', i.e. without any `NBC_Test` function calls inserted, offers only little benefit compared to the original version of the code which uses blocking, collective MPI I/O operation. The main problem is the limited ability to progress the non-blocking operations without a progress thread and with a very small number of calls to `NBC_Test`. On the other hand, using the patched version of the FFTW library ensures more progress and demonstrates significant benefits of the non-blocking collective I/O operations. The benefit is more obvious for the 64 processes test cases compared to the 96 processes test cases due to the increased execution time of the FFTs and the convolution for the 64 process test cases, which offer therefore more potential for overlapping computation and I/O operations. Hiding the entire costs of the I/O operations for a real application is however very difficult, since i) the application has to have compute intensive sections that can be used for overlapping computation and I/O operations and ii) the timespan between two subsequent calls to `NBC_Test` can not be controlled in the similar manner as for the micro-benchmark. Nevertheless, with some efforts we were able to reduce the time spent in I/O operation by up to 35% – which can be highly significant for large scale applications.

4 Discussion

To mitigate potential performance bottlenecks, MPI added support for nonblocking file routines. However, collective MPI file operations can only be expressed with the limited split collective interface. The main limitations are that (1) there must only be a single split collective active on a file handle at any time and (2) no other collective file I/O operations can be issued on a file handle when a split collective is active. The first limitation prevents optimization techniques such as pipelined communications for communication/communication overlap [11] and the second limitation reduces programmability. The MPI-2.2 standard also allows to perform a global synchronization in the begin call of a split collective. This limits certain usage patterns.

Our nonblocking collective I/O framework allows to offer two additional features: (1) explicit progress and (2) multiple outstanding operations.

Thus, we propose to extend the MPI standard similar to nonblocking collective operations, i.e., to add immediate versions of all split collective operations, e.g., MPI_File_iread_all(..., MPI_Request req) and adding a request as last parameter. For file operations, the file pointer is advanced within the immediate function call, so that following calls operate on the right offset. We omit a list of all functions for space reasons.

The new functions can be used like nonblocking point-to-point and collective operations and the returned requests can be tested and waited on for completion with the usual functions (e.g., MPI_Test). Implicit progress can be problematic under certain circumstances while explicit progress puts a higher burden on the user [12]. Our interface proposal allows the implementation to offer both choices to the user. In addition, having multiple outstanding operations allows to employ pipelining techniques for overlapping communication and computation.

5 Conclusion

In this paper we discussed the challenges associated with non-blocking collective I/O operations. We present a framework which provides non-blocking versions of the collective read and write operations by extending the libNBC library. The performance of write operation has been evaluated using a micro benchmark and parallel image processing application. The results indicate the potential to actually overlap computation and I/O operations using these functions. However, the main challenge is how to ensure progress of the non-blocking collective I/O operations in the absence of a progress thread. The currently ongoing work includes multiple domains. First, we plan to extend the analysis to collective read operations. Second, we plan to perform a similar set of analysis as shown in this paper on different file systems, specifically on a large scale Lustre installation.

References

1. Brightwell, R., Underwood, K.D.: An analysis of the impact of MPI overlap and independent progress. In: ICS '04: Proceedings of the 18th annual international conference on Supercomputing, New York, NY, USA, ACM Press (2004) 298–305
2. Baude, F., Caromel, D., Furmento, N., Sagnol, D.: Optimizing metacomputing with communication-computation overlap. In: PaCT '01: Proceedings of the 6th International Conference on Parallel Computing Technologies, London, UK, Springer-Verlag (2001) 190–204
3. Hoefler, T., Gottschling, P., Lumsdaine, A., Rehm, W.: Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations. Elsevier Journal of Parallel Computing (PARCO) **33**(9) (9 2007) 624–633
4. Hoefler, T., Lumsdaine, A., Rehm, W.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In: Proc. of the 2007 Intl. Conf. on High Perf. Comp., Networking, Storage and Analysis, SC07, IEEE Computer Society/ACM (Nov. 2007)
5. Kothe, D., Kendall, R.: Computational science requirements for leadership computing. Technical report, ORNL/TM-2007/44 (2007)
6. Chaarawi, M., Chandok, S., Gabriel, E.: Performance Evaluation of Collective Write Algorithms in MPI I/O. In: Proceedings of the International Conference on Computational Science (ICCS). Volume 5544., Baton Rouge, USA (2009) 185–194
7. Chaarawi, M.: Optimizing Parallel I/O Operations for High Performance Computing. PhD thesis, Department of Computer Science, University of Houston (2011)
8. 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) (2005) 67–79
9. 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) (2009) 231–240
10. Frigo, M., Johnson, S.G.: The Design and Implementation of FFTW3. Proceedings of IEEE **93**(2) (2005) 216–231 Special issue on “Program Generation, Optimization, and Platform Adaptation”.
11. Bell, C., Bonachea, D., Cote, Y., Duell, J., Hargrove, P., Husbands, P., Iancu, C., Welcome, M., Yelick, K.: An evaluation of current high-performance networks. In: Proc. of the 17th Int. Symp. on Par. and Distr. Proc. (2003) 28.1
12. Hoefler, T., Lumsdaine, A.: Message Progression in Parallel Computing - To Thread or not to Thread? In: Proceedings of the 2008 IEEE International Conference on Cluster Computing, IEEE Computer Society (Oct. 2008)