Extending the MPI-2 Generalized Request Interface

Robert Latham, William Gropp, Robert Ross, and Rajeev Thakur

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{robl,gropp,rross,thakur}@mcs.anl.gov

Abstract. The MPI-2 standard added a new feature to MPI called *generalized requests*. Generalized requests allow users to add new nonblocking operations to MPI while still using many pieces of MPI infrastructure such as request objects and the progress notification routines (`MPI_Test`, `MPI_Wait`). The generalized request design as it stands, however, has deficiencies regarding typical use cases. These deficiencies are particularly evident in environments that do not support threads or signals, such as the leading petascale systems (IBM Blue Gene/L, Cray XT3 and XT4). This paper examines these shortcomings, proposes extensions to the interface to overcome them, and presents implementation results.

1 Introduction

In a message-passing environment, a nonblocking communication model often makes a great deal of sense. Implementations have flexibility in optimizing communication progress; and, should asynchronous facilities exist, computation can overlap with the nonblocking routines.

MPI provides a host of nonblocking routines for independent communication, and MPI-2 added nonblocking routines for file I/O. When callers post nonblocking MPI routines, they receive an MPI request object, from which the state of the nonblocking operation can be determined. Generalized requests, added as part of the MPI-2 standard [1], provide a way for users to define new nonblocking operations. Callers of these user-defined functions receive a familiar request object and can use the same test and wait functions as a native request object. A single interface provides a means to test communication, I/O, and user-defined nonblocking operations.

Generalized requests, unfortunately, are difficult to use in some environments. Our experience with generalized requests comes from using them to implement nonblocking I/O in the widely available ROMIO MPI-IO implementation [2].

In the absence of generalized requests, ROMIO defines its own ROMIO-specific request objects to keep track of state in its nonblocking MPI-IO routines. With these custom objects, ROMIO does not need to know the internals of a given MPI implementation. The usual MPI request processing functions, however, cannot operate on ROMIO's custom objects, so ROMIO must also export its own version of the MPI test and wait routines (`MPIO_TEST`, `MPIO_WAIT`, etc.). Moreover, these custom objects and routines are not standards-conformant.

With the implementation of MPI-2 on many more platforms now, ROMIO can use generalized requests instead of custom requests and functions. Generalized requests allow ROMIO to adhere to the MPI standard and provide fewer surprises to users of ROMIO's nonblocking routines. Unfortunately, the current definition of generalized requests makes it difficult (or in some instances impossible) to implement truly nonblocking I/O. To carry out asynchronous I/O with generalized requests, ROMIO must spawn a thread or set up a signal handler that can then test and indicate an asynchronous operation has completed. Since threads or signal handlers cannot be used in all environments, however, an alternative approach is desireable.

In this work we examine the shortcomings of the existing generalized request system, propose improvements to the generalized request design, and discuss the benefits these improvements afford.

2 MPI Requests vs. Generalized Requests

The MPI standard addresses the issue of progress for nonblocking operations in Section 3.7.4 of [3] and Section 6.7.2 of [1]. MPI implementations have some flexibility in how they interpret these two sections. The choice of a weak interpretation (progress occurs only during MPI calls) or a strict interpretation (progress can occur at any point in time) has a measurable impact on performance, particularly when the choice of progress model affects the amount of overlap between computation and communication [4].

The MPI-2 standard addresses the issue of progress for generalized requests by defining a super-strict model in which no progress can be made during an MPI call. When creating generalized requests, users must ensure all progress occurs outside of the context of MPI.

Here's how one uses generalized requests to implement a new nonblocking operation. The new operation calls `MPI_GREQUEST_START` to get an MPI request object. After the operation has finished its task, a call to `MPI_GREQUEST_COMPLETE` marks the request as done. However, the completion call will never be invoked by any MPI routine. All progress for a generalized request must be made outside of the MPI context. Typically a thread or a signal handler provides the means to make progress.

When we used generalized requests to implement nonblocking I/O routines in ROMIO, we found this super-strict progress model limiting. In many situations we do not want to or are unable to spawn a thread. Moreover, we recognized that we could effectively apply generalized requests to more situations if we could relax the progress model. We could also achieve a greater degree of overlap between computation and file I/O.

3 Asynchronous File I/O

To illustrate the difficulties using generalized requests with ROMIO, we use asynchronous file I/O as an example. The most common model today is POSIX AIO [5], but Win32 Asynchronous I/O [6] and the PVFS nonblocking I/O interfaces [7] share a common completion model with POSIX AIO.

Table 1. Typical functions for several AIO interfaces

	POSIX AIO	Win32 AIO	PVFS v2
Initiate	`aio_write`	`WriteFileEx`	`PVFS_isys_write`
Test	`aio_error`	`SleepEx`	`PVFS_sys_test`
Wait	`aio_suspend`	`WaitForSingleObjectEx`	`PVFS_sys_wait`
Wait (all)	`aio_suspend`	`WaitForMultipleObjectsEx`	`PVFS_sys_waitall`

In Table 1 we show a few of the functions found in these three AIO interfaces. The completion model looks much like that of MPI and involves two steps: post an I/O request and then, at some point, test or wait for completion of that request. After posting I/O operations, a program can perform other work while the operating system asynchronously makes progress on the I/O request. The operating system has the potential to make progress in the background though all work could occur in either the initiation or the test/wait completion call. This model lends itself well to programs with a single execution thread.

We note that POSIX AIO does define an alternative mechanism to indicate completion via real-time signals. This signal-handler method fits well only with POSIX AIO, however. Neither the other AIO interfaces nor other situations where work is occurring asynchronously can make effective use of signals, and so we will not consider them further.

4 Generalized Request Deficiencies

ROMIO, one of the earliest and most widely deployed MPI-IO implementations, has portability as a major design goal. ROMIO strives to work with any MPI implementation and on all platforms. Because of this portability requirement, ROMIO cannot always use threads. While POSIX threads are available on many platforms, they are notably not available on the Blue Gene/L or the Cray XT3 and XT4 machines, for example.

As discussed in Section 2, the requirement of a super-strict progress model for generalized requests makes it difficult to create new nonblocking operations without spawning a thread. Under this super-strict progress model, common asynchronous I/O interfaces have no good thread-free mechanism by which to invoke their completion routine.

Consider the code fragment in Figure 1 implementing `MPI_File_iwrite`. If the implementation wishes to avoid spawning a thread, it must block: there is no

```
MPI_File_iwrite(..., *request) {
    struct aiocb write_cb = { ... }

    aio_write(&write_cb)
    MPI_Grequest_start(..., request)
    aio_suspend(write_cb, 1, MAX_INT)
    MPI_Grequest_complete(request)
    return;
}
```

Fig. 1. A thread-free way to use generalized requests. In the current generalized request design, the post and the test for completion of an AIO operation and the call to `MPI_GREQUEST_COMPLETE` must all happen before the routine returns. Future `MPI_Wait` routines will return immediately as the request is already completed.

other way to invoke `aio_suspend` and `MPI_Grequest_complete` yet, a thread or signal handler is unnecessary in the file AIO case: the operating system takes care of making progress. This pseudocode is not a contrived example. It is essentially the way ROMIO must currently use generalized requests. The current generalized request design needs a way for the MPI test and wait routines to call a function that can determine completion of such AIO requests.

Other Interfaces In addition to AIO, other interfaces might be able to make use of generalized requests were it not for portability issues. Coupled codes, such as those used in weather forecasting, need a mechanism to poll for completion of various model components. This mechanism could use generalized requests to initiate execution and test for completeness. Nonblocking collective communication lends itself well to generalized requests as well, especially on architectures with hardware-assisted collectives. These interfaces, however, must accommodate the lack of thread support on Blue Gene/L and Cray XT series machines and cannot use generalized requests in their current form if they wish to remain portable.

In this paper we suggest improvements to the generalized request interface. We use asynchronous I/O as an illustrative example. However, the benefits would apply to many situations such as those given above where the operating environment can do work on behalf of the process.

5 Improving the Generalized Request Interface

As we have shown, AIO libraries need additional function calls to determine the state of a pending operation. We can accommodate this requirement by extending the existing generalized request functions. We propose an `MPIX_GREQUEST_START` function similar to `MPI_GREQUEST_START`, but taking an additional function pointer (`poll_fn`) that allows the MPI implementation to make progress on pending generalized requests. We give the prototype for this function in Figure 3 in the Appendix.

When the MPI implementation tests or waits for completion of a generalized request, the poll function will provide a hook for the appropriate AIO completion function. It may be helpful to illustrate how we imagine an MPI implementation might make use of this extension for the test and wait routines ({MPI_TEST,MPI_WAIT}{,ALL,ANY,SOME}). All cases begin by calling the request's user-provided poll_fn. For the wait routines, the program continues to call poll_fn until either at least one request completes (wait, waitany, waitsome) or all requests complete (wait, waitall).

An obvious defect of this approach is that the MPI_WAIT{ANY/SOME/ALL} and MPI_WAIT functions must poll (i.e., busy wait). The problem is that we do not have a single wait function that we can invoke. In Section 7 we provide a partial solution to this problem.

6 Results

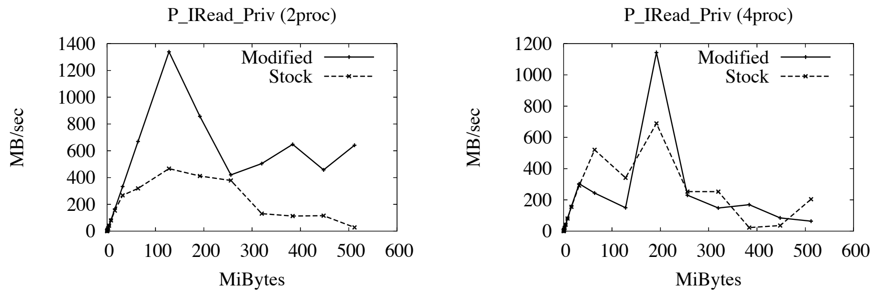
We implemented MPIX_GREQUEST_START in an experimental version of MPICH2 [8] and modified ROMIO's nonblocking operations to take advantage of this extension. Without this extension, ROMIO still uses generalized requests, but does so by carrying out the blocking version of the I/O routine before the nonblocking routine returns. With the extension, ROMIO is able to initiate an asynchronous I/O operation, use generalized requests to maintain state of that operation, and count on our modified MPICH2 to invoke the completion routine for that asynchronous I/O operation during test or wait. This whole procedure can be done without any threads in ROMIO.

Quantifying performance of a nonblocking file operation is not straightforward. Ideally, both the I/O and some unit of work execute concurrently and with no performance degradation. Capturing both performance and this measure of "overlap" can be tricky.

Nonblocking writes introduce an additional consideration when measuring performance. Write performance has two factors: when the operating system says the write is finished, and when the write has been flushed from buffers to disk. Benchmark results for both old and new MPICH2 implementations look quite similar, since MPI_FILE_SYNC dominates the time for both implementations. We will therefore focus on performance of the more straightforward read case.

We used the Intel®MPI Benchmarks package [9]. Our results are for "optional" mode, only because we increased the maximum message size from 16 MB to 512 MB in order to see how performance varied across a wider scale of I/O sizes. Our test platform is a dual dual-core Opteron (4 cores total), writing to a local software RAID-0 device.

We depict the results of the P_IreadPriv benchmark in Figure 2(a) (2 processes) and Figure 2(b) (4 processes). This MPI-IO test measures nonblocking I/O performance when each process reads data from its own file (i.e., one file per



(a) 2 Processes

(b) 4 Processes

Fig. 2. `P_IRead_Priv` test with two MPI processes. "Stock" depicts standard (blocking) generalized requests. "Modified" shows performance with our improvements.

process) while a synthetic CPU-heavy workload runs concurrently. The benchmark varies the size of the nonblocking I/O requests while keeping the CPU workload fixed (0.1 seconds). [1] When comparing two MPI implementations, we found computing the effective bandwidth at a given request size yielded a useful metric for evaluating relative overlap. "Effective bandwidth" in this case means the size of a request divided by the (inclusive) time taken to post the request, run the CPU-heavy workload, and detect completion of that request. Higher effective bandwidth means a higher degree of overlap between I/O and computation.

Both graphs have three regions of interest. For small I/O sizes, true nonblocking operations give little if any benefit. As the amount of I/O increases, however, effective bandwidth increases when the MPI implementation can carry out I/O asynchronously. Asynchronous I/O benefits most — nearly three times at peak — if there are spare CPUs, but even in the fully subscribed case we see almost a doubling of peak performance. At large enough request sizes, the amount of I/O dwarfs the fixed amount of computation, limiting the opportunities for I/O and computation to overlap. Even so, asynchronous I/O on this platform appears to benefit significantly from the unused cores in the two-process case. We suspect polling in the MPI implementation might have an impact on I/O performance. In Section 7 we propose a refinement that can limit busy-waiting.

We note that the work described in this paper *enables* asynchronous I/O. Whether asynchronous I/O is beneficial or not depends on many factors, such as application workload and the quality of a system's AIO libraries. Finding the ideal balance in overlapping I/O and computation is a fascinating area of research but is beyond the scope of this paper.

[1] This benchmark computes an "overlap" factor, but the computation in this case gave odd and inconsistent results.

7 Further Improvements: Creating a Generalized Request Class

With this simple extension to generalized requests we have already achieved our main goals: ROMIO has a hook by which it can determine the status of a pending AIO routine, and can do so without spawning a thread. If we observe that generalized requests are created with a specific task in mind, we can further refine this design.

In the AIO case, all callers are going to use the same test and wait routines. In POSIX AIO, for example, a nonblocking test for completion of an I/O operation (read or write) can be carried out with a call to `aio_error`, looking for `EINPROGRESS`. AIO libraries commonly provide routines to test for completion of multiple AIO operations. The libraries also have a routine to block until completion of an operation, corresponding to the `MPI_WAIT` family.

We can give implementations more room for optimization if we introduce the concept of a generalized request class. `MPIX_GREQUEST_CLASS_CREATE` would take the generalized request query, free, and cancel function pointers already defined in MPI-2 along with our proposed poll function pointer. The routine would also take a new "wait" function pointer. Initiating a generalized request then reduces to instantiating a request of the specified class via a call to `MPIX_GREQUEST_CLASS_ALLOCATE`. Prototypes for these routines are given in Figure 4 in the Appendix.

At first glance this may appear to be just syntax: why all this effort just to save passing two pointers? One answer is that in ROMIO's use of generalized requests, the query, free, and cancel methods are reused multiple times; Hence, a generalized request class would slightly reduce duplicated code.

A more compelling answer lies in examining how to deal with polling. By creating a class of generalized requests, we give implementations a chance to optimize the polling strategy and minimize the amount of time consuming CPU while waiting for request completion.

Refer back to Figure 2(b), where the unmodified, blocking MPICH2 outperforms the modified MPICH2 at the largest I/O request size. At this point, I/O takes much longer to compute than the computation. All available CPUs are executing the benchmark and polling repeatedly inside `MPI_Waitall` until the I/O completes. The high CPU utilization, aside from doing no useful work, also appears to be interfering with the I/O transfer.

Our proposed generalized request class adds two features that together solve the problem of needlessly consuming CPU in a tight testing loop. First, we introduce `wait_fn`, a hook for a blocking function that can wait until completion of one or more requests. If multiple generalized requests are outstanding, an implementation cannot simply call a user-provided wait routine (created with a specific generalized request in mind) on all of them. If all the outstanding requests are of the same generalized request class, however, the implementation might be able to pass several or even all requests to a user-provided wait routine, which in turn could complete multiple nonblocking operations. By avoiding repeated polling and aggregating multiple requests, our generalized request class thus can make processing user-defined nonblocking operations more efficient, particularly in those MPI functions such as MPI_WAITALL that take multiple requests.

Generalized request classes also open the door for the MPI implementation to learn more about the behavior of these user-provided operations, and potentially adapt. We imagine that an MPI implementation could keep timing information or other statistics about a class of operations and adjust timeouts or otherwise inform decisions in the same way Worringen automatically adjusts MPI-IO hints in [10]. Implementations cannot collect such statistics without a request class, since those implementations can glean meaningful information only by looking at generalized requests implementing a specific feature.

8 Conclusions

Generalized requests in their current form do much to simplify the process of creating user-provided nonblocking operations. By tying into an implementation's request infrastructure, users avoid reimplementing request bookkeeping. Unfortunately, while generalized requests look in many ways like first-class MPI request objects, the super-strict progress model hinders their usefulness. Whereas an MPI implementation is free to make progress for a nonblocking operation in the test or wait routine, generalized requests are unable to make progress in this way. This deficiency manifests itself most when interacting with common asynchronous I/O models, but it is also an issue when offloading other system resources.

We have presented a basic extension to the generalized request design as well as a more sophisticated class-based design. In reviewing the MPI Forum's mailing list discussions about generalized requests, we found early proposals advocating an approach similar to ours. A decade of implementation experience and the maturity of AIO libraries show that these early proposals had merit that perhaps went unrecognized at the time. For example, at that time it was thought that using threads would solve the progress problem, but today we are faced with machines for which threads are not an option.

Our extensions would greatly simplify the implementation of nonblocking I/O operations in ROMIO or any other library trying to extend MPI with custom nonblocking operations. Class-based approaches to making progress on operations would alleviate some of the performance concerns of using generalized requests.

Unlike many MPI-2 features, generalized requests have seen neither widespread adoption nor much research interest. We feel the extensions proposed in this paper would make generalized requests more attractive for library writers and for those attempting to use MPI for system software, in addition to opening the door for additional research efforts.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing

Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357.

Appendix: Function Prototypes

In this paper we have proposed several new MPI routines. Figure 3 and Figure 4 give the C prototypes for these routines.

```
int MPIX_Grequest_start(
                MPI_Grequest_query_function *query_fn,
                MPI_Grequest_free_function *free_fn,
                MPI_Grequest_cancel_function *cancel_fn,
                MPIX_Grequest_poll_function *poll_fn,
                void *extra_state,
                MPI_Request *request)
typedef int MPIX_Grequest_poll_fn(
                void *extra_state,
                MPI_Status *status);
```

Fig. 3. Prototypes for generalized request poll extension

```
typedef int MPIX_Grequest_wait_fn(
                int count,
                void *array_of_states,
                double timeout,
                MPI_Status *status);
int MPIX_Grequest_class_create(
                MPI_Grequest_query_function *query_fn,
                MPI_Grequest_free_function *free_fn,
                MPI_Grequest_cancel_function,
                MPIX_Grequest_poll_fn,
                MPIX_Grequest_wait_fn,
                MPIX_Request_class *greq_class);
int MPIX_Grequest_class_allocate(
                MPIX_Request_class greq_class,
                void *extra_state
                MPI_Request *request)
```

Fig. 4. Prototypes for generalized request classes and blocking wait function.

References

1. The MPI Forum: MPI-2: Extensions to the message-passing interface. The MPI Forum (July 1997)
2. Thakur, R., Gropp, W., Lusk, E.: On implementing MPI-IO portably and with high performance. In: Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems. (May 1999) 23–32
3. Message Passing Interface Forum: MPI: A message-passing interface standard. Technical report (1994)
4. Brightwell, R., Riesen, R., Underwood, K.: Analyzing the impact of overlap, offload, and independent progress for MPI. The International Journal of High-Performance Computing Applications **19**(2) (Summer 2005) 103–117
5. IEEE/ANSI Std. 1003.1: Single UNIX specification, version 3 (2004 edition)
6. Microsoft corporation: Microsoft Developer Network Online Documentation, http://msdn.microsoft.com. (accessed 2007)
7. PVFS development team: The PVFS parallel file system. `http://www.pvfs.org/` (accessed 2007)
8. MPICH2 development team: MPICH2. `http://www.mcs.anl.gov/mpi/mpich2`
9. Intel GmbH: Intel MPI benchmarks. http://www.intel.com
10. Worringen, J.: Self-adaptive hints for collective I/O. In: Proceedings of the 13th European PVM/MPI User's Group Meeting, Bonn, Germany (September 2006) 202–211