Enabling Non-Destructive Testing of the Statuses of Multiple Requests

William R. Williams
Technische Universität Dresden
Dresden, Germany
william.williams@mailbox.tu-dresden.de

Marc-André Hermanns
RWTH Aachen University
Aachen, Germany
hermanns@itc.rwth-aachen.de

Joachim Jenke
RWTH Aachen University
Aachen, Germany
jenke@itc.rwth-aachen.de

ABSTRACT

We propose extending the MPI interface to allow the non-destructive test of multiple statuses in a manner that is guaranteed to mimic the progress and fairness behavior of the corresponding `MPI_WaitXXX` and `MPI_TestXXX` functions: `MPI_Request_get_status_all`, `MPI_Request_get_status_some`, and `MPI_Request_get_status_any`. We show how this can simplify tool code and allow safe layering of tools that wish to wrap the wait and test families of MPI functions.

ACM Reference Format:
William R. Williams, Marc-André Hermanns, and Joachim Jenke. 2023. Enabling Non-Destructive Testing of the Statuses of Multiple Requests. In *Proceedings of EuroMPI 2023 (EuroMPI '23).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The existence of the PMPI interface since the very beginning of the Message Passing Interface (MPI) [2] has led to an abundance of performance and correctness tools being available to developers using MPI to aid in their development process. This high availability of tools has most likely impacted the wide acceptance of MPI. Such tools strive to perturb the original application behavior as little as possible, meaning that performance tools would strive to keep the performance overhead of measurements to a minimum. In contrast, correctness tools strive to keep as much of the original call sequences within the MPI library. As part of their measurements, tools often track non-blocking communication requests to retain a consistent view of ongoing operations for the user. As part of such tracking, these tools may need to check the state of an active request and its corresponding status non-destructively. The use of non-destructive tests is often (but not exclusively) connected to identifying the one or more requests the tool must act on (e.g., look up tracking information) while using the active request handle as a lookup key. While as of version 4.0 of the MPI standard [2], MPI allows users to perform destructive wait or test operations on multiple pending requests; it does not allow similar non-destructive status-checking operations except on individual requests.

EuroMPI '23, September 11-13, 2023, Bristol, UK

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
int MPI_Testsome(int incount, MPI_Request req[], int*
↪  outcount, int* indices, MPI_Status* statuses[])
{
  for(int i = 0; i < incount; i++) {
    saved_requests[i] = tool_request_data(req[i]);
  }
  int ret = PMPI_Testsome(incount, req, outcount,
↪  indices, statuses);
  for(int i=0; i < *outcount; i++) {
    orig_req = saved_requests[indices[i]];
    status = statuses[i];
    process_deactivated_request(orig_req, status);
  }
  return ret;
}
```

Figure 1: Example of saving requests pre-call when wrapping `MPI_Testsome` without `MPI_Request_get_statusXXX` available.

We, therefore, propose new API functions that allow this non-destructive status checking: `MPI_Request_get_status_all`, `MPI_Request_get_status_any`, and `MPI_Request_get_status_some`.

2 MOTIVATION

The absence of functions that allowed the non-destructive testing of status for multiple requests causes several problems, in particular for developers who wish to use the PMPI interface in order to wrap *destructive* multiple-request completion functions.

At the moment tools like Score-P [4] need to save all requests prior to calling the PMPI layer to process the request after completion as shown in Figure 1.

P^nMPI [5] has long been a tool to assist in correctly implementing the nested interception of MPI functions. A stack of tools calling, for example, `MPI_Testany`, will call each other in a well-defined order, eventually calling some set of PMPI functions that should be equivalent to `MPI_Testany`–possibly `MPI_Testany` itself, possibly an equivalent sequence of `MPI_Test` and/or `MPI_Request_get_status` calls. Nevertheless, each tool must take care to preserve any request that it might be interested in post-call.

The previous case shows a more general problem: within tool code wrapping a `MPI_WaitXXX` or `MPI_TestXXX` call, it is guaranteed that if the *status* for a request is of interest and thus valid post-PMPI-call, the *request* itself has been invalidated by the call.

This, in turn, complicates any tool which tracks the cancellation of requests or any other information that is dependent on both the request object and the status object. In general, it is impossible for both the statuses resulting from MPI_WaitXXX or MPI_TestXXX and the requests corresponding to those statuses to be observed simultaneously.

One might *emulate* a MPI_WaitXXX or MPI_TestXXX call via iteration over the array of requests with MPI_Request_get_status, followed by appropriate calls to MPI_Wait or MPI_Test on individual completed requests in that array.

However, a tool performing this emulation does not necessarily provide the same progress or, more importantly, in the case of MPI_Waitsome/MPI_Testsome, fairness guarantees as the original call. This means that introducing a tool that performs such emulation can alter the behavior of the original program in unforeseen ways. Such imprecision is particularly undesirable in correctness tools, such as MUST [3]. Finally, the absence of these three proposed functions is an apparent asymmetry in the standard, which this proposal corrects.

3 PROPOSED SOLUTION

We propose the addition of MPI_Request_get_status_all, MPI_Request_get_status_any, and MPI_Request_get_status_some to the MPI 4.1 Standard. These new functions allow for the non-destructive examination of multiple statuses, and allow a tool to ensure that the implementation is allowed the freedom to perform internal optimization in the same manner that it may for the corresponding wait and test functions. Further, by replacing the external iteration of requests with internal iteration, tool code becomes more straightforward and easier to understand while retaining the original progress behavior of the application with the tool. Also, describing the form and effects of these iterations in the MPI standard help avoid the case where a well-meaning but misguided tool might attempt to write a wrapper that preserves both the behavior of the unwrapped call and the status of all requests completed by the call, but instead substantially alters the behavior of the call (and thus the program) in an unexpected way.

3.1 MPI_Request_get_status_any

We propose that MPI_Request_get_status_any should be equivalent to calling MPI_Request_get_status on all *active* requests in the input array in an arbitrary order consistent with MPI_Testany and MPI_Waitany. Inactive requests, including null requests, would trivially return $flag = true$ from MPI_Request_get_status, and are skipped in this processing. This ensures that MPI_Request_get_status_any will always either return $flag = true$ or attempt progress, possibly both. This is symmetric with the guarantees provided by MPI_Waitany and MPI_Testany, while accounting for the possibility that MPI_Request_get_status_any may be called repeatedly on the same array of requests, potentially completing but not freeing any of the active requests in the input array.

Note that MPI_Waitany and MPI_Testany, because they complete *and* free a request if they return $flag = true$, are guaranteed *not* to be called repeatedly with the same input data: any call that returns $flag = true$ will alter the input array irrevocably.

3.2 MPI_Request_get_status_all

We propose that MPI_Request_get_status_all should work in an obvious manner: it will either attempt progress on at least one of the requests in the input array, or it will return $flag = true$ as a result of all input requests being completed. Thus, repeated calls to MPI_Request_get_status_all will eventually return $flag = true$ provided that matching sends or receives, as appropriate for the requests in the input array, are eventually posted. Congruent with the guarantees given by MPI_Request_get_status_any, the input array itself will not be altered, i.e., none of the requests will be freed in the process. If all input requests are inactive or null, as with MPI_Waitall and MPI_Testall, MPI_Request_get_status_all will return immediately with $flag = true$.

3.3 MPI_Request_get_status_some

Finally, we propose that MPI_Request_get_status_some should present, conceptually, the same fairness guarantees as its counterparts MPI_Waitsome and MPI_Testsome, namely the following:

> If a request for a receive repeatedly appears in a list of requests passed to MPI_Waitsome, MPI_Testsome, or MPI_Request_get_status_some and a matching send has been posted, then the receive will eventually succeed unless the send is satisfied by another receive; and similarly for send requests.

It should otherwise function similarly to MPI_Waitsome and MPI_Testsome: it will return in *outcount* the number of completed requests, with the corresponding statuses and indices of the original requests in output arrays. Note that while an input array consisting entirely of inactive or null requests should return immediately, *outcount* in this case will be MPI_UNDEFINED. *outcount* and its corresponding output arrays should only include the active requests that have been completed.

4 USAGE

To illustrate the benefits of these new functions, we present examples of tool code that can be simplified using the new MPI_Request_get_statusXXX functions.

4.1 Code simplification

Without MPI_Request_get_status_some, a tool that wished to wrap MPI_Testsome and inspect statuses might take the incorrect approach shown in Figure 2.

Such a wrapper function is faulty because by the time PMPI_Testsome is called additional requests might have reached completion and will be reported as completed by the MPI_Testsome call. These additionally completed requests will lead to unexpected results for the tool.

The modified version in Figure 3, still using MPI_Request_get_status, provides consistent results. Note that, because each MPI_Request_get_status and MPI_Wait call may make progress, Figure 3 is not equivalent to the unmodified program—more requests may be completed when the tool is present.

With MPI_Request_get_status_some, we can instead use the approach in Figure 4.

```
int MPI_Testsome(int incount, MPI_Request req[], int*
↪  outcount, int* indices, MPI_Status* statuses[])
{
  // allocate temp_statuses array of statuses,
↪  size=incount
  int flag=0;
  for(int i=0; i < incount; i++) {
    PMPI_Request_get_status(req[i], &flag,
↪  temp_statuses[i]);
    // save status along with each request
  }
  PMPI_Testsome(incount, req, outcount, indices,
↪  statuses);
  // now post-process saved requests and statuses based
↪  on result of Testsome
  return ret;
}
```

Figure 2: An example of faulty wrapping of `MPI_Testsome`.

```
int MPI_Testsome(int incount, MPI_Request req[], int*
↪  outcount, int* indices, MPI_Status* statuses[])
{
  MPI_Status temp_status;
  for(int i=0; i < incount; i++) {
    int flag=0;
    PMPI_Request_get_status(req[i], &flag, temp_status);
    if (flag) {
      // process request completion
      PMPI_Wait(req, statuses+*outcount);
      indices[*outcount]=i;
      *outcount++;
    }
  }
  return ret;
}
```

Figure 3: An example of semantic-altering wrapping of `MPI_Testsome`.

Provided that `MPI_Request_get_status_some` will complete the same requests as `MPI_Testsome`, we now have the ability to perform the non-destructive query first, and then have `PMPI_Test` trivially complete and free each completed request, as above.

With the new proposed functions, old tool code, such as in Figure 1, can be replaced with new tool code as shown in Figure 5. This is in particular relevant when attempting to detect whether and when a request has been cancelled: if the necessary data about the original request has not been preserved, a tool has no way of knowing *which* request has been cancelled when examining the output statuses from `MPI_WaitXXX` or `MPI_TestXXX`.

```
int MPI_Testsome(int incount, MPI_Request req[], int*
↪  outcount, int* indices, MPI_Status* statuses[])
{
  int ret = PMPI_Request_get_status_some(incount, req,
↪  outcount, indices, statuses);
  for(int i=0; i < *outcount; i++){
    // tool code to handle the successful test on
↪  req[indices[i]]
   PMPI_Test(req[indices[i]], &flag, MPI_STATUS_IGNORE);
  }
  return ret;
}
```

Figure 4: Use of `MPI_Request_get_status_some` to simply and correctly inspect statuses in a wrapper for `MPI_Testsome`

```
int MPI_Testsome(int incount, MPI_Request req[], int*
↪  outcount, int* indices, MPI_Status* statuses[])
{
  int ret = PMPI_Request_get_status_some(incount, req,
↪  outcount, indices, statuses);
  for(int i=0; i < *outcount; i++) {
    int flag = 0;
    orig_req = req[indices[i]];
    process_deactivated_request(orig_req, statuses[i]);
    PMPI_Test(orig_req, &flag, MPI_STATUS_IGNORE);
  }
  return ret;
}
```

Figure 5: Simplified `MPI_Testsome` wrapper with no need to save requests pre-call, via transformation to use `MPI_Request_get_status_some` and `MPI_Test`.

4.2 Nested tools

It is important to remember when considering these proposed new functions that a tool substituting `MPI_Request_get_statusXXX` calls for `MPI_TestXXX` or `MPI_WaitXXX` calls is in fact altering the MPI calls that are visible to other tools that lie between them and the actual MPI implementation, whether this nesting is performed by P^nMPI, the forthcoming QMPI interface, or any other mechanism. If, as in Figure 5, a tool replaces `MPI_Testsome` with `MPI_Request_get_status_some` and individual `MPI_Test` calls, another tool would see request completions via those calls to `MPI_Test`, not as part of the original application's `MPI_Testsome`. This is not a novel concern for nested tool usage; cases such as the replacement of collective operations with an equivalent sequence of point-to-point operations, such as in the work of Zhang et al. [6], have been considered critical motivating use cases for the development of QMPI [1]. However, the `MPI_Request_get_statusXXX` functions particularly encourage this sort of substitution and require corresponding care in a multi-tool environment. In a wrapper as shown in Figure 4, the tool already knows that the request is completed. Therefore, the call to

`PMPI_Test` could also be replaced with `PMPI_Wait` telling a potential other tool, that the request will certainly be completed by this call. If the nested tool is only interested in the actual completion of requests, this information would be sufficient and the nested tool might not even intercept the `MPI_Request_get_statusXXX` calls.

5 CONCLUSION

The proposed additions to the MPI 4.1 Standard will simplify tool and library code and allow internal layering within implementations, while ensuring consistency across non-destructive and destructive checks of the status of multiple requests.

REFERENCES

[1] Elis, B., Yang, D., and Schulz, M. Qmpi: A next generation mpi profiling interface for modern hpc platforms. In *Proceedings of the 26th European MPI Users' Group Meeting* (2019), pp. 1–10.
[2] Forum, M. *MPI 4.0 Standard*, 2022.
[3] Hilbrich, T., Schulz, M., de Supinski, B. R., and Müller, M. S. Must: A scalable approach to runtime error detection in mpi programs. In *Tools for High Performance Computing 2009: Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden* (2010), Springer, pp. 53–66.
[4] Knüpfer, A., Rössel, C., Mey, D. a., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., et al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, September 2011, ZIH, Dresden* (2012), Springer, pp. 79–91.
[5] Schulz, M., and De Supinski, B. R. P^nmpi tools: A whole lot greater than the sum of their parts. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing* (2007), pp. 1–10.
[6] Zhang, J., Zhai, J., Chen, W., and Zheng, W. Process mapping for mpi collective communications. In *Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings 15* (2009), Springer, pp. 81–92.