Compiler-enabled optimization of persistent MPI Operations

Tim Jammer
Department of Computer Science
Technical University of Darmstadt
Darmstadt, Germany
tim.jammer@tu-darmstadt.de

Christian Bischof
Department of Computer Science
Technical University of Darmstadt
Darmstadt, Germany
christian.bischof@tu-darmstadt.de

2022 IEEE/ACM International Workshop on Exascale MPI (ExaMPI) | 978-1-6654-6341-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/EXAMPI56604.2022.00006

***Abstract*—MPI is widely used for programming large HPC clusters. MPI also includes persistent operations, which specify recurring communication patterns. The idea is that the usage of those operations can result in a performance benefit compared to the standard non-blocking communication. But in current MPI implementations, this performance benefit is not really observable. We determine the message envelope matching as one of the causes of overhead. Unfortunately, this matching can only hardly be overlapped with computation. In this work, we explore how compiler knowledge can be used to extract more performance benefit from the usage of persistent operations. We find that the compiler can do some of the required matching work for persistent MPI operations. As persistent MPI requests can be used multiple times, the compiler can, in some cases, prove that message matching is only needed for the first occurrence and can be entirely skipped for subsequent instances.**

In this paper, we present the required compiler analysis, as well as an implementation of a communication scheme that skips the message envelope matching and directly transfers the data via RDMA instead. This allows us to substantially reduce the communication overhead that cannot be overlapped with computation. Using the Intel IMB-ASYNC Benchmark, we can see a communication overhead reduction of up to 95 percent for larger message sizes.

***Index Terms*—MPI, message passing interface, message matching, persistent MPI communication, compiler analysis, HPC**

I. Introduction

The Message-Passing Interface (MPI, [1]) is the de-facto standard for distributed memory computing in high-performance computing (HPC). With MPI, one can overlap communication and computation in order to hide the overhead necessary for communication. Unfortunately, not all communication overhead can be hidden behind computation. In particular, overlapping the work necessary to perform message envelope matching with computations is quite challenging, although proposals have been made to offload this matching to specialized hardware [2], [3]. The requirements for tag matching hardware for different HPC workloads have been studied [4], but hardware-based approaches will always have less flexibility regarding possible future changes in communication patterns. Therefore, message envelope matching is a particularly costly part of MPI communication [5]. In this work, rather than further optimizing tag matching, we want to focus on avoiding it when possible, using compiler knowledge.

```
1 MPI_Request request;
2 //Initialize the persistent request, this does not start any communication:
3 MPI_Send_init(buf, count, dtype, dest, tag, comm,
      &request);
4 for (i=1; i<N; i++) {
5   // start the communication, (basically the same as):
6   // MPI_Isend(buf, count, dtype, dest, tag, comm, &request)
7   MPI_Start(&request);
8   do_some_computation()
9   // wait for completion:
10  MPI_Wait(&request, MPI_STATUS_IGNORE);
11 }
12 // free the persistent request
13 MPI_Request_free (&request);
```

Listing 1: Usage of a persistent MPI sending operation

The compiler can aid with the efficient usage of MPI. For example, a compiler can automatically extend the computation-communication overlap [6], [7], [8]. In previous work, we showed that the compiler can use static analysis to find out if the assertions introduced in MPI 4.0, hold for a given application [9]. Based on this approach, in this work, we want to explore if the compiler could use static analysis to assist with message matching. In particular, we want to statically find opportunities where we can avoid the usually required matching procedure altogether.

MPI also includes persistent operations that can be used multiple times. The usage of a persistent MPI operation is illustrated in Listing 1: First, one needs to initialize a persistent request (line 3), then one can use the request for an arbitrary number of non-blocking communication operations. Communication is initialized with `MPI_Start` (line 7 in Listing 1) and can be completed like any non-blocking operation with the usual function calls like `MPI_Wait` or `MPI_Test` (line 10). Finally, one needs to clean up all resources used by the persistent request (line 13 in Listing 1). To allow programming flexibility, a message sent by a persistent operation may be received with any receive operation and vice versa. But if a persistent send operation will always match with a persistent receive operation, one may skip the message matching once it has been established which two persistent operations match. This is possible as the message envelope of a persistent operation never changes (as can be seen in Listing 1). Therefore, we want to explore the compiler's capability to decide if a persistent operation can only be matched by another persistent

2831-3372/22/$31.00 ©2022 IEEE
DOI 10.1109/ExaMPI56604.2022.00006

operation. In this case, message matching is only really needed for the first instance of communication and can be skipped for all following ones.

In this paper, we make the following contributions:

- Present the compiler analysis necessary to avoid most of the message matching for persistent MPI operations
- Present a rendezvous protocol to send persistent MPI messages without the need for message matching
- Evaluate the possible performance benefit of this approach

This paper is structured as follows. First, we will present the necessary compiler analysis in Section II, before discussing possible communication protocols without the need for extensive message envelope matching in Section III. We evaluate the performance gained by this approach in Section IV. We briefly discuss the limitations of our current implementation and how our approach relates to another approach for optimizing persistent MPI operations in Section V, before concluding in Section VI.

II. Compiler Analysis

In this section, we describe the compiler analysis we implemented as an LLVM compiler pass in order to find out if we can eliminate some of the necessary message matching. We first explain the idea of our analysis and what information is necessary in order to avoid unnecessary matching operations in Section II-A. Subsequently, we explain how the compiler is able to gather the required information at compile time in Section II-B.

A. Analysis Overview

If we want to skip proper message matching for persistent operations, we have to make sure that a persistent send/receive pair that matches once will also match for every subsequent communication operation of the two involved persistent requests. This way, we only need to do the matching for the first occurrence of the persistent communication and can skip it for the other occurrences of the involved persistent communication operation.

For each persistent sending operation, we need to decide if there is a possible "matching conflict" with any other MPI communication that may take place during the lifetime of the persistent request. A "matching conflict" for a send operation means that there is another outgoing message that possibly can have a matching message envelope. In this case, we know that a matching persistent receive operation on the other side can match both: the persistent operation as well as the other conflicting message sent. Therefore, in this case, we can *not* skip the proper message matching to ensure a correct execution. If there is no possibility of a "matching conflict", we know that *if* the receiver matches with a persistent receive operation (with no wildcards), the persistent receive can only match the message sent by this sending operation, as we know that no other matching messages will be sent. In this case, we have found a possibility to skip some of the message matching (as long as no wildcards are used on the receiver side).

```
1  MPI_Request request1, request2;
2  MPI_Recv_init(buf, count, dtype, source, tag,
       comm, &request1); //has no matching conflicts
3  MPI_Recv_init(buf, count, dtype, source, tag+1,
       comm, &request2); //can conflict with MPI_Recv (line 6)
4  MPI_Start(&request1);
5  MPI_Wait(&request1, MPI_STATUS_IGNORE);
6  MPI_Recv(buf, count, dtype, MPI_ANY_SOURCE, tag
       +1, comm, status);
7  MPI_Barrier(MPI_COMM_WORLD);
8  MPI_Start(&request2);
9  MPI_Wait(&request2, MPI_STATUS_IGNORE);
10 MPI_Request_free (...); // free both requests
```

Listing 2: Illustration of matching conflicts on the receiver side

```
1  MPI_Request request1, request2;
2  MPI_Send_init(buf, count, dtype, dest, tag, comm
       , &request1); //has no matching conflicts
3  MPI_Send_init(buf, count, dtype, dest, tag+1,
       comm, &request2); //no matching conflicts
4  MPI_Start(&request1);
5  MPI_Wait(&request1, MPI_STATUS_IGNORE);
6  MPI_Start(&request2);
7  MPI_Wait(&request2, MPI_STATUS_IGNORE);
8  MPI_Barrier(MPI_COMM_WORLD);
9  MPI_Start(&request2);
10 MPI_Wait(&request2, MPI_STATUS_IGNORE);
11 MPI_Request_free (...); // free both requests
```

Listing 3: A possible sender side for Listing 2, which does provoke a matching conflict on the receiver besides the barrier synchronization, although no local matching conflicts exist

But at this stage, the receiver may still match with a mix of persistent and standard communication operations. Therefore, we also "mirror" this analysis for the receive operation and check if there exist any "matching conflict" on the receiver side. This time the analysis also needs to take wildcards into account, to check for possible "matching conflicts". If a persistent receive operation without a "matching conflict" is found, it means that when a matching message arrives, it can only be matched by the persistent operation in question.

Combining both "directions" of the analysis gives us all the knowledge necessary to state, that if a persistent send/receive pair is matched once, it will always match. In this case, we can skip the matching for the subsequent occurrences of this persistent operations. The combination of both parts will happen at runtime, when the first message is matched. Therefore, the compiler only "annotates" that this operation is eligible for skipping further matching, *if* the matching operation on the other rank is also eligible for it. This means, that the compiler can find out the necessary information by a local analysis only, without the need to match send/receive operations at compile time.

This is illustrated in Listings 2 and 3. For the receiver side in Listing 2, the persistent operation in line 2 does not have any matching conflict, as both the persistent operation in line 3 as well as the blocking operation in line 6 use a different message tag. In this case, it does not matter that the operation in line 6 does use a wildcard for message matching. In this example, *any* incoming message that has a matching envelope with the first operation in line 2 can only be matched by this operation. But the second persistent operation (line 3 in Listing 2) does

conflict with the blocking operation in line 6 of Listing 2, because an incoming message with the envelope required to match the operation in line 3 can also be matched with the operation in line 6. The barrier in line 7 prevents this sort of matching conflict from the programmers perspective, as a message received before the barrier can never match a receive operation after the barrier. But our compiler analysis does *not* take into account any information about the sender side. The sender side in Listing 3 by itself does not contain any matching conflict. But we see that the combination of both does *not* have the desired property, "If two persistent operations match once, they will match always", as the send in line 2 of Listing 3 will match both receive operations in lines 3 and 6 of Listing 2. This means that in this case message matching can *not* be skipped.[1] The first operations in line 2 of each listing do have the desired property, and in this case further message matching can be skipped for those operations.

B. Technical details

For each occurrence of `MPI_Send_Init`, the compiler can often determine the corresponding `MPI_Request_Free` by tracking the usage of the MPI Request pointer. In order to find if a "matching conflict" exists, the compiler will then explore all possible codepaths between `Init` and `Request_Free` in order to find all other MPI operations that may result in a conflict.[2] In case of a persistent send operation, possible conflicting operations are all other sending operations, including the usage of other persistent requests. The compiler then tries to prove that no conflict exists, for all possibly conflicting operations.

Often the compiler can statically prove that two message envelopes are different i.e. use a different tag, target-rank or communicator. We note that the compiler does not need to know the complete message envelope at compile time in order to prove two envelopes different. For example, the compiler can prove that message tag `x` and `x+1` are different, as long as `x` does not change between these two communication operations. In this case, the value of `x` does not need to be known at compile time. We use existing features of the LLVM tool-chain, such as Scalar evolution analysis, to check if two values can be considered different. As wildcards are predefined constants, the compiler can also check at compile time if any wildcards are used. This checking for a potential matching conflict works very similar to the approach presented in [9] to detect if messages can overtake each other.

If the compiler has proven that no message conflicts exist, it will replace all MPI calls that use the given request (such as `MPI_Send_Init`, `MPI_Start`, ...) with equivalent functions that we implemented (e.g. a call to `MPI_Start` will be replaced with a call to `MPIOPT_Start`). These functions employ the same API as the original ones. The difference is that at the first message, our modified functions will check if the matching counterpart is also a modified request. If it is matched with a modified counterpart, our functions will skip the matching for subsequent operations, otherwise they will call the standard MPI communication. We will discuss the details of our implementation in Section III.

[1] In the example, it may be the case, that the wildcard operation matches a message from another rank, which leads either to a deadlock or the first sent message matching to the second receive after the barrier, depending on whether the message is buffered.

[2] If the compiler is unable to find the corresponding free, it uses the end of the execution e.g., `MPI_Finalize` as the stopping point for the conflict analysis.

III. Communication Protocol

In this section, we explain different possible communication schemes that can be utilized when no tag-matching is required. Our focus is on communication schemes that allow for as much overlap of communication with computation as possible. Therefore, we first introduce the existing methods for progressing asynchronous MPI operations.

A. Asynchronous MPI progress

In order to handle the communication, the MPI Library needs to use the CPU for various actions, such as responding to control messages, posting network operations or performing message envelope matching. These operations are commonly referred to as "progressing the communication" or simply "progress" [10].

When overlapping communication with computation, the CPU usually is busy with performing the calculation and therefore not calling into the MPI Library. This means that it is necessary to achieve communication progress without the control flow going to the MPI library.

Approaches to tackle this problem can, in general, be categorized as either thread based (such as [11]), process based (such as [12]) or interrupt-based (such as [13]). Another approach is to modify the application code, to regularly pass the control flow to the MPI Library e.g. by calling `MPI_Test` , so that communication can progress. Besides the overhead added by calls to the MPI Library this technique has the major drawback, that the application code needs to be adjusted. As [14] pointed out, the efficiency of such an approach depends on the interval where MPI is called, as too many calls into MPI leads to unnecessary overhead, while not enough calls leads to insufficient communication progress, so that the communication hasn't finished in time. This means that it is very cumbersome to use such an approach efficiently, as one has to manually select an appropriate amount of calls into the MPI library for each specific application manually.

Interrupt-based approaches suffer from a lack of scalability and a high overhead associated with context switches and are therefore not common today [15].

Process and thread based approaches usually suffer from the necessity to reserve some CPU resources, for example for a dedicated progress thread, as these resources are no longer available to the applications threads [16]. Additionally, similar to the rate of calls to the MPI library, the number of additional CPU resources depends on the particular communication scheme and therefore also needs to be fine-tuned by the user [15]. This is especially problematic when the application has different phases with different resource requirements. [15]

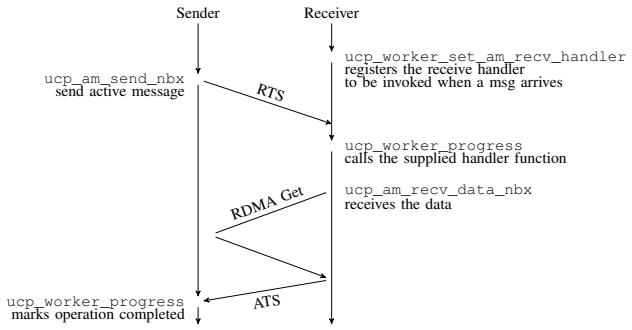

Fig. 1: Illustration of the rendezvous protocol used for UCX active messages. Based on [17]

tries to mitigate the need for specifying the amount of spare resources in advance, by using work stealing, where one process of a node can also progress the communication of all other processes.

Nonetheless, all these approaches need CPU resources in order to perform message matching, meaning that these CPU resources cannot be used by the application. But in our case, the compiler has determined that this message matching is not necessary and could be avoided entirely. Clearly, some progress is still needed in our case, as one needs to respond to control messages for a rendezvous protocol. We do think however, that this progress costs considerably less CPU resources, as less work needs to be done. We will discuss this in the next sections.

B. UCX Active Messages

MPI implementations such as Open MPI or MPICH often build on top of low level layers such as UCX (Unified Communication X)[3]. As UCX is a common building block for many MPI Implementations, it is natural to consider active messages, when no tag matching is required. One advantage of these active messages is lower overhead, due to the absence of tag matching [17].

Active messages are "matched" with an ID. When an active message arrives, UCX calls a user-provided callback function, based on the ID of the arriving message. The user needs to register a callback for each ID that it wants to receive. Therefore, when the first message of a persistent MPI operation is exchanged (and matched), the sender and receiver can also perform a handshake and negotiate a message ID to use for all later communication, as no tag matching is needed for the further communication operations for this persistent request. The receiver in turn registers the corresponding callback, so that from this point on, UCX active messages are used instead of the tag-based ones.

Active messages can be sent either eagerly, or using a rendezvous protocol. Usually an eager protocol is used for smaller message sizes, whereas larger messages are sent with a rendezvous. The rendezvous protocol for UCX active message is shown in Fig. 1.

[3]https://openucx.org/

One thing that we note for this rendezvous protocol is that the receiver needs to progress communication in the case that the rendezvous has not started when the receiver wants to begin receiving the data. One way to mitigate this problem is to call `MPI_Test` during the computation and start the data transfer once the message has arrived. On the other hand, inserting regular calls into the computation can add additional overhead. As we want to avoid such overhead, we propose to use a two-sided rendezvous protocol instead, as described in the next subsection.

We note, that this is not an "artificial" problem, as the sender may first need to calculate the necessary data before communicating it, meaning that the situation where the sender enters the communication later than the receiver will happen frequently in real-world applications.

C. Two-sided Rendezvous Protocol

In contrast to a UCX active message, the sender does know, where the receiver needs the data as the message buffer cannot change for a persistent operation. Therefore, it is also possible for the sender to initiate the RDMA operation for the data transfer.

In order to synchronize the communication, we propose a lightweight two-sided rendezvous protocol. We illustrated this protocol in Figures 2, 3 and 4, each with a different possible order of processes.

At the first execution of a persistent operation, in addition to the original MPI matching, both communication partners also perform a handshake, where all the relevant information for RDMA transfer is exchanged. In addition to the data buffer, both partners also expose a flag for synchronization of RDMA operations.

After this handshake, the following communication operations are very lightweight:

When one communication partner arrives at `MPI_Start`, it basically only needs to start one RDMA operation. Either one writes to the flag of the other communication partner that one is ready for the data transfer, or, if the local flag indicated that the other partner is ready for the data transfer, one starts the transfer of the data. When the data transfer is initialized, a second RDMA operation, signaling the communication partner the completion of the operation, is also scheduled. This is illustrated in Figures 2 and 3, depending on which process arrives first. This protocol does not require any participant to further progress communication, if the RDMA data transfer can be completely offloaded to the network hardware. When the corresponding `MPI_Wait` call is encountered, one needs to wait until the RDMA Operation for the data transfer has completed, or the other communication partner has signaled completion via the synchronization flag. We call this a two-sided rendezvous protocol, as both communication partners can initiate the data transfer with RDMA. This is a key difference to other rendezvous protocols used for communication, as usually only the receiver will initiate the data transfer after it has been told where the data is located on the sender. In our case with persistent MPI operations, where the compiler

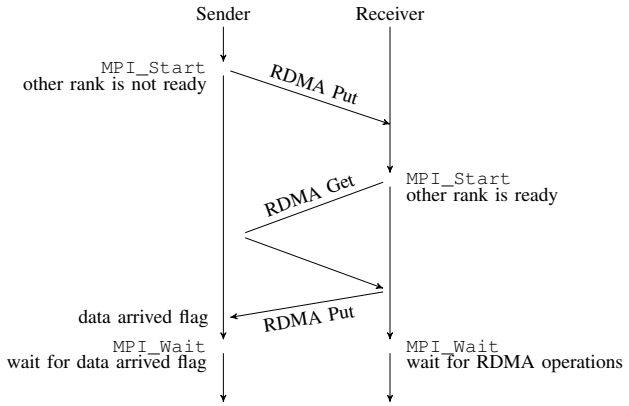

Fig. 2: Illustration of the two-sided rendezvous, in case sender arrives first

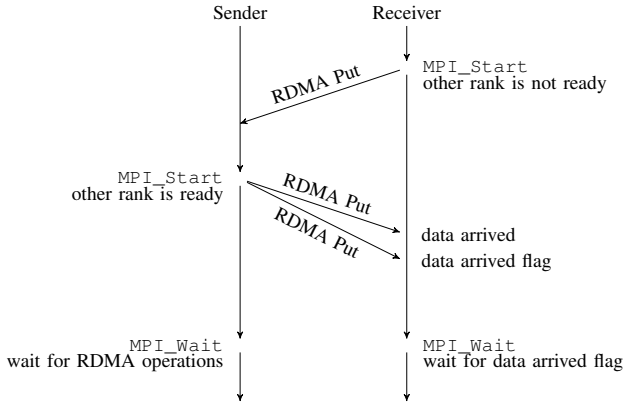

Fig. 3: Illustration of the two-sided rendezvous, in case receiver arrives first

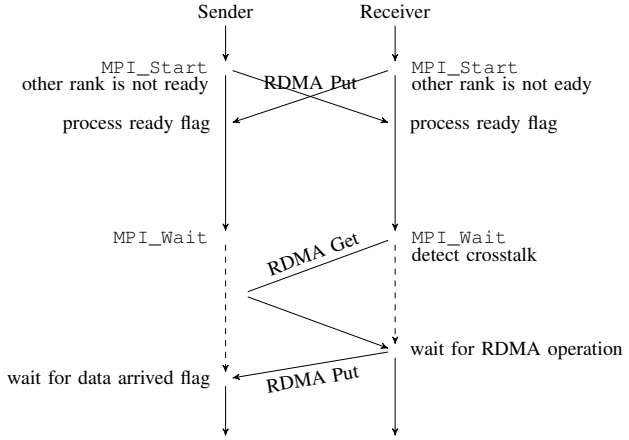

Fig. 4: Illustration of the two-sided rendezvous, in case crosstalk occurs

has determined that message matching is not necessary, it is possible to use such a protocol, as both communication partners can perform a handshake in advance, exchanging all the necessary information. We see that such a protocol lacks flexibility, compared to a one-sided rendezvous. For example, it is not possible to receive data from different communication partners without performing a distinct handshake with each one in advance and also the same message buffers needs to be used for each communication operation. Therefore, such a two-sided rendezvous protocol is not widespread, but in our case this lack of flexibility is no impediment.

There are different variations of "one-sided" rendezvous protocols outlined in [18]. They all have in common that the processes agree beforehand which one initiates the data transfer, by selecting the protocol to use. We can utilize a "two-sided" protocol, as we do not need to agree beforehand which process initiated the RDMA data transfer. One reason why this kind of rendezvous may not be applicable to other MPI operations so easily is the possibility of crosstalk, when both processes arrive roughly at the same time, both expecting the other to initiate the data transfer. This case is shown in Fig. 4. For non-persistent MPI Operations it may be hard to tell if an incoming control message is the result of crosstalk, or if it is originating from the initiation of a different operation. This is one of the disadvantages encountered by [19], where a "two-sided" protocol, where both the sender and receiver can initiate the data transfer, has been proposed previously called a speculative rendezvous protocol. In our implementation the receiver will detect crosstalk and initiate the data transfer in this case, as shown in Fig. 4. The detection of crosstalk and distinguishing it from the initiation of another operation is easily possible in our case, as each pair of operations is matched in advance and has a dedicated flag for synchronization. In our implementation, we use this flag to count the number of operations performed on each request, to make sure that both participants try to execute the same operation. This means that the approach presented in [19], does need additional control messages to ensure that matching is done correctly even if miss-predictions in the speculative protocol occur. This approach by [19] does additionally create the problem, that miss-predictions in the speculative protocol will slow down matching for all other messages, as they will cause "unexpected matchings", which have been shown as particularly harmful for good matching performance [20].

When crosstalk happens often, the advantage of our protocol, that you do not need to call `MPI_Test` to achieve progress, is lost. A general downside of all rendezvous protocols is that they are not ideal when the sender and receiver arrive at the same time [18]. [18] therefore proposes to use a protocol where one of the communication partners will wait for the control message to arrive in those cases. We will discuss how this affects the overall performance in Section IV.

We do note an additional benefit over other proposed rendezvous protocols is the smaller size of the control messages. Even less than one byte will suffice for our purpose, as the control messages in our case no *not* need to contain the information necessary for the RDMA data transfer, such as the message size and buffer location, as this information already has been exchanged during the initial handshake. In our implementation, these control messages are also realized with RDMA operations. TAs an additional benefit this means, that the incoming control messages do not take any resources (e.g.

buffer space) from the receiving end until they are handled.

Implementation Details of the Two-sided Rendezvous Protocol: As described in Section II, we defined our functions as replacement for the original functions for persistent operations. This means that our implementation adheres to almost[4] all requirements of the MPI standard, as long as *if* two of these persistent operations matches once, they match always. Matching these operations with other MPI operations as well as the original implementations of the persistent operations is still allowed, in this case the existing implementation will be used as a fallback option.

To make the replacement as seamless as possible, we added a proof-of-concept implementation of those functions into our build of Open MPI. This way, the compiler can replace the functions without the need to adjust linker settings, but one can also still replace calls by hand if desired. In order to make this extension as modular as possible, we also introduced functions for init and finalize, that must be called after the normal `MPI_Init` or before `MPI_Finalize` respectively. But this could easily be included in the original procedure for initialization and finalization.

At the first execution of a modified persistence operation, our implementation initiates a handshake to find out if the other process also matches with a modified persistent operation. The handshake message is achieved by using a different communicator[5] but otherwise the same message envelope, so that it matches the correct operation on the receiver side. Therefore, our current implementation only works with the `MPI_COMM_WORLD` communicator, but can be adapted to support other communicators as well. As there cannot be any matching conflict on the receiver, it cannot happen that this handshake is accidentally matched by the wrong persistent operation. The receiver then either

1) responds to the handshake, or
2) receives the payload without responding to the handshake, indicating that it has matched with a standard receive implementation

If no successful handshake at the first usage of the persistent operation was performed, we use the existing implementation as a fallback option for all further operations on this persistent request.

If the handshake was successful though, we know that we can skip matching for all subsequent usages of the given operation. Therefore, the handshake will exchange the necessary information (e.g. location of the send/receive buffer on the target rank), so that subsequent usages of the operation can result in a direct RDMA data transfer without the need for prior message matching. For subsequent operations, we use our proposed rendezvous protocol, as explained in Section III-C. We note that this handshake can introduce a "hiccup" at the first usage of the operation. Performing such a handshake will disturb the normal flow of communication, as it introduces additional communication overhead. But as the design goal of a persistent operation is that it is used multiple times, we think that this initial overhead is reasonable and justified.

Once the handshake was established, all subsequent usages of the operation will skip the message matching. This allows us to keep the overhead to a minimum, minimizing the time spend in the MPI layer and directly calling into UCX. When a send operation is started, either

1) the information that the data is ready is signaled, or
2) if the receiver has already signaled that it is ready: the RDMA data transfer is initialized.

The receive operation behaves analogous. Therefore, our implementation only checks a local flag and than immediately calls UCX to initiate an RDMA data transfer. Either the actual data transfer is started, or one directly writes to the flag of the other process indicating that this other process should start the data transfer.[6] This keeps the time spent in the MPI layer close to none. This is a significant improvement considering that, for example, MPICH spends up to 92 percent of the latency of an `MPI_Isend` call inside of MPICH and only 8 percent in the lower UCX level [21].

IV. Evaluation

In order to evaluate the performance gained by our approach, we use the IMB-ASYNC benchmark [22], [14], an extension of Intel's MPI benchmarks[7] specifically designed to measure the computation-communication overlap. As all persistent operations are non-blocking, we see this benchmark as suitable for our evaluation. In our opinion, the overhead that could *not* be overlapped by computation is the most important to reduce, as the other part of communication overhead can be hidden behind the computation. We extended the original IMB-ASYNC benchmark to also include a test case for persistent operations. Basically, we duplicated the existing case for non-blocking point to point communication and replaced the communication with a persistent one[8]. Our extension of Intel's IMB-ASYNC benchmark is based on version 0.0.5[9]. The basic idea of the IMB-ASYNC benchmark is to measure the computation time as well as the total time, to derive the communication overhead as the difference of these two.

We ran our experiment on the Lichtenberg cluster at TU Darmstadt. The nodes are running Red Hat Enterprise Linux 8.5 and are equipped with 2 Intel Xeon Platinum 9242 processors. The nodes are connected with InfiniBand HDR100 in a fat tree topology. The cluster uses the Slurm scheduler version 20.02.5. We used clang 11.1 as the compiler.

The capabilities of our compiler analysis are demonstrated by successfully using it on the IMB-ASYNC benchmark code. For the compilation of the IMB-ASYNC benchmark, we did

[4]see Section V for a discussion on where our proof of concept implementation currently does not strictly comply to the standard.

[5]only one duplication of the used communicator is necessary for all of the handshakes

[6]The flag has been agreed upon at the handshake as well.

[7]https://github.com/intel/mpi-benchmarks

[8]The extended IMB-ASYNC benchmark is also included in our github repository: https://github.com/tudasc/MPI-CompMatch.

[9]https://github.com/a-v-medvedev/mpi-benchmarks/tree/b3ec89acf23c7b804b860294564d495ba9d2d4fc

not notice a large impact on compilation time when using our analysis. In the case of the IMB-ASYNC benchmark code, our analysis added less than 0.2% overhead to the ≈ 6 seconds it took us to compile the benchmark.

For our experiments, we used the UCX component of Open MPI (4.1.1) with OpenUCX version 1.12.0. We used four different implementations of the communication, all based on the same Open MPI version, as described in Section III:

Normal: The unaltered implementation as the baseline,
Eager: UCX active messages (without tag matching) with an eager protocol (see Section III-B),
Rendezvous 1: UCX active messages (without tag matching) with a rendezvous protocol (see Section III-B),
Rendezvous 2: The proposed two-sided rendezvous protocol described in Section III-C.

In all cases, we do *not* include extra measures to progress the communication, such as an additional CPU set aside for a progress thread or regular calls to `MPI_Test` (also refer to Section III-A).

We repeat each individual measurement 25 times. Inside each measurement, we measure at least 64 communication operations. To measure the cost of network communication, we split the involved processes among different compute nodes. In all measurements, we use a computation-workload that in theory is sufficient to completely hide all communication overhead. In order to measure a real world scenario, we do not take special measures for exact placement of the processes in the network topology and used slurm's assignment. We note, that the latency of the network should be hidden behind the computation by all of the approaches anyway, therefore sampling a random distribution of process placements is sufficient to judge the general applicability of our proposed approach. There are various ways, how system noise may affect the time necessary to perform the *computation*, such as the operating system interrupting processes for some reason, or difference in CPU speed, for example, due to thermal constraints. In order to filter out this system noise, we only take into account the overhead values for the processes with the slowest *computation* time. For any process, where the *computation* time is below 90% of the *computation* time of the slowest process, we do not consider it as overhead, as the "overhead" measured by the benchmark does also include the time spend waiting for the other slower processes to finish their computation. This case is illustrated in Fig. 5, where one can see that the time spend inside MPI does not only include the message transfer overhead but also the time necessary to wait for the other process. In these cases, we observed the overhead measured on all of the faster processes to be basically the same for all approaches, as it is dominated by the waiting time. For the slower rank, the messages are sent in time and therefore can be overlapped with computation, meaning that only the relevant overhead is measured.

Fig. 6 shows the communication overhead for varying message buffer sizes after a warm-up period of 8 communication operations for the communication of two processes. Each measurement includes 64 communication operations. The Y

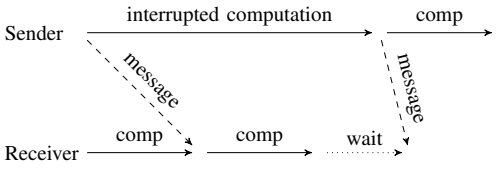

Fig. 5: Illustration of the case that computation takes longer on one process.

axis depicts the communication overhead that can ***not*** be overlapped by computation. The X axis shows different message buffer sizes. In order to show the distribution of measurements, we show a violin plot for each set of measurements and also depicted all measured overhead values as individual points. As one can see in the violin plots, we have not included all outliers for a better overview. For each measurement, no more than two data points are outside of the plotted area.

Forcing UCX to use an eager protocol does lead to a deadlock due to the lack of resources in our implementation, which is the reason why no data is plotted for the larger message sizes. We do not see a significant improvement, when using eager UCX active messages instead of tag-matching ones for smaller message sizes. We think that this is attributable to the fact, that copying the data from the network buffer to the destination is more expensive than tag matching in this case.

For the usage of the rendezvous protocol of UCX active message (depicted as Rendezvous 1 in Fig. 6), we see that it does not bring a performance benefit. The reason is that in our implementation, the receiver does not initiate the data-transfer until `MPI_Wait` is called for most communication operations, as the control-flow does not return to MPI during the calculation. There are strategies to circumnavigate that, for example a thread that will be scheduled when a message arrives, even if there isn't a spare CPU core around for it. As we see, the normal implementation does successfully employ those strategies, in order to achieve a good performance. But the normal implementation falls short, when comparing it to our proposed two-sided rendezvous protocol (explained in Section III-C and depicted as Rendezvous 2 in Fig. 6). For the median communication overhead, we see a reduction of communication overhead of 95% for message sizes of at least 1 MiB. This is attributable to the fact that the time spent in the MPI layer is kept to a minimum and no tag matching takes place. In the default implementation, the actual data transfer can *not* start until the message matching has completed and the implementation therefore knows where the correct receive buffer is located, a downside we can also circumvent by eliminating the need for message matching.

When measuring the overhead incurred during the first 8 communication operations, we see that especially for smaller buffer-sizes, our implementation performs worse with a higher variability, due to the "hiccup" introduced by the handshake in our implementation, as described in Section III. The normal implementation does also perform worse during this initial warmup period, although not to the of our proposed communication protocol. But this effect is not present once this

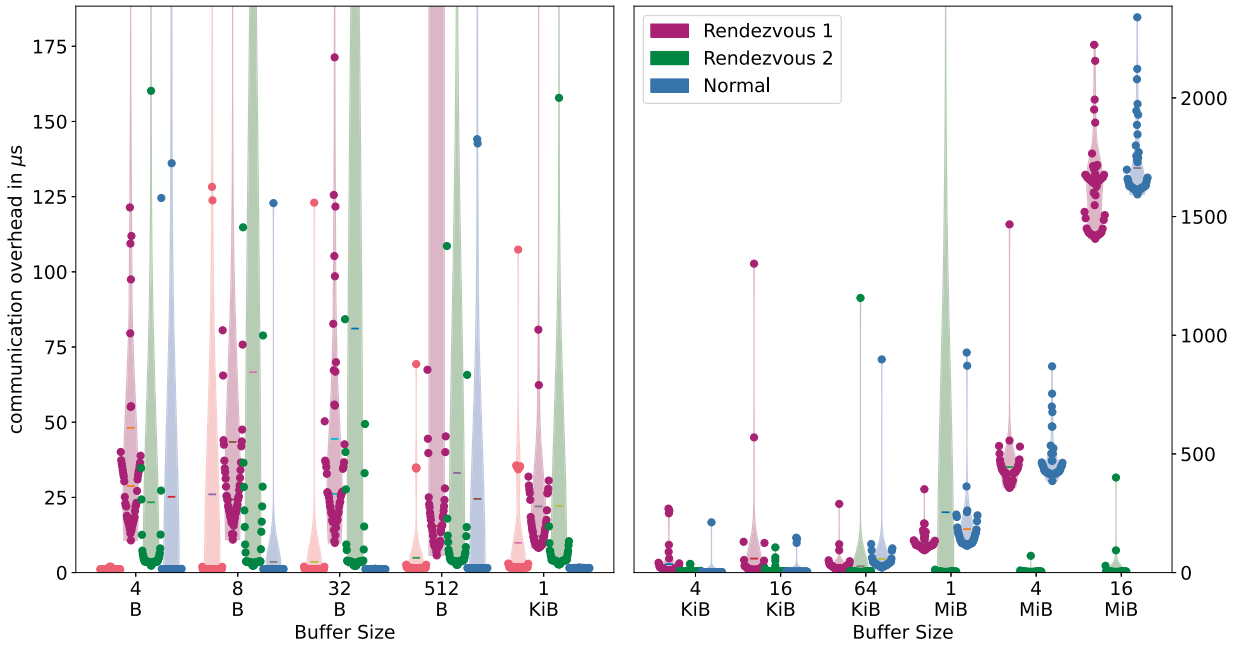

Fig. 6: Communication overhead for varying message buffer sizes for two processes. The Y axis depicts the communication overhead that can ***not*** be overlapped by computation, while the X axis shows different message buffer sizes. Normal means the existing implementation. Eager and Rendezvous 1 show the usage of UCX active messages with the eager and rendezvous protocol respectively, as explained in Section III-B. Rendezvous 2 depicts our two-sided rendezvous protocol described in Section III-C.

initial warm-up phase is over. As the idea behind persistent communications is that they can be used multiple times, we do think that having this additional overhead during the warm-up period is justifiable. But this of course depends on the number of communication operations performed afterwards to amortize this initial overhead for the handshake. In our test, the cost of this initial hiccup amortizes after 40 communication operations[10]. But when one factors in, that the normal implementation also performs better after a warm-up phase, our implementation outperforms the original one after as little as two to three communication operations.

When repeating the experiment for two processes with the processes confined to one compute node, instead of spread among different ones, we observed similar results. The only difference we observe are a couple of more outliers for the standard implementation, but the relative difference between the different protocols stays roughly the same.

Fig. 7 shows the (median) communication overhead for various number of processes for larger message sizes. We do note that, although the processes only communicate pairwise in this test, the standard implementation does suffer a performance penalty, when more processes are around. As OpenMPI maintains multiple matching queues, one for each other processes, the cost of message matching grows with more involved processes [5]. We do see that the absence of tag matching does eliminate this performance penalty (refer to the dotted lines), presenting an opportunity for improved scalability.

[10]measured with the 16KiB message size

V. Discussion

In this section we discuss technical details where our current implementation falls short of the MPI standard's requirements, as well as the possibility to enhance our approach by combining it with other approaches for optimization of persistent communication.

Depending on how strict one reads the MPI Standard, our current implementation does not fully adhere to all of its requirements. The MPI standard mandates that the "point-to-point persistent initialization calls involve no communication" (section 3.9 of the standard [1]). But in our implementation in order to speed up the handshake, the initialization of a persistent operation will also initialize a non-blocking send/receive operation for the handshake. One can mitigate this "flaw" of our implementation by only performing the handshake when the persistent request is used for the first time. Nevertheless, as we use a non-blocking operation to start the handshake, we think that this still follows the meaning of the standard insofar as the initialization of a persistent operation is a local operation and will return independent of the other processes' state.

Another point, where our current proof-of-concept implementation may not meet the expectation of a quality MPI im-

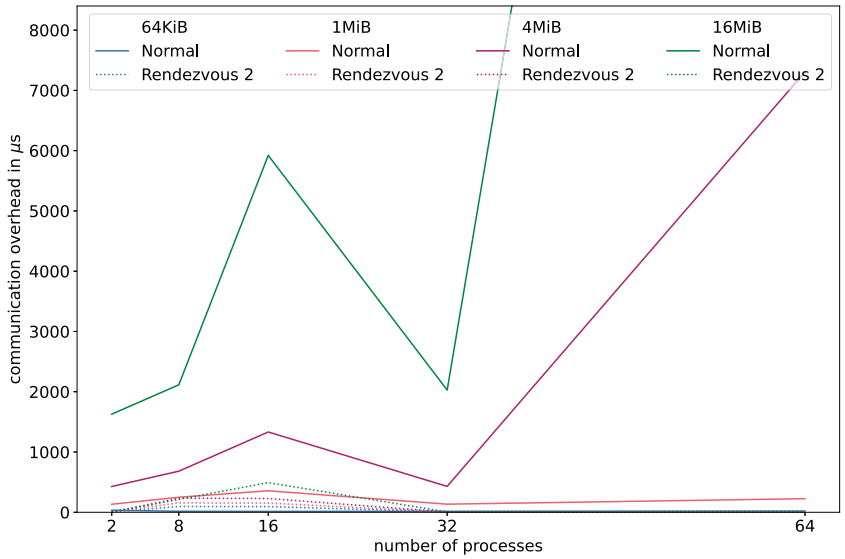

Fig. 7: Communication overhead for varying number of processes. The Y axis depicts the communication overhead that can ***not*** be overlapped by computation, while the X axis shows hte amount of processes. Normal means the existing implementation. Rendezvous 2 depicts our two-sided rendezvous protocol described in Section III-C.

plementation, is the possible leak of resources. If a handshake message does not get a response, the associated `MPI_Isend` is never properly finished. This problem can be mitigated by using a better handshake implementation which does clean up the resources used for an unsuccessful handshake. For this initial experiment, we have not tested our implementation for multi-threaded usage, but according to MPI, the user is responsible for proper synchronization of the usage of persistent requests anyway.

Regarding the effect of possible crosstalk in our two-sided rendezvous protocol, we observed that after the initial warm-up phase, the processes naturally "ordered" themselves, so that crosstalk does not occur in our experiments anymore. We note that this effect of the processes "ordering" themselves may be influenced by other MPI operations present, especially collective operations. Maybe our approach can benefit from work that models idle waves and studies the influence collective operations have on them, such as [23]. As the presence of a particular collective can be detected by the compiler, it may be beneficial to include such a model and only use the two-sided rendezvous protocol when crosstalk is unlikely to happen after the warm-up phase.

We note that our implementation of the proposed two-sided rendezvous protocol, as described in Section III-C, may not be fully optimized yet. It may be worth to investigate how it can further benefit from proposed optimizations of rendezvous protocols, such as [24], to remove one of the necessary control messages. This means that we also have not yet taken into account if our approach can also benefit from other approaches that were suggested to enhance the communication performance specifically for persistent MPI operations. We mention [25] in particular, where communication is scheduled via multiple RDMA engines in order to achieve a better performance. Coincidentally, this approach also has the "downside" that it assumes that persistent send operations always match with persistent receive operations. The authors stated that "these restrictions can be considered to be very natural and not a big problem in most MPI programs"[25]. But with our proposed compiler analysis one can make sure that persistent operations can only be matched with persistent operations and one needs not solely to rely on the programmer to adhere to this restriction that is not mandated by the MPI standard. Therefore, this other approach may also benefit from our proposed compiler analysis.

Currently, the scope of our compiler analysis is limited, in the way, that the lifetime of a persistent request, from initialization until the call to `MPI_Request_free`, has to be confined to one function. In the future, we want to extend the compiler analysis to also be able to detect matching conflicts between different functions. One idea based on MetaGC [26] is to annotate a call-graph of the program with all the possible matching conflicts and then check the graph for paths between these.

We propose that the information that a persistent operation will always be matched with one persistent counterpart be added as a new MPI assertion, so that the user can still benefit from the performance gain even if the compiler analysis fails to detect that this holds. We have seen that this restriction of MPI's behavior opens up the possibility for several different optimization strategies. Therefore, having this assertion may encourage other researchers as well as MPI implementers to seek out further optimization potential in this area.

VI. Conclusion

In this work, we demonstrated the compiler's ability to find opportunities where MPI message matching can be avoided for persistent operations. As we explained in Section II, this can be done as an entirely local analysis without the need to match messages statically. Additionally, the concrete values of the message envelopes do not need to be determined at compiler time, as the compiler can often symbolically determine if message envelopes differ. With our two-sided rendezvous protocol explained in Section III, we find that this can achieve a communication overhead reduction of up to 95 percent for larger message sizes, for the part of the overhead that can not be hidden behind computation (see Section IV).

As presented in Section V, we want to further enhance the scope of our compiler analysis and optimize our implementation of the communication protocol in the future.

The code for the compiler analysis, as well as the implementation of the communication, is available at our github: https://github.com/tudasc/MPI-CompMatch

Acknowledgment

This work was funded by the Hessian Ministry for Higher Education, Research and the Arts through the Hessian Competence Center for High-Performance Computing. The authors gratefully acknowledge the computing time provided by them on the high-performance computer Lichtenberg at the NHR Center NHR4CES at TU Darmstadt. NHR is funded by the Federal Ministry of Education and Research and the state governments participating on the basis of the resolutions of the GWK for the national high performance computing at universities (www.nhr-verein.de/unsere-partner).

References

[1] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 4.0," https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf, 2021.

[2] Q. Xiong, A. Skjellum, and M. C. Herbordt, "Accelerating MPI message matching through FPGA offload," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 191–1914.

[3] K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell, "A hardware acceleration unit for MPI queue processing," in *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2005, pp. 10–pp.

[4] S. Levy and K. B. Ferreira, "Evaluating tradeoffs between MPI message matching offload hardware capacity and performance," in *Proceedings of the 26th European MPI Users' Group Meeting on - EuroMPI '19*. ACM Press, 2019.

[5] W. Schonbein, M. G. Dosanjh, R. E. Grant, and P. G. Bridges, "Measuring multithreaded message matching misery," in *European Conference on Parallel Processing*. Springer, 2018, pp. 480–491.

[6] A. Danalis, L. Pollock, M. Swany, and J. Cavazos, "MPI-aware compiler optimizations for improving communication-computation overlap," in *Proceedings of the 23rd international conference on Supercomputing*, 2009, pp. 316–325.

[7] V. M. Nguyen, E. Saillard, J. Jaeger, D. Barthou, and P. Carribault, "Automatic code motion to extend MPI nonblocking overlap window," in *International Conference on High Performance Computing*. Springer, 2020, pp. 43–54.

[8] J. Guo, Q. Yi, J. Meng, J. Zhang, and P. Balaji, "Compiler-assisted overlapping of communication and computation in MPI applications," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2016, pp. 60–69.

[9] T. Jammer, C. Iwainsky, and C. Bischof, "Automatic detection of MPI assertions," in *International Conference on High Performance Computing*. Springer, 2020, pp. 34–42.

[10] A. Ruhela, H. Subramoni, S. Chakraborty, M. Bayatpour, P. Kousha, and D. K. Panda, "Efficient asynchronous communication progress for MPI without dedicated resources," in *Proceedings of the 25th European MPI Users' Group Meeting*, 2018, pp. 1–11.

[11] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Joó, "Improving concurrency and asynchrony in multithreaded MPI applications using software offloading," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Nov. 2015.

[12] M. Si and P. Balaji, "Process-based asynchronous progress model for MPI point-to-point communication," in *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, Dec. 2017.

[13] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda, "RDMA read based rendezvous protocol for MPI over InfiniBand," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '06*. ACM Press, 2006.

[14] A. V. Medvedev, "IMB-ASYNC: a revised method and benchmark to estimate MPI-3 asynchronous progress efficiency," *Cluster Computing*, pp. 1–15, 2022.

[15] K. Ouyang, M. Si, A. Hori, Z. Chen, and P. Balaji, "Daps: A Dynamic Asynchronous Progress Stealing Model for MPI Communication," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 516–527.

[16] T. Hoefler and A. Lumsdaine, "Message progression in parallel computing - to thread or not to thread?" in *2008 IEEE International Conference on Cluster Computing*. IEEE, Sep. 2008.

[17] M. Brinskii, "UCX Active Messages," Dec. 2020. [Online]. Available: https://ucfconsortium.org/wp-content/uploads/2021/02/Mikhail_Brinskii_UCX_Active_Messages.pdf

[18] M. Small, Z. Gu, and X. Yuan, "Near-Optimal Rendezvous Protocols for RDMA-Enabled Clusters," in *2010 39th International Conference on Parallel Processing*. IEEE, Sep. 2010.

[19] M. J. Rashti and A. Afsahi, "A speculative and adaptive MPI rendezvous protocol over RDMA-enabled interconnects," *International Journal of Parallel Programming*, vol. 37, no. 2, pp. 223–246, Mar. 2009.

[20] M. Flajslik, J. Dinan, and K. D. Underwood, "Mitigating MPI message matching misery," in *International conference on high performance computing*. Springer, 2016, pp. 281–299.

[21] R. Zambre, M. Grodowitz, A. Chandramowlishwaran, and P. Shamis, "Breaking band: a breakdown of high-performance communication," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[22] A. V. Medvedev, "Towards benchmarking the asynchronous progress of non-blocking MPI operations," in *Parallel Computing: Technology Trends, Proceedings of the International Conference on Parallel Computing*, vol. 36, 2020, pp. 419–428.

[23] A. Afzal, G. Hager, and G. Wellein, "Analytic Modeling of Idle Waves in Parallel Programs: Communication, Cluster Topology, and Noise Impact."

[24] M. Takagi, Y. Nakamura, A. Hori, B. Gerofi, and Y. Ishikawa, "Revisiting rendezvous protocols in the context of RDMA-capable host channel adapters and many-core processors," in *Proceedings of the 20th European MPI Users' Group Meeting on - EuroMPI '13*. ACM Press, 2013.

[25] M. Hatanaka, A. Hori, and Y. Ishikawa, "Optimization of MPI persistent communication," in *Proceedings of the 20th European MPI Users' Group Meeting*, 2013, pp. 79–84.

[26] J.-P. Lehr, A. Hück, Y. Fischler, and C. Bischof, "MetaCG: annotated call-graphs to facilitate whole-program analysis," in *Proceedings of the 11th ACM SIGPLAN International Workshop on Tools for Automatic Program Analysis*. ACM, Nov. 2020.