Pattern-Independent Detection of Manual Collectives in MPI Programs

Alexandru Calotoiu[1,2], Christian Siebert[1,2], and Felix Wolf[1,2,3]

[1] German Research School for Simulation Sciences, 52062 Aachen, Germany
[2] RWTH Aachen University, Computer Science Department, 52056 Aachen, Germany
[3] Forschungszentrum Jülich, Jülich Supercomputing Centre, 52425 Jülich, Germany

Abstract. In parallel applications, a significant amount of communication occurs in a collective fashion to perform, for example, broadcasts, reductions, or complete exchanges. Although the MPI standard defines many convenience functions for this purpose, which not only improve code readability and maintenance but are usually also highly efficient, many application programmers still create their own, manual implementations using point-to-point communication. We show how instances of such hand-crafted collectives can be automatically detected. Matching pre- and post-conditions of hashed message exchanges recorded in event traces, our method is independent of the specific communication pattern employed. We demonstrate that replacing detected broadcasts in the HPL benchmark can yield significant performance improvements.

Keywords: MPI, collective operations, performance optimization, HPL.

1 Introduction

The most scalable parallel application codes today use message passing as their primary parallel programing model, which offers explicit communication primitives for the exchange of messages. While pair-wise communication is most common, the majority of applications require communication among larger groups of processes [5]. The latter is needed, for example, to distribute data, gather results, make collective decisions, or broadcast their outcomes. Although all those communication objectives can be mapped onto point-to-point messages between two processes, their efficient realization is often challenging.

For this reason, the Message Passing Interface (MPI) [10], the de-facto standard for message passing, defines 17 so-called *collective operations* to support the most common group exchange patterns. For example, sending data from one process to all other processes is encapsulated in the functionality of MPI_Bcast(). Although equivalent semantics could be achieved by sending the same piece of data iteratively to all processes, one at a time, using MPI_Bcast() is simpler, the resulting code looks cleaner and is easier to maintain. In addition, sophisticated implementations of MPI_Bcast() are likely to be much more efficient.

In general, MPI collectives offer advantages in terms of simplicity, expressiveness, programmability, performance, and predictability [5]. In particular, they

Published in "Proc. of Euro-Par 2012", LNCS 7484, pp. 28–39, Springer, 2012.
The original publication is available at www.springerlink.com

allow users to profit from both efficient algorithms [2,13,14] and platform-specific optimizations [8,9,7]. Such improvements have often been reported to make collective implementations several times faster. Some of them exploit hardware features such as multicast or utilize special networks for collectives not even accessible via MPI point-to-point communication. These advantages cannot be overemphasized as the efficient implementation of collective operations is a complex task, which requires detailed knowledge not only of parallel algorithms and programming but also of the specific physical properties of the target platform.

However, in spite of such benefits, not all applications today make consequent use of predefined collectives and still deploy hand-crafted ensembles of point-to-point messages instead. One way of encouraging their adoption, is to recognize manually-implemented collectives in existing codes and to suggest their replacement. Existing recognition methods available for this purpose rely on the specifics of the underlying message exchange pattern [11,3]. But given the multitude of ways collectives can be implemented, any such attempt is too restrictive.

In this paper, we show how to overcome these disadvantages using a novel approach based on a semantic detection. We propose a method for identifying patterns of point-to-point messages in compact communication traces of MPI applications that are semantically equivalent to predefined collective operations. Relying exclusively on pre- and postconditions derived from the specification of the collective operation, we do not make any assumptions regarding the specific characteristics of the pattern. Our method detects broadcasts and operations composed of broadcasts fully automatically. It detects more sophisticated collective operations with a certain degree of prior user instrumentation. Applying our method to the HPL benchmark [1] pinpoints all contained collective communication operations. Replacing those manual collectives with the corresponding MPI collectives improves the HPL performance by up to 44%.

The remainder of the paper is organized as follows: In Section 2 we formalize the semantics of collective operations and show how pre- and postconditions can be derived that can be verified based on trace data. Proving such conditions requires analyzing both the contents of messages and the paths along which they travel. How we store all the necessary information in trace files is explained in Section 3. There, we place special emphasis on the hash functions we apply to avoid excessive memory requirements and their structure-preserving properties. The actual identification of manual collectives is outlined in Section 4. A major part of it is devoted to the parallel message replay we need to track communication pathways and the additional challenges posed by more complex collective operations such as scatter or reduce. Experimental results demonstrating the benefits of our method are presented in Section 5. Finally, we compare our approach to related work in Section 6, before we conclude the paper with an outlook on future work in Section 7.

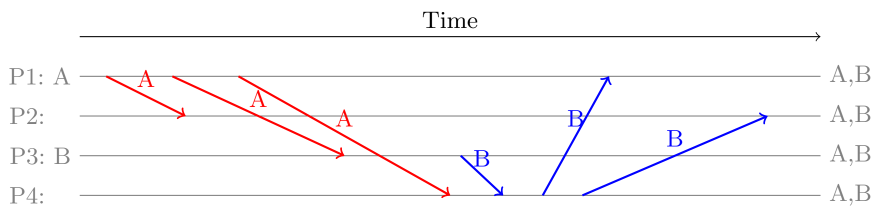


Fig. 1: Timeline diagram of two broadcast implementations

2 Semantics of Collective Operations

The MPI standard specifies only the semantics of collective operations, but does not dictate how they must be implemented. Therefore, any method capable of detecting a wide variety of manual collectives can not rely on any specific implementation. We will start our discussion with the simplest collective, the broadcast, and later move on to more challenging collectives such as scatter and reduce. For broadcast, the standard provides the following definition:

> `MPI_BCAST` broadcasts a message from the process with rank `root` to all processes of the group [. . .].

This definition does, however, not imply that a correct implementation needs to send a message from the root to all other processes directly. On the contrary, efficient implementations typically involve other processes to forward messages. By reversing this semantic definition, it is possible to infer pre- and postconditions that can tell whether or not a broadcast occurred. If at some point in time only one process owns a certain message and at a later time all processes within a group own the same message, then there must have been a collective communication that is semantically equivalent to a broadcast.

Figure 1 illustrates the behavior of two different broadcasts in the form of a timeline diagram. The diagram shows a timeline for every process and arrows between them to depict point-to-point communication. In addition, the letters A and B represent message contents. At the beginning, processes 1 and 3 own contents A and B, respectively. All processes receive further contents via messages as the time progresses. The communications with message A are semantically equivalent to a broadcast carried out using a simple centralized algorithm. The communications with message B are also semantically equivalent to a broadcast but in a hierarchical fashion. Identifying those different communication patterns as the same collective operation needs some deterministic rules. A precondition that needs to be true before the broadcast happens is that one of the processes, which is called the *root*, owns some data X. During the broadcast, X travels to the other processes in the group to which the broadcast applies. In other words, exactly one process in the group must **not** receive X before sending it. Although more receives are valid, a postcondition that needs to be fulfilled after the broadcast happened is that all processes in the group except the root must

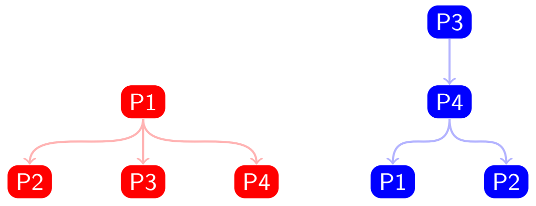


Fig. 2: Communication graphs for the two broadcast variants

have received X. Only then it is guaranteed that, at the end of the analyzed period of time, all processes share X. This fulfills the semantics of a broadcast.

Figure 2 maps the two broadcast variants from Figure 1 onto simpler communication graphs, ignoring temporal relationships. If any of the two communication graphs was not connected, then it would describe independent communications, which is in contradiction to the collective character of the operation. This implicit connectivity requirement among the processes of the group is checked during the analysis to prove the presence of a collective operation (see Section 4).

Defining pre- and postconditions for scatter and gather is more intricate, as messages can be split or concatenated. In the case of scatter, a message at the root is split into pieces to be distributed to all processes. Again, the precondition requires the original message to be located at the root and the postcondition requires non-overlapping parts of the message to be located at specific ranks. We use hashes with homomorphic properties to handle message splitting and concatenation. Even more challenging are collectives computation operations (a.k.a. reductions), where the messages are combined using, for example, arithmetic or logical operations. Nevertheless, even this can be formalized. While all such conditions can theoretically be verified under the assumption of unlimited access to the memory and message buffers used by the application, difficulties arise in practice when knowledge is restricted to manageable amounts of trace data. Two specific challenges need to be addressed:

1. For reasons of space efficiency, we only store message hashes in our trace files. Section 3 explains how we can still track many of the above-mentioned transformational relationships even with hashes.
2. In manual collectives, data destined to remain at the root may never appear in a message buffer and is thus not recorded in the traces. In Section 4, we suggest a method to make those visible again.

A further advantage of our technique, which does not require any knowledge of a collective operations' implementation, is that it can also be used to search for collective exchanges that do not have an existing primitive yet. This could motivate the standardization of new collective operations such as neighborhood or sparse collectives (under consideration for MPI-3.0). In this sense, it is not only an instrument for application optimizers but also for MPI developers.

3 Analysis Workflow and Trace Generation

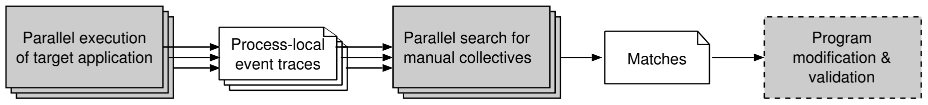

Parallel execution of target application
Process-local event traces
Parallel search for manual collectives
Matches
Program modification & validation

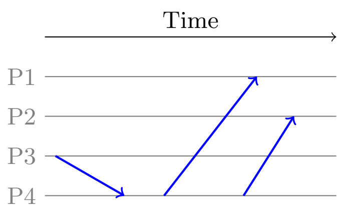
Fig. 3: Workflow of detecting manual collectives within parallel applications. Stacked boxes denote parallel programs. The user is in charge of the last step.

Figure 3 illustrates the workflow of detecting manual collectives. First, the target application is prepared for tracing by linking it to a library of PMPI interposition wrappers. Not to duplicate development efforts, we leverage the tracing infrastructure of the Scalasca toolset [4]. Only if more sophisticated collectives should be detected, the application needs further preparation as described in Section 4. The prepared application is then executed to generate a trace of happened communication events. As a next step, these traces are searched for manual collectives, which is done with the help of communication replays. This search is carried out by our analyzer, which is a parallel program in its own right. The analysis creates a list of matches, which the user then can decide to replace with predefined collectives. Since the matches characterize only a single run, the user not only needs to validate whether performance objectives are met but also has to ensure that a replacement does not violate the program's correctness.

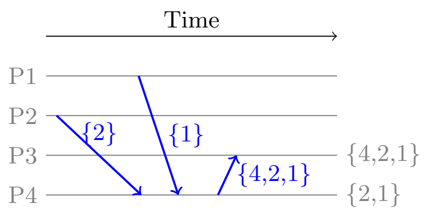
In addition to the default information Scalasca stores with communication events, we record a hash of the message payload, the starting address of the message buffer, and the MPI data type. The first item is needed to track the path along which a particular message is forwarded, and the last two to support concatenation and split of messages, as explained in Section 4. Hashing message payloads avoids storing full messages, which would consume a prohibitive amount of storage space. A hash provides a fixed-length value regardless of the message size. If two messages have the same hash value, they are identical with high probability. If their hashes are different, the messages are different for sure. Although testing for equality is a fundamental application of hashes and sufficient to detect for example broadcasts, it is not enough to identify collectives such as scatter, gather, or reduce, which split, concatenate, or combine messages. For those, we exploit homomorphic properties of certain hash functions h that ensure the following conditions for operations $\oplus$ on messages m_1 and m_2:

$$h(m_1 \oplus m_2) = h(m_1) \oplus h(m_2)$$

As a default, we use the established *CRC-32* checksum from *zlib* because it is fast, needs only 32 bits per message, gives acceptably-low collision probability, and even supports split and concatenation. We also identified further hash functions to support arithmetic or logical reductions of certain data types. Many hash algorithms, however, entail difficult compromises. For example, cryptographic



(a) Hierarchical broadcast



(b) Backward replay

Fig. 4: Identifying a hierarchical broadcast via backward replay

hashes have a lower collision probability but are much slower, need more memory and can neither be concatenated nor combined. In general, the choice of hash function is configurable, a feature which can be used to expand the coverage of our method. Unfortunately, MPI reduction operations can be arbitrarily complex as they support both user-defined data types and user-defined reduction operations. Correctly identifying reductions via hashes is already challenging for predefined data types such as floating point values, which is why we also store an 8-byte message prefix in addition to the hash. This enables testing against a predefined set of potential reduction operations. To support arbitrary user-defined data types, we utilize the *MPITypes* library by Rob Ross [12], which allows hashes to be calculated for arbitrary MPI messages.

4 Search for Manual Collectives

Our collectives detector searches for manual collectives in communication traces enriched with message hashes such as illustrated using a hierarchical broadcast example in Figure 4a. The actual search is performed via a backward replay of the traces. During this replay, we traverse the traces from end to beginning and reenact the recorded point-to-point communication in backward direction, that is, the roles of sender and receiver are reversed. To simplify the replay, our analyzer runs with the same number of processes as were used to trace the target application, giving a one-to-one mapping between application and analysis processes. The objective of the replay is to determine all processes that a particular message has visited on its way to the final destination and to propagate this information back to its origin. At the end, each process checks whether it acted as root (i.e., did not receive the message from anyone else) and whether the message has reached all other processes—directly or indirectly.

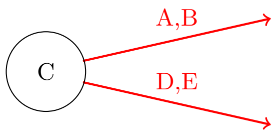
For this purpose, each process maintains a set of receivers for each message hash that will later contain the ranks of all processes that have received a message with this particular hash. Figure 4b shows these sets for the hash involved in a broadcast. At the beginning, all sets are empty. Whenever a process encounters a receive event during the backward replay for this hash, it adds its rank to the set and sends its own set along with the backward message. The receiver of a replay message then constructs the union of its own set with the set just received. If at the end of the replay one process has a set containing all other processes but not itself, a broadcast has happened with this process acting as the root. Sending the hash along with the message is one way of separating the traffic related to different broadcasts. In the example, processes 1 and 2 add themselves to the set once they hit their local receive events. After replaying the first two messages, the set of process 4 therefore includes $\{2, 1\}$. Before replaying the third message, process 4 adds itself to the set and sends it to process 3. The set of process 3 finally includes $\{4, 2, 1\}$, satisfying the condition for a broadcast with 3 as the root. Using this method, we can detect any broadcast irrespective of its particular implementation. The backward replay ensures that we can track every conceivable message pattern. In general, the direction of the replay depends on the nature of the collective operation. If information is spread as in the case of broadcast, we replay in backward direction. If information is concentrated as in the case of gather, we replay in forward direction. The goal is always to end up at the root. The all-to-all collective does not have a root, but can be detected by decomposing it into either its 1-to-all or all-to-1 components.

There are two major challenges arising in the context of operations such as gather, scatter, and reduce that transform messages through concatenation, split, or combine operations. To check whether a message has been created as a result of such a transformation, we need to send message hashes along with our replay messages. The checks are then performed as we go, exploiting homomorphic properties of the hash function as far as this is possible. Unfortunately, not all operands of such a transformation are necessarily stored in the trace because they might involve buffers that never appear in any communication.

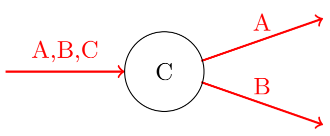
Figure 5 illustrates the problem for scatter. For example, we cannot tell whether the process shown in Figure 5a is the root of a scatter operation that intends to disseminate the vector A, B, C, D, E because C is only stored locally and as such never part of a communication that causes its message hash to be recorded in the trace. To make such information available to our analyzer, we introduced a new function that a user can insert into the program:

```
void Send_To_Self(void *buf, int count, MPI_Datatype datatype);
```

A call to this function records the missing information, about a buffer that a process utilizes locally, in the trace. Calling this function for C before doing both sends, makes C available in the trace. With this information, the analyzer can infer that A,B,C,D, and E belonged together before they were scattered across the processes. Without this information, however, no positive match can be made. A similar situation is shown in Figure 5b. Here, an inner node of a scatter tree receives a message, stores a part of it locally and forwards the rest to other processes. A message containing A, B and C is received but only A and B are sent further. Only with the hash of C written to the trace, the relationship becomes visible. Nevertheless, finding all related message parts requires testing of every combination of hashes. As this would be impractical for large numbers of messages, our detector checks only adjacent messages for split and concatenation. This is accomplished by looking at their starting addresses and MPI data types.



(a) Root process



(b) Non-root, non-leaf process

Fig. 5: Inputs and outputs of processes during scatter operations

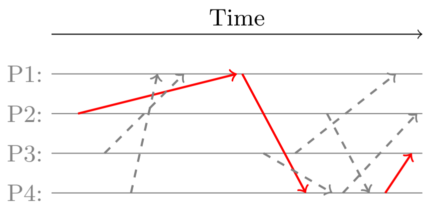
Currently, our prototype supports the fully automatic detection of broadcast and alltoall variants composed of multiple point-to-point broadcasts. Barriers are currently recognized as alltoall with an empty payload. Moreover, our prototype is capable of semi-automatically recognizing scatter, gather, and reduce with the help of the extra information described above. Reduce is restricted to some arithmetic and logical operations on certain datatypes. Floating-point arithmetics are not supported. We believe, that our collectives detector can be upgraded to eventually support all regular MPI collective operations. Irregular collectives, however, still present serious challenges due to their high degree of flexibility, which makes it hard to formulate manageable pre- and postconditions.

5 Evaluation

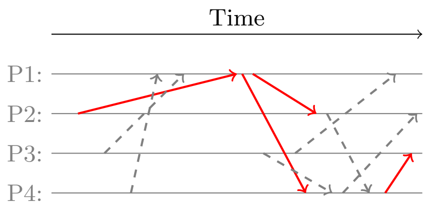
To demonstrate that our method can handle even very challenging cases, we apply it first to a set of micro-benchmarks that implement different variants of broadcast. A case study with the HPL benchmark shows the potential for actual performance improvements. We conducted all experiments on the IBM Blue Gene/P *Jugene* installation located at the Jülich Supercomputing Centre.

5.1 Microbenchmarks

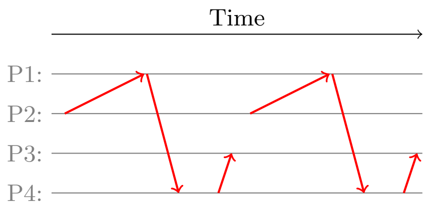
We start with a linear broadcast, as illustrated with solid arrows in Figure 6a. In a linear broadcast, a message visits one process after another until all processes have seen it. Thus, the communication is effectively serialized. In spite of the unrelated message traffic (i.e., the noise depicted as dashed arrows), the pattern is correctly identified. Adding a redundant message in Figure 6b offers two choices for the root process (P1 and P2). Both options are reported. Repeating the same broadcast twice as in Figure 6c results in the recommendation to replace all messages involved with a single predefined broadcast. Finally, we perform a nested broadcast by passing a token from the root to each other process (Figure 6d). Whoever owns the token initiates an inner broadcast. Both types of broadcasts are reported, although replacing the outer broadcast might change the order of the inner broadcasts. The ultimate decision is left to the user. If token-passing is extended to ring communication, our algorithm will report one instance for each process involved because each process could be a potential root.



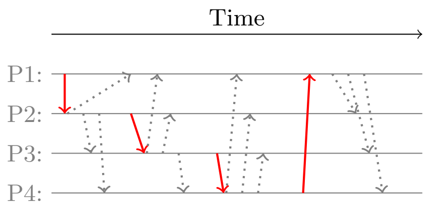
(a) Linear broadcast with noise



(b) Broadcast with a redundant message



(c) Repeated broadcast



(d) Nested broadcast with token passing

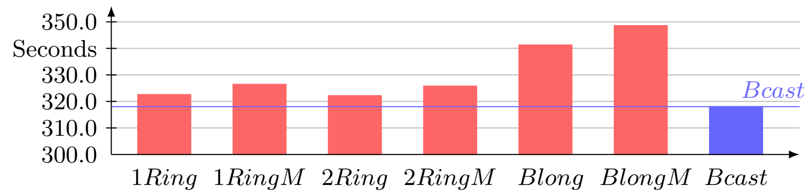
Fig. 6: Four test scenarios for broadcast. All solid arrows show messages with the same payload. Dashed arrows represent noise messages with different payloads. The dotted arrows illustrate correctly detected broadcasts.

5.2 High-Performance Linpack

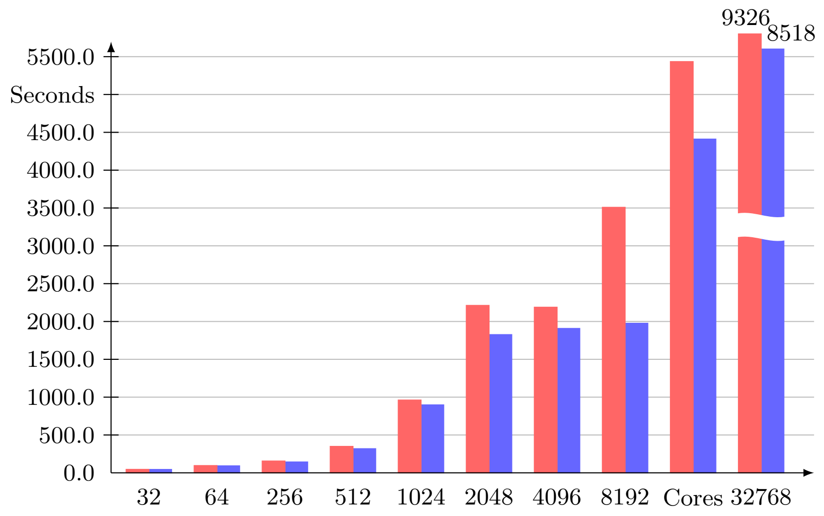
The High-Performance Linpack Benchmark [1] solves a dense linear system in double precision and is used to rank the world's fastest supercomputers in the Top500 list. We selected HPL as a test candidate because it makes heavy use of collective operations implemented via point-to-point communications. Prior to running this benchmark, the user needs to select in a configuration file one out of six hand-crafted broadcast implementations.

Regardless of whether the broadcasts are implemented with blocking or non-blocking semantics in any send mode, our collectives detector correctly reported all six HPL broadcast implementations including their source-code location plus some others that are presumably used for synchronization. Figure 7a compares the HPL execution time for any of the six broadcast options with the execution time after replacing the manual variant with MPI_Bcast(). The experiments reflect the performance for 256 cores and an input problem size of 32,000. Our results not only show significant performance differences among the six manual variants, but also show that the MPI broadcast always delivers superior performance with an overall improvement of up to 9%.

Although BLongM performed worst in this test with 256 cores, we chose it for the scaling study in Figure 7b because its bandwidth-optimized implementation reveals benefits for larger core counts. The number of matrix elements per core was kept at four million. Indeed, the difference between BlongM and MPI_Bcast() is most pronounced between 2k and 16k cores. Above, BLongM plays out its



(a) Comparison between the HPL-included broadcasts and MPI_Bcast



(b) Weak-scaling of HPL using BLongM (left) and MPI_Bcast (right)

Fig. 7: Total execution time of HPL with and without manual collectives

own strength. Overall, MPI_Bcast() was faster in all cases and with 8k cores the difference was even 44%. This demonstrates that the performance advantage of predefined collectives can be substantial and that the replacement of manual collectives is often worthwhile. In addition, using MPI_Bcast() in HPL makes several hundred lines of code obsolete, reducing its code complexity.

6 Related Work

In this section, we compare our approach with alternative routes taken earlier. Preissl et al. [11] pursue an almost identical objective. They not only attempt to find collective point-to-point exchanges but also to automatically replace their occurrences in the code with equivalent predefined MPI operations. Their solution is built around ROSE, a generator for source-to-source translators. They trace the messages sent during program execution, match the structure of the message graph with predefined structures representing collective communications, find the corresponding places in the code, and carry out the substitution.

The advantage of their solution is that the user does not have to do anything except running the tool to get—in the optimal case—better code. A disadvantage, however, is that no collective communication pattern can be matched except those already thought of and stored in the tool. While this might not seem to be very important at first, one has to consider that a user can implement a collective communication in any way that suits him and his particular application. For example, since different hardware topologies support different communication topologies, the precise shape of the communication pattern might be platform dependent. Moreover, since new network topologies are being created alongside new hardware, new patterns can emerge that are not thought of yet. While it is possible to add them to such a solution, the maintenance cost will rise with the number of possibilities. Also, if the number of possible patterns increases, the time to check against all of them will grow as well. At the time of publishing their work, the tool developed by Preissl et al. was able to detect only one simple broadcast implementation.

Whereas the previous approach and ours search for manual collectives in traces with dynamic content, di Martino et al. [3] attempt to detect them relying only on static information. They define collectives using mathematical abstractions and then use algebraic methods to find them. This methodology can provide a strong basis for matching potential candidates to models of communication patterns. However, PPAR, a prototypical tool based on this method, seems restricted in the range of patterns it can interact with. Just like Preissl et al., PPAR also tries to match known patterns of collective communications, only that PPAR does it via source-code analysis of the program.

7 Conclusion and Outlook

We proposed a practical method to detect manually-implemented collective operations in MPI programs written in any language without making any assumptions about the actual communication pattern. This eliminates the detection of false negatives, which sets our approach apart from earlier work in the field. However, there are still limitations which we want to address in the future.

Our prototype needs further work to evolve into a productive tool. As collective operations involving only a subset of the processes in a given communicator are not yet recognized, we plan to extend our method to also detect those cases and to suggest the creation of sub-communicators. Moreover, our current scheme allows fully automatic detection only for collectives such as broadcast and alltoall. For more sophisticated collectives, the user needs to supply extra information by inserting function calls into the program. Future versions could extend the coverage of our detector by also reporting matches that are incomplete to a certain degree, carefully balancing false negatives with false positives. Finally, to guide the user to the most promising replacement candidates in terms of potential performance improvements, we plan to simulate the effects of a replacement using a real-time replay of modified traces [6]. This would permit the user to compare the required effort with the benefit that is likely to materialize.

References

1. HPL – A portable implementation of the high-performance Linpack benchmark for distributed-memory computers. http://netlib.org/benchmark/hpl/.
2. Massimo Bernaschi, Giulio Iannello, and Mario Lauria. Efficient Implementation of Reduce-scatter in MPI. In *Parallel, Distributed and Network-based Processing, 2002. Proceedings. 10th Euromicro Workshop on*, pages 301–308, 2002.
3. Beniamino Di Martino, Antonio Mazzeo, Nicola Mazzocca, and Umberto Villano. Parallel program analysis and restructuring by detection of point-to-point interaction patterns and their transformation into collective communication constructs. *Science of Computer Programming*, 40:235–263, 2001.
4. Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.
5. Sergei Gorlatch. Send-Receive Considered Harmful: Myths and Realities of Message Passing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26:47–56, January 2004.
6. Marc-André Hermanns, Markus Geimer, Felix Wolf, and Brian J. N. Wylie. Verifying causality between distant performance phenomena in large-scale MPI applications. In *Proc. of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), Weimar, Germany*, pages 78–84. IEEE Computer Society, February 2009.
7. Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. Group Operation Assembly Language - A Flexible Way to Express Collective Communication. In *The 38th International Conference on Parallel Processing*. IEEE, September 2009.
8. Torsten Hoefler, Christian Siebert, and Wolfgang Rehm. A practically constant-time MPI Broadcast Algorithm for large-scale InfiniBand Clusters with Multicast. In *Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium*, pages 1–8. IEEE Computer Society, March 2007.
9. Sameer Kumar, Yogish Sabharwal, Rahul Garg, and Philip Heidelberger. Optimization of All-to-All Communication on the Blue Gene/L Supercomputer. In *Proc. of the 37th International Conference on Parallel Processing*, pages 320–329, Washington, DC, USA, 2008. IEEE Computer Society.
10. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2.* High Performance Computing Center Stuttgart (HLRS), 2009.
11. Robert Preissl, Martin Schulz, Dieter Kranzlmuller, Bronis R. de Supinski, and Daniel J. Quinlan. Transforming MPI Source code based on communication patterns. *Future Generation Computer Systems*, 26:147–154, May 2009.
12. Robert Ross, Robert Latham, William Gropp, Ewing Lusk, and Rajeev Thakur. Processing MPI Datatypes Outside MPI. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 42–53, Berlin, Heidelberg, 2009.
13. Peter Sanders and Jesper Larsson Träff. Parallel Prefix (Scan) Algorithms for MPI. In *Proc. of the Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting*, pages 49–57, 2006.
14. Jesper Larsson Träff, Andreas Ripke, Christian Siebert, Pavan Balaji, Rajeev Thakur, and William Gropp. A Pipelined Algorithm for Large, Irregular All-Gather Problems. *International Journal of High Performance Compututing Applications*, 24:58–68, February 2010.