Transforming Blocking MPI Collectives to Non-blocking and Persistent Operations

Hadia Ahmed*
Dept. of Computer Science
University of Alabama at Birmingham
hadia@uab.edu

Anthony Skjellum†
Dept. of Computer Science and Software Engineering
Auburn University
skjellum@auburn.edu

Purushotham Bangalore
Dept. of Computer Science
University of Alabama at Birmingham
puri@uab.edu

Peter Pirkelbauer‡
Dept. of Computer Science
University of Alabama at Birmingham
pirkelbauer@uab.edu

ABSTRACT

This paper describes Petal, a prototype tool that uses compiler-analysis techniques to automate code transformations to hide communication costs behind computation by replacing blocking MPI functions with corresponding nonblocking and persistent collective operations while maintaining legacy applications' correctness. In earlier work, we have already demonstrated Petal's ability to transform point-to-point MPI operations in complement to the results shown here. The contributions of this paper include the approach to collective operation transformations, a description of achieved functionality, examples of transformations, and demonstration of performance improvements obtained thus far on representative sample MPI programs. Depending on system scale and problem size, the transformations yield a speedup of up to a factor of two. This tool can be used to transform useful classes of new and legacy MPI programs to use the newest variants of MPI functions to improve performance without manual intervention for forthcoming HPC systems and updated versions of the MPI standard.

CCS CONCEPTS

• **Networks** → *Programming interfaces*; • **Software and its engineering** → Massively parallel systems; **Software maintenance tools**;

KEYWORDS

MPI, collective operations, blocking /nonblocking operations, communication-computation overlap, persistent operations

*Present address: Lawrence Berkeley National Laboratory, 1 Cyclotron Rd, Berkeley, CA 94720

†Present address: SimCenter, University of Tennessee, Chattanooga, TN; email: *tony-skjellum@utc.edu.*

‡Corresponding author.

EuroMPI/USA '17, September 25–28, 2017, Chicago, IL, USA

ACM ISBN 978-1-4503-4849-2/17/09…$15.00
https://doi.org/10.1145/3127024.3127033

ACM Reference Format:
Hadia Ahmed, Anthony Skjellum, Purushotham Bangalore, and Peter Pirkelbauer. 2017. Transforming Blocking MPI Collectives to Non-blocking and Persistent Operations. In *Proceedings of EuroMPI/USA '17, Chicago, IL, USA, September 25–28, 2017,* 11 pages.
https://doi.org/10.1145/3127024.3127033

1 INTRODUCTION

The Message Passing Interface (MPI) [18] defines a set of communication primitives for large-scale systems. Communication primitives between two processes are called *point-to-point operations*, whereas primitives where all processes on the same communicator participate are called *collective operations*. Collective operations simplify the development process by abstracting away implementation details of the precise communication call sequence and data transfer pattern among all processes in a group. In turn, a specific MPI implementation can offer optimized versions that take advantage of underlying hardware and network topology (at least in high quality MPI implementations) for achieving higher performance. Collective operations are considered to be an extremely important aspect of MPI functionality. They are heavily used in high performance computing (HPC) applications [21, 24, 27].

MPI-3.0 [19] added nonblocking MPI collectives. Under certain conditions (system, buffer sizes, and implementation quality), nonblocking collectives can increase communication-computation overlap and reduce synchronization delays. Many existing HPC applications built with blocking collectives are expected to improve in performance by migrating to nonblocking collective primitives. However, with the increasing complexity of HPC applications, manual code transformation is becoming difficult even for expert programmers. Typical HPC applications span more than 10,000 lines of codes (LOC) (*e.g.*, a quantum molecular dynamics code from LLNL has 47,000 LOC). There are many much larger "legacy MPI" codes and libraries as well.

Transforming such applications requires significant time and effort and is potentially error prone. For most applications, a simple textual replacement of blocking to nonblocking communication does not suffice to derive any significant benefit. To attain advantages of nonblocking features, developers have to identify independent parts in the code that can be interposed between a collective

call and _WAIT/_TEST (compatibly across the group of the MPI communicator) in order to expose computation that might overlap with the nonblocking collective. In order to simplify developers' work and to help ensure periodic introduction of new library features seamlessly into these applications, the goal of this work is to automate the process of changing collective operations from blocking to nonblocking as well as to the expected corresponding near-future persistent variants thereof. The work reported here is implemented as an extension to the Petal tool [3].

Petal is an automatic transformation tool, built on top of the ROSE [22] source-to-source translation infrastructure. With minimal user intervention, Petal analyzes code statically, improves the use of MPI by modifying the program's intermediate representation, and generates an improved version of the source code as output. Petal is able to change blocking to nonblocking calls and add persistent communication for point-to-point and collective MPI primitives. It uses static analysis techniques to identify potential communication-computation overlap in application source code while maintaining a program's correctness. It also detects fixed argument values for nonblocking operations and hence allows for the replacement of these operations with their corresponding persistent counterparts (a currently proposed feature of MPI-4).

This paper makes the following contributions: (1) analyses of MPI collective operations; (2) the transformation of blocking collectives to their nonblocking and persistent counterparts; and, (3) an evaluation on available benchmark codes.

The remainder of this paper is organized as follows. The next section discusses the importance of communication-computation overlap and persistent communication and also provides key information about the ROSE source-to-source infrastructure tool. Section 3 presents the design and implementation details for the Petal tool. Section 4 discusses experiments and results. Section 5 gives an overview of key related work. Finally, Section 6 concludes our study and explores pertinent future work.

2 BACKGROUND

Here, we introduce background information of MPI's collective communication and the ROSE compiler infrastructure.

2.1 MPI Communication

Since the introduction of supercomputers and their thousands upon thousands of processors, communication between processes distributed across these myriad processors is becoming an increasing burden on application performance. Communication incurs relatively more overhead than computation operations and is a pure cost of concurrency.

Many applications have been using the traditional collective operations. However, with the continuous progress towards more concurrency on next generation hardware, Exascale, the additional cost of data communication using the blocking mode is expected to incur increasingly significant overhead. One way to mitigate this problem is by overlapping communication and computation operations using nonblocking operations. To overlap means that the MPI library is able to initiate a particular communication and then the application can do some other independent computations while the MPI library makes progress on the message passing operation

```
1  MPI_Request req;
   MPI_Status stat;
3  MPI_Iallreduce(buf1, rbuf1, SIZE, MPI_INT,MPI_SUM, MPI_COMM_WORLD, &req);
   /* independent computation*/
5  MPI_Wait(&req, &stat);
```

Figure 1: MPI-3 Nonblocking Collective Example

```
1  MPI_Request req;
   MPI_Status stat;
3  MPI_Info info;
   MPIX_Iallreduce_init(buf1, rbuf1, SIZE, MPI_INT, MPI_SUM, MPI_COMM_WORLD, info, &req);
5  for(i=1;i<BIGNUM;i++) {
       MPIX_Start(MPI_COMM_WORLD,&req);
7      /* independent computation*/
       MPI_Wait(&req, &stat);
9  }
   MPI_Request_free(&req);
```

Figure 2: Proposed MPI Persistent Collective Example

independent of the computation. With MPI-3.0 [19], nonblocking collectives were added to the standard. They follow the same design as the point-to-point ones. Figure 1 shows an example of a nonblocking collective operation. It shows that nonblocking separates initiation and completion of communication. Hence, application can overlap independent computation with communication. The advantages of using nonblocking collectives have been described, for instance, by Hoefler *et al.* [14, 15, 30].

Persistent point-to-point communication (half-channels of senders or receivers) were first introduced in MPI-1 [18]. They provide a means to reduce the critical path on one end of a communication when only send-buffer contents change over a set of repetitious calls, a property common to many data-parallel applications. A fixed cost _INIT operation reduces the variable cost of each reuse, and, ideally, an initialization may lockdown resources or perform other steps that reduce the critical path during reuse. Currently, some of the authors and others have introduced persistent nonblocking collective operations as a proposed addition to MPI-4 [1, 17]. These operations provide a one-to-one correspondence with MPI-3 nonblocking collective operations (traditional and sparse functions). As such, they provide a one-to-one mapping of either blocking or nonblocking collectives that may be used to improve performance and/or predictability. They consequently comprise one of the transformation targets. In order for a persistent collective transformation to be net performance-enhancing, it must be used many times. In such settings, an _INIT operation (a collective operation) outside the critical path enables MPI to make an efficient algorithmic choice for the subsequent operations that defrays that fixed cost of initialization. Figure 2 shows an example of the proposed persistent collective operation.

2.2 ROSE compiler infrastructure

The ROSE source-to-source translation infrastructure is under active development at the Lawrence Livermore National Laboratory (LLNL) [22]. ROSE provides front ends for many languages, including C/C++, FORTRAN 77/95/2003, and Java. ROSE also supports several parallel extensions, such as OpenMP and CUDA. ROSE generates an Abstract Syntax Tree (AST) for the source code. The ASTs

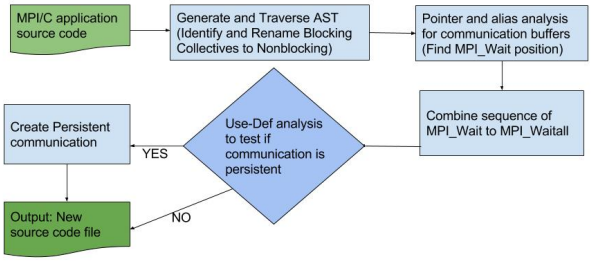


Figure 3: Petal workflow

are uniformly built for all input languages. ROSE offers many specific analyses (*e.g.,* pointer alias analysis) and makes these available through an API. Users can write their own analyses by utilizing frameworks that ROSE provides. With the representation of the code as an AST and using the static analysis provided from the ROSE libraries, one can explore the code and determine how to improve it by looking for certain code style, inserting new code, changing and/or removing old code, hence generating modified source code while preserving the semantics of the original code.

3 PETAL DESIGN AND PROTOTYPE

We have designed and prototyped Petal, a transformation tool based on the ROSE source-to-source infrastructure. Using ROSE's analysis APIs, Petal transforms calls to blocking collectives primitives to equivalent nonblocking and forthcoming (pre-standard) persistent communication primitives in a semi-automatic way with minimal user intervention.

3.1 Design of Petal

Petal's workflow is shown in Figure 3. Petal reads an application's source code file(s) and produces transformed nonblocking/persistent source code file(s). It consists of three main analysis components:

- Count the number of calls to blocking collectives within each block. The count is needed to allocate the maximum number of MPI request and status objects that are needed and reuse the same request for communications on mutually exclusive paths.
- Pointer alias analysis is used to identify the code location where an `MPI_Wait` can be placed. In addition, multiple adjacent wait calls are merged into a single `MPI_Waitall`.
- If the MPI collective operations are located within a loop, we use use-def analysis [31] to check if all MPI function arguments' definitions except the input buffer values are loop invariant. In such case, the nonblocking communication is replaced by calls to the persistent communication primitives.

3.2 Preprocessing

The first step in analyzing source code is through ROSE compiling and generating an intermediate representation in the form of an AST from the input file(s). Before we analyze the code, we also inline all functions where implementations are available. Function inlining serves two purposes: (1) inlining increases the code region after an MPI communication has been initiated. This allows us to move the corresponding `MPI_Wait` call further away from the initiation call, thereby creating more potential overlap between computation and communication; (2) inlining simplifies source code analysis, as it eliminates the need to use inter-procedural analysis.

3.3 Counting MPI Function Calls

After inlining, the first analysis is to count the number of MPI communication calls in a source code region. The count is needed so we can create a sufficient amount of MPI request and status objects. We use the CodeThorn [23] framework to build a data flow analysis (DFA) that counts and keeps track of the MPI blocking functions locations. CodeThorn is an analysis tool built in ROSE with the goal of providing approaches for combining static DFA with model checking verification methods. We defined our own DFA that does a forward analysis to pass information of how many blocking functions are found so far and to assign an index subscript number to each of these found functions. For MPI calls in sequence, Petal just increments the number of calls. For selection statements (*i.e.,* branches such as if and switch-case), Petal takes the maximum among all branches. As shown in Figure 4, our approach allows two mutually exclusive MPI communication calls share the same request object. This reduces the number of MPI request/status objects.

```
MPI_Request req[1];
MPI_Status stat[1];

if (cond)
  MPI_Ibcast(..,&req[0]);
else
  MPI_Ibcast(..,&req[0]);

MPI_Wait(&req[0], &stat[0]);
```

Figure 4: Sharing of request and status objects in mutually exclusive code paths

Once Petal has determined the total number of MPI calls, it allocates the necessary request and status objects in the form of two arrays at the beginning of the main function; one for `MPI_Request` which is set to `MPI_REQUEST_NULL`, and the other for `MPI_Status`.

Finally, Petal transforms the blocking collective calls to nonblocking collective primitives. To this end, it changes the callee name of function calls to the corresponding nonblocking function. In addition, it appends the MPI request object to the argument list. The request handle is an element from the request array with the index subscript assigned by DFA.

3.4 Nonblocking using Pointer alias analysis

The next step is concerned with finding a position for inserting `MPI_Wait` that allows for sufficient communication-computation overlap. The more computation we can overlap with communication, the better we can hide costs associated with communication. MPI communication calls take one (or two) pointers to message buffers as arguments. Petal takes advantage of this fact and uses pointer alias analysis to find code that introduces dependencies. ROSE's pointer alias analysis implements Steensgaard's algorithm, which has linear time complexity [26]. This allows our tool to scale well with large applications. One restriction for using this algorithm

is that it treats any access to a part of the array as an access to the whole array [26]. This might lead to placing the `MPI_Wait` in overly conservative positions in some applications.

The idea is to check the use of a buffer or its aliases along the control flow edges. Figure 5 shows a snippet of the iterative code for solving $Ax = b$ using the conjugate gradient method [2]. It shows the original version using blocking collectives. Petal applies Steensgard's analysis to check for a buffer or its aliases' use along control flow edges. We start with the collective call. For each statement that follows a collective MPI call, Petal collects read and write variables. If the buffer or its aliases is among the write variables then the statement is considered unsafe. The `MPI_Wait` that concludes the collective call has to be inserted before it. For a read variable, this requirement can be relaxed under certain conditions. If the buffer is used only for sending, a read does not introduce an unsafe dependency and the `MPI_Wait` can be moved across it. However, if it is a receive buffer, then the read introduces a true dependency and the `MPI_Wait` has to be inserted before it.

Figure 6 shows Petal's output for this code. Petal detected that the `MPI_Allgather` for gathering the new vector `x` is independent from computing `Hvector` and its vector dot product with the residue vector. The collective call is also independent from aggregating `Delta1`. Hence, its corresponding `MPI_Wait` can be placed at the end of the loop. Meanwhile an argument to the `MPI_Iallreduce` message is used directly after the call, hence no independent computation is identified. In this call, `&Delta1` serves as the receive buffer. The computation in Line 18 reads from `Delta1` and hence cannot be started before `Delta1` has been computed.

```
1  /* Definitions of variables used
     * Allocate memory for vectors */
3  do {
     ...
5    iter++;
     for(index = 0; index < Bloc_VectorSize; index++){
7      Bloc_Vector_X[index] = Vector_X[MyRank*Bloc_VectorSize + index] + Tau*
           Bloc_Direction_Vector[index];
       Bloc_Residue_Vector[index] = Bloc_Residue_Vector[index] + Tau*Buffer[index];
9    }
     MPI_Allgather(Bloc_Vector_X, Bloc_VectorSize, MPI_DOUBLE, Vector_X, Bloc_VectorSize,
           MPI_DOUBLE, MPI_COMM_WORLD);
11   SolvePrecondMatrix(Bloc_Precond_Matrix, Bloc_HVector, Bloc_Residue_Vector,
           Bloc_VectorSize);
     Bloc_Delta1 = ComputeVectorDotProduct(Bloc_Residue_Vector, Bloc_HVector, Bloc_VectorSize
           );
13   MPI_Allreduce(&Bloc_Delta1, &Delta1, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
     Beta = Delta1 / Delta0;
15   Delta0 = Delta1;
     for (i=0; i < Bloc_VectorSize; i++) {
17     Bloc_Direction_Vector[i] = -Bloc_HVector[i] + Beta*Bloc_Direction_Vector[i];
     }
19 } while (Delta0 > EPSILON && iter < MAX_ITERATIONS);
```

Figure 5: Blocking version of Conjugate Gradient Computation

Petal takes into account that most of the collective operations have two buffers involved in the communication. For some operations, such as `MPI_Reduce, MPI_Scatter`, one of the buffers is only effective at the root process and is not part of the communication for other processes. In `MPI_Reduce`, the receive buffer is only of importance for the root rank, while in `MPI_Scatter` it is

```
1  MPI_Request reqs[2];
   int flags[2];
3  MPI_Status stats[2];
   memset(flags,0,(sizeof(flags)));
5  do {
     ...
7    iter++;
     for(index = 0; index < Bloc_VectorSize; index++){
9      Bloc_Vector_X[index] = Vector_X[MyRank*Bloc_VectorSize + index] + Tau*
           Bloc_Direction_Vector[index];
       Bloc_Residue_Vector[index] = Bloc_Residue_Vector[index] + Tau*Buffer[index];
11   }
     MPI_Iallgather(((void *)Bloc_Vector_X), Bloc_VectorSize, MPI_DOUBLE, ((void *)Vector_X),
           Bloc_VectorSize, MPI_DOUBLE, MPI_COMM_WORLD, &reqs[0]);
13   /*Inline SolvePrecondMatrix*/
     /*Inline ComputeVectorDotProduct*/
15   MPI_Iallreduce(((void *)(&Bloc_Delta1)),((void *)(&Delta1)),1,MPI_DOUBLE,MPI_SUM,
           MPI_COMM_WORLD,&reqs[1]);
     MPI_Wait(&reqs[1],&stats[1]);
17   Beta = Delta1 / Delta0;
     Delta0 = Delta1;
19   for (i = 0; i < Bloc_VectorSize; i++) {
       Bloc_Direction_Vector[i] = -Bloc_HVector[i] + Beta * Bloc_Direction_Vector[i];
21   }
     MPI_Wait(&reqs[0],&stats[0]);
23 } while (Delta0 > 1.0E-20 && iter < 10000);
```

Figure 6: Transformed, nonblocking version of Conjugate Gradient Computation

the send buffer. Figure 7 shows an example where the best location for `MPI_Wait` depends on whether the process is the root or not. Petal computes the `MPI_Wait`'s position for both root rank and other ranks. If the computed position is the same, Petal just uses `MPI_Wait`. If the positions are different, it inserts `MPI_Wait` at both positions guarded by the condition testing the rank to MPI process. In some cases, these allows non-root ranks to proceed with computation, while the root has to wait for the completion of scatter or reduce as seen in Figure 8.

If a function call cannot be inlined, due to it its implementation not being available, Petal analyzes the calls' arguments. If one of them relates to the message buffer in question, Petal treats this call as unsafe. Otherwise, the call is considered independent and can be overlapped safely with communication. The exception of this rule is calls to other MPI functions. Since MPI is a standard specification, Petal understands its API and arguments. Petal has stored information for each MPI function and whether an argument is read or written. Using these information as part of the message buffer analysis, Petal decides whether concurrent communications is feasible or not. If there is no dependency between all the communication message buffers, (*i.e.*, only sending buffers can be identical), communication calls can overlap as well. Figure 9 shows an example of this case. By looking at the message buffer of the three consecutive

```
1  float nums[size];
   if (rank == root) {
3      //insert some values in nums
   }
5  float sub_nums[len];
   MPI_Scatter(nums, len, MPI_FLOAT,sub_nums, len, MPI_FLOAT, root, MPI_COMM_WORLD);
7  nums = do_computation();
```

Figure 7: Example of Different Positions for Same MPI_Wait

```
float nums[size];
if (rank == root) {
    //insert some values in nums
}
float sub_nums[len];

MPI_Iscatter(nums, len, MPI_FLOAT, sub_nums, len,
    MPI_FLOAT, root, MPI_COMM_WORLD, &reqs[0]);
if (rank == root) {
  MPI_Wait(&reqs[0],&stats[0]);
}

//nums = do_computation() inlined

if (rank != root) {
  MPI_Wait(&reqs[0],&stats[0]);
}
```

Figure 8: Transformed Output for Code in Figure 7

```
MPI_Bcast(&mass, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&nrho, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&drho, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Figure 9: Blocking Consecutive Communication

`MPI_Bcast`, (mass, nrho, drho), Petal is able to detect that they are all distinct and hence can be safely overlapped. Figure 10 shows the transformation's output.

```
MPI_Ibcast(&mass, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD,&reqs[0]);
MPI_Ibcast(&nrho, 1, MPI_INT, 0, MPI_COMM_WORLD,&reqs[1]);
MPI_Ibcast(&drho, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD,&reqs[2]);
MPI_Waitall(3,reqs,stats);
```

Figure 10: Nonblocking Concurrent Communication

Petal considers `MPI_Wtime`, `MPI_Barrier`, and `MPI_Finalize` as special operations that cannot be overlapped with communication. Hence, any communication that has been initiated before a call to such function must be completed before these special operations. Petal inserts `MPI_Wait` just before any of these special operations.

To ensure that the MPI Wait in its new position gets executed only if its corresponding nonblocking call is executed, Petal checks whether the nonblocking initiation and wait for completion are in the same scope. If they are in different scopes, Petal uses a flag that is set to true after initiating a nonblocking call. The corresponding `MPI_Wait` is guarded with the condition that the corresponding flag is `true`. After wait's execution, the flag is reset. This guarantees that the wait is executed if and only if the initiation call is executed.

Merging adjacent wait calls: Once all the `MPI_Wait` positions are identified, Petal checks if they all are to be placed in the same position. If multiple wait calls are using a set of contiguous request objects, Petal replaces multiple calls with a single call to `MPI_Waitall` provided that no other communication is active simultaneously. Petal ensures they are contiguous by checking the request's subscript values passed in the set with wait positions. If the difference between maximum and minimum+1 subscript values is not equal to the number of identified wait calls, Petal decides that there is at least one communication that is active but has a different wait position and opts to insert each wait call individually.

3.5 Persistent Mode using Use-Def Analysis

When a nonblocking MPI routine is called repeatedly with the same argument list, MPI creates the same request object many times. Similarly, when the communication is repeatedly completed, MPI needs to repeatedly free the request object. This leads to overhead incurred for every single send or receive. Persistent communication primitives were introduced to avoid the redundancy of creating the message request each time it is needed in a communication. Persistent operations allow the request object to persist, and be available for reuse after a `MPI_Wait` call. The request object is created once (using `_INIT`), then the application can initiate and complete the communication as many times as needed (using `MPI_Start` and `MPI_Wait`), and then at the end it can free the request object (using `MPI_Request_free`), when it is no longer needed or when one of the arguments, except the input buffer values has changed.

Being able to detect and transform to persistent communication is useful in improving communication performance between parallel tasks. In particular, applications where processes exchange information with the same partners (*e.g.,* exchange boundary region), as well as general collectives that have fixed parameters. These applications can benefit from using persistent communications because MPI will be able to choose the best operation once, lock-down resources at initialization, and reduce the critical path of operations on each actual communication instance. Persistent collectives will often be much faster than non-persistent collectives provided they are well implemented. Results thus far are positive with much additional headroom for improvement [17]. In order to determine if it is legal to perform such optimization, Petal uses data flow analysis to statically detect if the arguments associated with the nonblocking communication change inside the loop.

Petal generates use-def pairs for each argument in a communication call. These data describe the potential reaching definitions for this argument's specific use. Petal checks whether any of these definition is located within the loop. If all definitions are outside of the loop, the communication call can be safely changed to persistent. If at least one of the definitions is dependent on the loop variable or if it is found inside the loop, the communication is considered not persistent. The exception to this is the message buffer data where the data itself is expected to change.

Once a communication is identified as persistent, Petal replaces this function call with three corresponding function calls. First, `_INIT` call is created with nonblocking arguments and inserted before the loop to create a persistent request handle. Next, replace nonblocking call with `MPI_Start` with request handle and an `MPI_Info` handle as its arguments to perform prescribed communication operation. At last, directly after the end of the loop, add `MPI_Request_free` to explicitly deallocate the request object.

Since conjugate gradient is an iterative algorithm, the computation process is repeated until a result below a certain threshold is obtained. Hence it can be observed that persistent communication is suitable in this situation. Figure 11 shows transformed code using persistent communication. Petal found that the argument list for both `MPI_Allgather` and `MPI_Allreduce` are defined outside the loop and the message buffer addresses are fixed, only their data is changing over iterations, Petal created the persistent requests and replaced their usage accordingly.

```
MPI_Request reqs[2];
int flags[2];
MPI_Status stats[2];
MPI_Info infos[2];
memset(flags,0,(sizeof(flags)));
...
MPIX_Iallgather_init(((void *)Bloc_Vector_X), Bloc_VectorSize, MPI_DOUBLE, ((void *)Vector_X)
        , Bloc_VectorSize, MPI_DOUBLE, MPI_COMM_WORLD, infos[0], &reqs[0]);
MPIX_Iallreduce_init(((void *)(&Bloc_Delta1)),((void *)(&Delta1)),1,MPI_DOUBLE,MPI_SUM,
        MPI_COMM_WORLD,infos[1], &reqs[1]);
do {
    ...
  iter++;
  for(index = 0; index < Bloc_VectorSize; index++){
    Bloc_Vector_X[index] = Vector_X[MyRank*Bloc_VectorSize + index] + Tau*
            Bloc_Direction_Vector[index];
    Bloc_Residue_Vector[index] = Bloc_Residue_Vector[index] + Tau*Buffer[index];
  }
  MPIX_Start(MPI_COMM_WORLD,&reqs[0]);
  /*Inline SolvePrecondMatrix*/
  /*Inline ComputeVectorDotProduct*/
  MPIX_Start(MPI_COMM_WORLD,&reqs[1]);
  MPI_Wait(&reqs[1],&stats[1]);
  Beta = Delta1 / Delta0;
  Delta0 = Delta1;
  for (i = 0; i < Bloc_VectorSize; i++) {
    Bloc_Direction_Vector[i] = −Bloc_HVector[i] + Beta * Bloc_Direction_Vector[i];
  }
  MPI_Wait(&reqs[0],&stats[0]);
} while (Delta0 > 1.0E−20 && iter < 10000);
MPI_Request_free(&reqs[0]);
MPI_Request_free(&reqs[1]);
```

Figure 11: Transformed, persistent version of Conjugate Gradient Computation

4 EXPERIMENTAL RESULTS

In this section, we evaluate the effectiveness of Petal's transformation against three HPC applications that uses MPI blocking collectives. We compare the performance of the original code against the performance obtained by transformed code using nonblocking collectives.

Experimental Environment: For persistent collective experiments we used the UAB Cheaha cluster. It consists of 96 compute nodes each with two Intel Xeon E5-2680 v3 2.5 GHz processors (12 cores each) with at least 128GB RAM. The nodes are interconnected with FDR InfiniBand. The MPI version used is openmpi 4.0.0a1 with the default settings.

The rest of experiments were conducted on LLNL cab cluster. It has 1,296 nodes with 16 core/node and 32GB memory/node with Intel Xeon E5-2670. The nodes are interconnected by InfiniBand in QDR mode. The MPI version used is openmpi-gnu-2.0.0 with the default settings.

4.1 Evaluation

Our first test uses a C version of a benchmark code discussed by James Tullos [29]. In this example, the kernel operates on three arrays that are distributed across the processes. The code is shown in Figure 12. It uses `MPI_Allreduce` to compute the average of the first array. Then the code uses the average to compute new entries for the second array. The next step computes the global minimum and maximum values of the second array by two `MPI_Allreduce` operations. The last step adds the average of the first array to all elements in the second array and uses the global min and max values

```
MPI_Allreduce(sub_arr1, sumA1temp, num_elements, MPI_DOUBLE, MPI_SUM,
        MPI_COMM_WORLD);
sumA1 = sum(sumA1temp);
avgA1 = sumA1/(num_elements*p);
sub_arr2 = scalarMultiply(sub_arr2,avgA1);
MPI_Allreduce(sub_arr2, minA2temp, num_elements, MPI_DOUBLE, MPI_MIN,
        MPI_COMM_WORLD);
MPI_Allreduce(sub_arr2, maxA2temp, num_elements, MPI_DOUBLE, MPI_MAX,
        MPI_COMM_WORLD);
sub_arr3 = scalarAdd(sub_arr3,avgA1);
minA2 = min(minA2temp);
maxA2 = max(maxA2temp);
sub_arr3 = scalarMultiply(sub_arr3,(minA2+maxA2)*0.5);
MPI_Allreduce(sub_arr3, sumA3temp, num_elements, MPI_DOUBLE, MPI_SUM,
        MPI_COMM_WORLD);
finalsum = sum(sumA3temp);
```

Figure 12: Blocking C version of Intel Benchmark

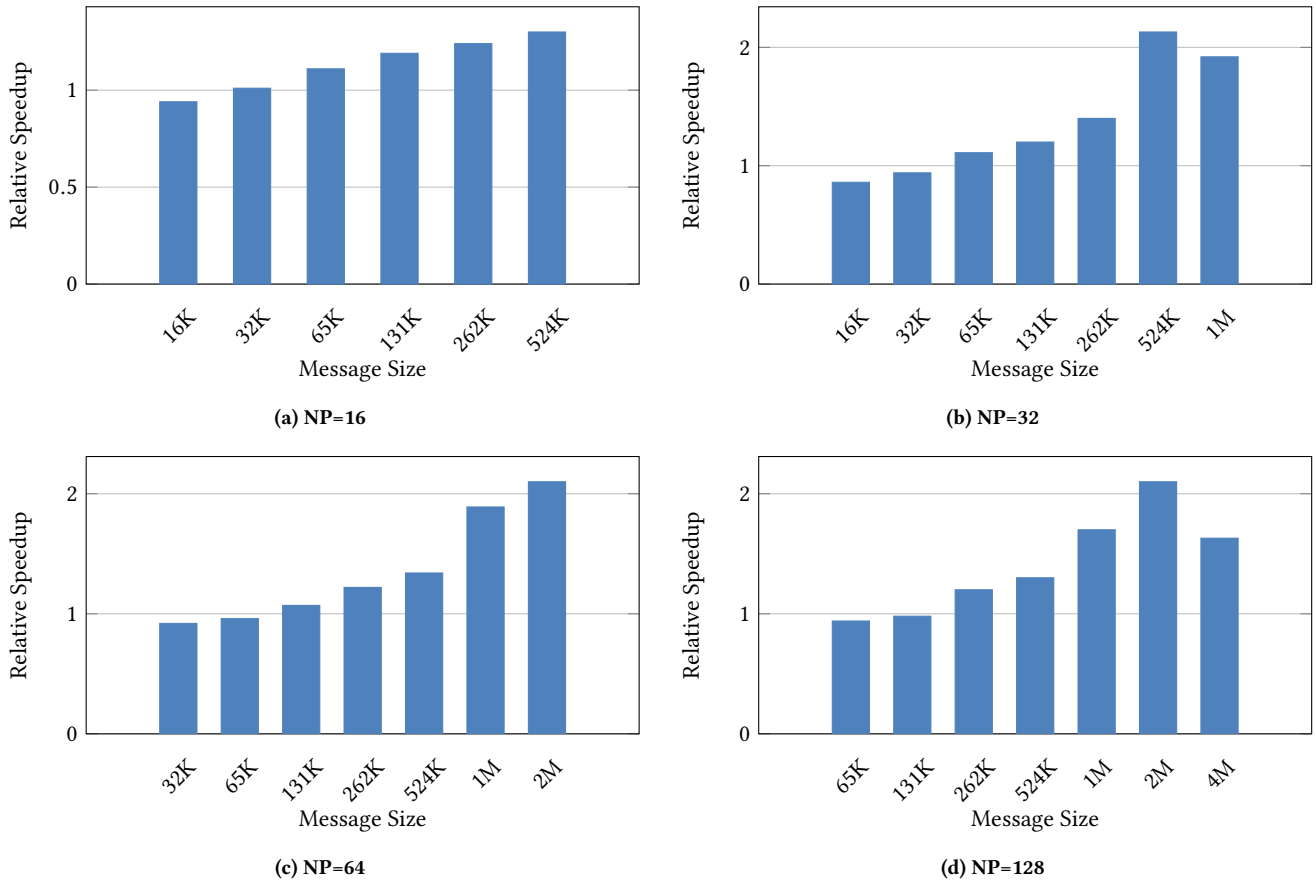
to update the entries of the third array. A last call to `MPI_Allreduce` computes the sum of the third array.In this example, Petal overlaps the `MPI_Allreduce` calls that compute the min and max values of the second array with the first part of computing the third array (the average of the first array can be added to the third array without the need for min and max values). In addition, the communication of max can be overlapped with processing the results of finding the min value. The transformed code is illustrated in Figure 13.

We compared the run-time of the two versions using weak scaling. We varied the number of MPI processes from 16 to 128, and tested with different message sizes, ranging from 16K to 524K with 16 processes, and from 65K to 4M with 128 processes.

Figure 14 shows the relative speedup of the nonblocking version obtained over the original code. With 32 processes, we see improvements when the message size exceeds 32K. However with increasing number of processes, the message threshold increases to 64K, resp. 131K and 262K, before we see positive effects from nonblocking collectives. This could be due to the overlapped computation which takes a relative short time compared to communication. In the best case scenarios, the nonblocking primitives achieve a relative

```
MPI_Request reqs[4];
int flags[4];
MPI_Status stats[4];
MPI_Iallreduce(sub_arr1,sumA1temp,num_elements,MPI_DOUBLE,MPI_SUM,
        MPI_COMM_WORLD,&reqs[0]);
MPI_Wait(&reqs[0],&stats[0]);
sumA1 = sum(sumA1temp);
avgA1 = sumA1/(num_elements*p);
sub_arr2 = scalarMultiply(sub_arr2,avgA1);
MPI_Iallreduce(sub_arr2,minA2temp,num_elements,MPI_DOUBLE,MPI_MIN,
        MPI_COMM_WORLD,&reqs[1]);
MPI_Iallreduce(sub_arr2,maxA2temp,num_elements,MPI_DOUBLE,MPI_MAX,
        MPI_COMM_WORLD,&reqs[2]);
sub_arr3 = scalarAdd(sub_arr3,avgA1);
MPI_Wait(&reqs[1],&stats[1]);
minA2 = min(minA2temp);
MPI_Wait(&reqs[2],&stats[2]);
maxA2 = max(maxA2temp);
sub_arr3 = scalarMultiply(sub_arr3,(minA2+maxA2)*0.5);
MPI_Iallreduce(sub_arr3,sumA3temp,num_elements,MPI_DOUBLE,MPI_SUM,
        MPI_COMM_WORLD,&reqs[3]);
MPI_Wait(&reqs[3],&stats[3]);
finalsum = sum(sumA3temp);
```

Figure 13: Petal-generated Output for Code in Figure 12



(a) NP=16

(b) NP=32

(c) NP=64

(d) NP=128

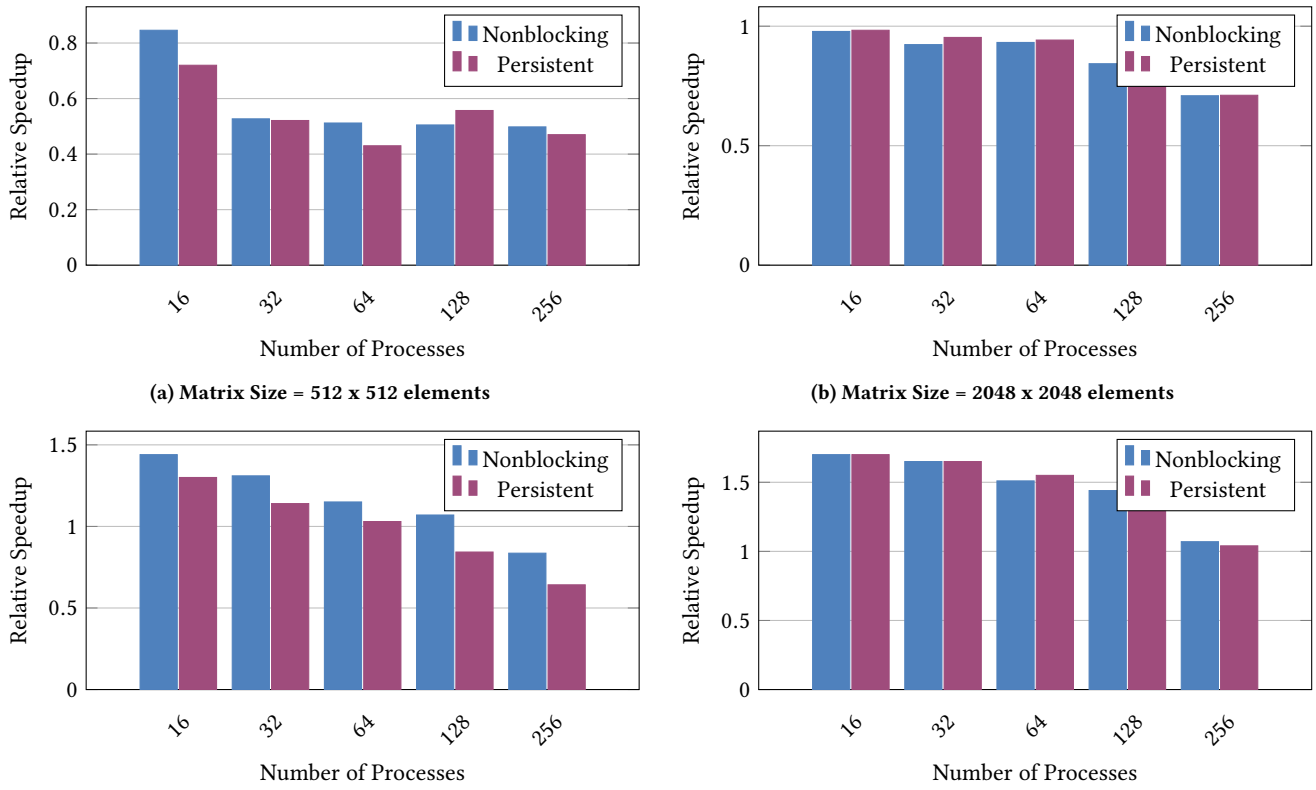
Figure 14: Weak Scaling for MPI_Allreduce Example.

speedup of 2x over corresponding blocking primitives. We observed that sometimes we see more than 2x which is the theoretical possible improvement. One of the causes can be related to cache usage. We plan to perform in-depth performance analysis for these cases.

Another example we used is the conjugate gradient code discussed in sections 3.4 and 3.5. In this example, Petal was able to identify independent work to overlap with `MPI_Allgather` communication. Figure 15 shows the relative speedup of running both the nonblocking and nonblocking-persistent versions of the transformed conjugate gradient application, against original blocking version, while varying the size of the matrix (and vector) and number of MPI processes. Again, it is clear that as we increase the problem size the nonblocking version outperforms the blocking version. However, when we increase the number of processes, the effect of nonblocking improvements decreases. This is because, as we increase the number of processes, the number of elements exchanged decreases. In addition, at really large matrix size our conjugate gradient code for blocking collectives did not complete in the allocated time on the machine while the nonblocking finished successfully. With persistent we did not see an improvement over nonblocking counterparts with respect to speedup.

Another example for persistent transformation is the game of life. The original nonblocking and Petal's generated persistent source code is depicted in Figure 16. Petal was successful in statically detecting the persistent mode for both `MPI_Ineighbor_alltoallw` and `MPI_Iallreduce`. In both function calls, none of their arguments are redefined inside the loop. Hence, they are considered persistent. Figure 17 shows the total runtime for the application for both the nonblocking and persistent communication mode. Like in the previous example, both achieved similar runtime results. While we did not see any runtime improvement for using persistent, we have to note that these experiments were done on initial version of persistent collectives with no optimizations. In addition, persistent is proven to reduce communication latency [17] and we expect newer versions to provide better performance results.

While benefits from using nonblocking collectives depend on the message size and amount of work to overlap, another benefit can be achieved from running multiple communications simultaneously. We tested that using miniMD; a molecular dynamics microapplication in the Mantevo project at Sandia National Laboratories [13]. To compute force between atoms using an EAM interactions algorithm, the application uses multiple independent communications to pass the same information regardless of problem size. Petal changed these communications from blocking to nonblocking. For example, Petal replaced six adjacent calls to `MPI_Bcast` with their nonblocking counterparts and added a single `MPI_Waitall` at the end. This



(a) Matrix Size = 512 x 512 elements

(b) Matrix Size = 2048 x 2048 elements

(c) Matrix Size = 4096 x 4096 elements

(d) Matrix Size = 8192 x 8192 elements

Figure 15: Conjugate Gradient Results.

```
for (k = 0; k < NTIMES && gflag != 0; k += 2) {
    gflag = 0;
    // start boundary exchange
    MPI_Ineighbor_alltoallw(&life[0][0], sendcounts, sdispls, sendtypes,
        &life[0][0], recvcounts, rdispls, recvtypes, commgraph, &reqs[0]);
    // compute inner cells
    flag = compute_inner(life, temp, myM, myN);

    // wait for boundary exchange to complete
    MPI_Wait(&reqs[0], &stats[0]);

    // compute boundary cells
    flag += compute_boundary(life, temp, myM, myN);

    MPI_Iallreduce(&flag, &gflag, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD, &reqs[1]);
    MPI_Wait(&reqs[1], &stats[1]);
    if (gflag != 0) {...}
}
```

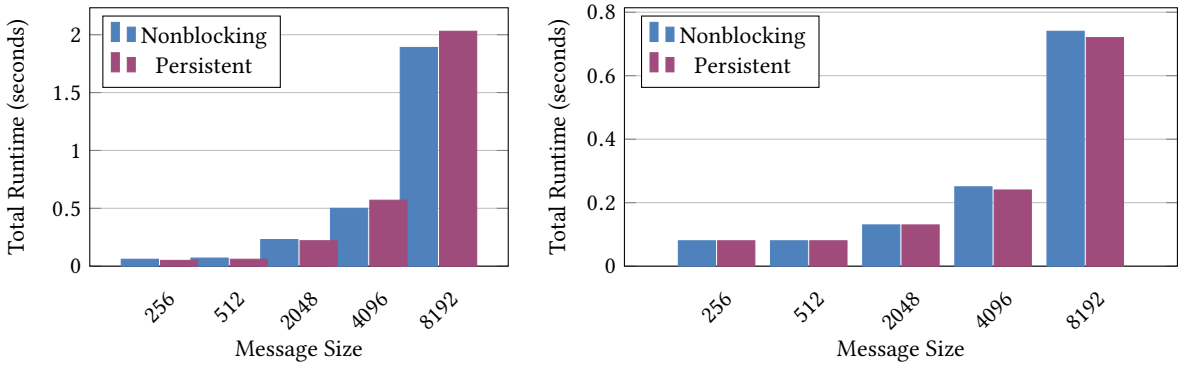
(a) Nonblocking

```
MPIX_Ineighbor_alltoallw_init(&life[0][0], sendcounts, sdispls, sendtypes,
    &life[0][0], recvcounts, rdispls, recvtypes, commgraph, info[0], &reqs[0]);
MPIX_Iallreduce_init(&flag, &gflag, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD, info[1], &reqs[1]);

for (k = 0; k < NTIMES && gflag != 0; k += 2) {
    gflag = 0;
    // start boundary exchange
    MPIX_Start(commgraph, &reqs[0]);

    // compute inner cells
    flag = compute_inner(life, temp, myM, myN);

    // wait for boundary exchange to complete
    MPI_Wait(&reqs[0], &stats[0]);

    // compute boundary cells
    flag += compute_boundary(life, temp, myM, myN);

    MPIX_Start(MPI_COMM_WORLD, &reqs[1]);
    MPI_Wait(&reqs[1], &stats[1]);
    if (gflag != 0) {...}
}
MPI_Request_free(&reqs[0]);
MPI_Request_free(&reqs[1]);
```

(b) Persistent

Figure 16: Game of Life

transformation allows for communication-communication overlap. Figure 18 shows the effect of this transformation on total communication time in the application. In this case we saw 1.5x speedup from switching to nonblocking collective in total communication time. However, for total runtime of application, the improvement was negligible. These results suggest that even though we can



(a) Number of Processes = 64

(b) Number of Processes = 256

Figure 17: Game of Life Runtime.

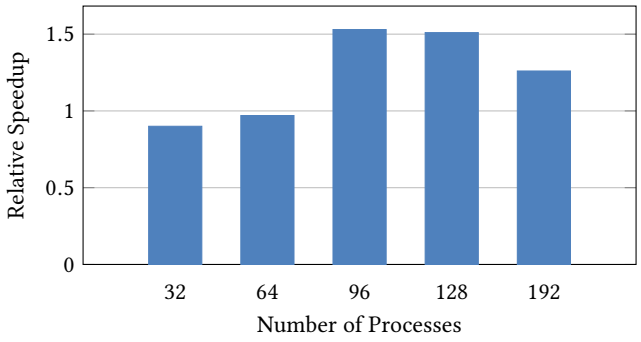


Figure 18: miniMD Results.

see improvements on communication side for using nonblocking collectives with fixed message size, additional refactoring on the computation side is still needed to improve the overall application performance.

```
1  MPI_Allreduce(&atom.nlocal, &natoms, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

3  evflag = 1;
   compute(atom, neighbor, comm, me);
```

Figure 19: MiniMD's LJ communication code

In addition to the improvements to `MPI_Bcast` for the EAM interaction code, Petal was also able to modify the MiniMD's LJ interaction code, shown in Figure 19. Petal was able to detect that a reduction on the number of atoms can be overlapped with the LJ computation operation. The overlap proved possible, because the computation only reads the send buffer (`atom`) and does not access the receive buffer (`natoms`). However, this transformation also did not result in any significant improvement in either the measured runtime or communication time of the application. One reason why the transformation does not yield better speedup is that the reduction is performed on a single element regardless of problem size. Therefore the compute time dominates the reduction time.

5 RELATED WORK

With increasing complexity of existing software, simple, text-based (*e.g.,* find and replace) program transformations alone are insufficient to solve current transformation requirements. Compiler analysis has been adopted as a requisite to understand source code and ensure correctness of complex transformations. A variety of tools have been built with the purpose to simplify the use of program transformations. Coupling compiler analysis with program transformation is highly recognized and its main goal is optimizing applications to manage resource utilization [4].

One way to simplify transformations is to raise the level of abstraction, for example, by providing a domain specific annotation or transformation language. COMPOSE-HPC [6] offers an annotation language (PAUL) and automatically transforms legacy parallel applications. COMPOSE-HPC is built on top of ROSE. It allows programmers to build custom transformations. PAUL annotations take the form of comments. It extracts these comments and attaches them to nodes in ROSE's internal code representation. A ROSE pass generates rewrite rules from the annotated AST. These rules are then passed to an actual structural rewrite engine to perform program transformation and generate the new translated code. COMPOSE-HPC works with C/Fortran with MPI or GlobalArray.

CoMPI [5] is another compiler-driven tool proposed by Bronevetsky *et. al.* with the goal of allowing incremental changes for increasing scalability of legacy MPI applications for Exascale systems. CoMPI offers a set of source code annotations to explain MPI usage in the applications. Using these annotations together with compile-time and run-time analysis techniques, CoMPI identifies MPI ranks executing on the same node. Then CoMPI fuses send and receive and removes intermediate send and receive buffers by replacing message serialization and deserialization loops with direct memory accesses. CoMPI is applied to the class of applications that explicitly serializes and deserializes data.

Because of the importance of overlapping computation with communication, there's a clear need for tools to aid HPC users in

taking advantage of MPI's nonblocking primitives. This encouraged different studies to introduce computation-communication overlap in point-to-point and/or collective operations. Worringen [32] proposed a pipelined-based optimization for SCI (Scalable Coherent Interface) [16] interconnection network. In this approach, the message buffer in the collective operation is split into N parts where each of these parts are transferred to ring buffer of SCI shared memory at the receiving process. Additionally, Danalis et al. [7] proposed a program transformation to replaces synchronous collective operations by a set of point-to-point asynchronous operations and send the data as soon as it is available. This works on applications where data dependency can be resolved statically, i.e. generates data using local arrays and have loops that can be tiled with no dependency. Their work restructures loops as tiles and places the sends between the tiles enabling the tiles to proceed in a pipeline fashion.

Most of the research has been done to find ways to improve communication-computation overlap using static analysis. However most of these tools are aimed at point-to-point primitives and the ones targeting collectives are doing changes inside collective implementation itself rather than at application level. Bamboo [20], MPI-aware [9], ASPhALT [8], and Das et al.'s work [10] are compiler-based analysis tools that aim at maximizing overlap of communication and computation. They use different analyses to move `MPI_Wait` away from `MPI_Isend` or `MPI_Ireceive`.

Bamboo [20] is based on representing MPI as a task dependency graph maintaining partial ordering over the execution of tasks. It generates a data-driven model that conceals communication overheads automatically. It supports point-to-point and collective routines. Bamboo replaces collective routines with its own implementation of point-to-point communication routines.

MPI-aware [9] achieve communication-computation overlap through generating set of data flow rules to describe behavior of point-to-point data exchange. Using these rules, MPI-aware tests if three different transformations can help in improving communication-computation overlap. It tries variable cloning to remove data dependency, loop fission to relax dependency by dividing the loop into two computation loops where one has communication calls and the other is independent of the communication. Then it test for different loop transformations such as peeling, fusion, and tiling. However all these transformations are done manually.

Similarly, Guo et al. [11] developed an approach that moves computation method execution of current iteration to be performed before communication of the previous iteration completes. Both use loop dependency analysis to identify statements that can be moved across iterations. Additionally, they insert `MPI_Test` call in the computation to ensure progress of the communication. They focus on overlapping communication computation across loop iterations.

While MPI-aware works automatically on a specific class of applications, Guo et al.'s approach identifies code locations, while the actual code changes and movement is done manually. Both of them target Fortran+MPI code.

ASPhALT [8] uses reaching definitions to identify code that generates data and move up transformation to send data as soon as it is ready. This transformation is applicable to only a certain class of applications, where a data generating loop is followed by communication calls that exchange the results of an iteration.

In a different way Das et al. [10] work is based on MPI that is already nonblocking and trying to move `MPI_Wait` further down. This work generates an intermediate representation in SSA form representation and uses use-def analysis to match `MPI_Wait` with its corresponding `MPI_Isend, MPI_Irecv` and then identifies the buffer(s) after the `_Wait` for which it is blocked.

Current research for persistent communications are focused on optimizing performance (*e.g.,* [12, 28]). To our knowledge, no tools have been developed to allow automatic detection and transformation to persistent communication.

6 CONCLUSION AND FUTURE WORK

In this paper, we have presented Petal, a compiler analysis tool for transforming MPI blocking collectives into their counterpart nonblocking collectives and forthcoming persistent variants. Petal identifies opportunities for overlapping communication with independent work to take advantage of nonblocking collectives routines (new in MPI-3) in order to improve application performance. This is achieved through the use of pointer and alias analysis to insert `MPI_Wait` before reuse of communication message. As a result, we can overlap independent communications and/or independent computation work. In addition, our tool examines each blocking collective for possible changes into persistent mode by associating reaching definitions with the communication arguments. Many scientific parallel applications are found to be compatible with persistent mode (their communication patterns are static) [25] and most of their simulation code is repeated for a certain number of time-steps, often 100 or more; hence the use of persistent collectives is expected to further improve performance of such applications.

Our results show that applications benefit from the use of nonblocking collectives in certain cases. Mainly, this occurs when messages are large and there is enough work available to be overlapped with the communication. Generally speaking, the communication-computation overlap was possible in certain regimes. These demonstrations of value will be magnified further in future for applications that have greater percentage of work in communication, and on systems with MPIs offering strong progress and blocking communication completion (the conditions best for overlapping communication and computation). Networks with greater message-passing concurrency will also benefit from code restructuring.

Building on that theme, our tool currently replaces all blocking with nonblocking irrespective of whether the particular function will provide improvement or not as a function of data size, concurrency, etc. Hence we plan either to have the user annotate which operations work with large message and/or use dynamic analysis to identify size and whether available computation work is large enough to overlap communication latency. Building analysis techniques to remove dependency inside the loop and move `_WAIT`s outside of the loop are also important features to be added. In addition, modifying alias analysis to differentiate between different parts of the array access elements is one of the main future goals.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grants Nos. CCF-1562659, CCF-1562306, OAC-1541310, and CNS-1229282. Any opinions, findings, and conclusions

or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] MPI Forum. http://www.mpi-forum.org. Accessed: May 20, 2017.

[2] Programming using MPI 1.x implementing solution of matrix system of linear equations algorithms. www.cse.iitd.ernet.in/~dheerajb/MPI/codes/day-3/c/congrad.c. Accessed: May 21st, 2017.

[3] H. Ahmed, A. Skjellum, and P. Pirkelbauer. Petal tool for analyzing and transforming legacy MPI applications. In *Revised Selected Papers of the 28th International Workshop on Languages and Compilers for Parallel Computing - Volume 9519*, LCPC 2015, pages 156–170, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[4] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, Dec. 1994.

[5] G. Bronevetsky, D. Quinlan, A. Lumsdaine, and T. Hoefler. Compiled MPI: Cost-Effective Exascale Applications Development. Technical report, Indiana University and Lawrence Livermore National Laboratory, 04 2012.

[6] D. Chavarría-Miranda, A. Panyala, W. Ma, A. Prantl, and S. Krishnamoorthy. Global transformations for legacy parallel applications via structural analysis and rewriting. *Parallel Computing*, 43:1 – 26, 2015.

[7] A. Danalis, K.-Y. Kim, L. Pollock, and M. Swany. Transformations to parallel codes for communication-computation overlap. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 58–70, Washington, DC, USA, 2005. IEEE Computer Society.

[8] A. Danalis, L. Pollock, and M. Swany. Automatic MPI application transformation with ASPhALT. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007.

[9] 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*, pages 316–325. ACM, 2009.

[10] D. Das, W. Gupta, R. Ravindran, W. Shivani, P. Sivakeshava, and R. Uppal. Compiler-controlled extraction of computation-communication overlap in MPI applications. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

[11] 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)*, 2016.

[12] M. Hatanaka, A. Hori, and Y. Ishikawa. Optimization of MPI persistent communication. In *Proceedings of the 20th European MPI Users' Group Meeting*, pages 79–84. ACM, 2013.

[13] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[14] T. Hoefler, P. Kambadur, R. L. Graham, G. Shipman, and A. Lumsdaine. A case for standard non-blocking collective operations. In *Proceedings of the 14th European Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, PVM/MPI'07, pages 125–134, Berlin, Heidelberg, 2007. Springer-Verlag.

[15] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007.

[16] IEEE. Standard for scalable coherent interface (SCI). *IEEE Std 1596-1992*, 1993.

[17] B. Morgan, D. J. Holmes, A. Skjellum, and S. Sridharam. Planning for performance: Persistent collective operations for MPI. In *EuroMPI/USA '17, September 25–28, 2017, Chicago, IL, USA*, September 2017.

[18] MPI Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

[19] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.0. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, September 21st 2012.

[20] T. Nguyen, P. Cicotti, E. Bylaska, D. Quinlan, and S. B. Baden. Bamboo: Translating MPI applications to a latency-tolerant, data-driven form. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 39:1–39:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[21] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance analysis of MPI collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[22] D. Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[23] M. Schordan, P.-H. Lin, D. Quinlan, and L.-N. Pouchet. Verification of polyhedral optimizations with constant loop bounds in finite state space computations. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 493–508. Springer, 2014.

[24] G. Shainer, T. Liu, P. Lui, and R. Graham. Accelerating high performance computing applications through MPI offloading. *HPC Advisory Council*, 2011.

[25] S. Shao, A. K. Jones, and R. Melhem. A compiler-based communication analysis approach for multiprocessor systems. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 85–85, Washington, DC, USA, 2006. IEEE Computer Society.

[26] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[27] S. Sur, M. J. Koop, and D. K. Panda. High-performance and scalable MPI over Infiniband with reduced memory usage: An in-depth performance analysis. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[28] J. L. Träff, A. Rougier, and S. Hunold. Implementing a classic: Zero-copy all-to-all communication with MPI datatypes. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 135–144. ACM, 2014.

[29] J. Tullos. Improving performance with MPI-3 non-blocking collectives. https://software.intel.com/en-us/articles/improving-performance-with-mpi-3-non-blocking-collectives. Accessed: May 21st, 2017.

[30] P. M. Widener, S. Levy, K. B. Ferreira, and T. Hoefler. On noise and the performance benefit of nonblocking collectives. *The International Journal of High Performance Computing Applications*, 30(1):121–133, Jan. 2016.

[31] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[32] J. Worringen. Pipelining and overlapping for MPI collective operations. In *Local Computer Networks*, pages 548–557, Oct 2003.