LLNL-CONF-438991

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Exploitation of Dynamic Communication Patterns through Static Analysis

R. Preissl, B. de Supinski, M. Schulz, D. Quinlan, D. Kranzlmueller, T. Panas

June 29, 2010

International Conference on Parallel Processing (ICPP)
San Diego, CA, United States
September 13, 2010 through September 16, 2010

Exploitation of Dynamic Communication Patterns through Static Analysis

Robert Preissl[1], Bronis R. de Supinski[2], Martin Schulz[2], Daniel J. Quinlan[2], Dieter Kranzlmüller[3] and Thomas Panas[2]
[1]*NERSC, Lawrence Berkeley National Laboratory, USA*
[2]*CASC, Lawrence Livermore National Laboratory, Livermore, USA*
[3]*IFI, Ludwig-Maximilians-Universität München (LMU) & Leibniz-Rechenzentrum Garching (LRZ), Germany*
rpreissl@lbl.gov, bronis@llnl.gov, schulzm@llnl.gov, dquinlan@llnl.gov, kranzlmueller@ifi.lmu.de and panas2@llnl.gov

***Abstract*—Collective operations can have a large impact on the performance of parallel applications. However, the ideal implementation of a particular collective communication often depends on both the application and the targeted machine structure. Our approach combines dynamic and static analysis techniques to identify common collective communication patterns expressed as point-to-point calls and transforms them into equivalent MPI collectives. We first detect potential collective communication patterns in runtime traces and associate them with the corresponding source code regions. If our static analysis verifies that the introduction of collectives is safe for any program flow, we then replace the original communication primitives with their collective counterpart. In this paper we introduce the necessary algorithms to determine the safety of these transformations and we demonstrate several use cases, including automatic use of new extensions to the MPI standard such as nonblocking collective operations. The use of dynamic analysis significantly reduces compile times, resulting in a speed-up of about 50 for source transformations of HPL due to more directed analysis capabilities and also dramatically decreases complexity of the underlying static analysis.**

***Keywords*-MPI Code Transformation, MPI Optimization, Collective Operations, MPI traces, Pattern Detection**

I. Introduction

The efficient use of collective communication often determines the performance of large scale parallel applications [1], [2]. For this reason, the MPI standard, the most widely used API for message passing in parallel systems, provides dedicated routines for a broad range of collective communication patterns. Each MPI implementation can use this abstraction to provide optimized versions for specific target architectures. In practice, however, such optimizations are non-trivial and depend on many factors, including the machine architecture, the application's communication pattern, and the layout of the partition used to run the job.

Several researchers have concentrated on dynamically adjusting the implementations of collective routines or transparently converting the underlying communication topologies by substituting collectives with point-to-point calls that better suit the target architecture [3], [1], [4]. However, none of these approaches automatically introduces collectives not explicitly expressed in an application and hence they miss a significant opportunity for optimizations.

We fill this gap in the optimization space by introducing novel techniques to identify collective communication patterns that are not explicitly expressed. We verify the patterns hold for all program flows and automatically transform them into explicit MPI collectives. We first use dynamic analysis of runtime traces to detect collective communication patterns and we then turn to static analysis to verify the safety of any transformation based on the dynamic information. The latter requires several steps and we present algorithms that prove the detected patterns are both input and scale independent and maintain message integrity.

Figure 1 gives an overview of our approach: we instrument the target MPI application, generate an MPI trace of the program executed under a given set of parameters, and then use pattern matching to isolate recurring collective communication structures. Next, we generate an abstract syntax tree (AST) of the application and perform static analysis to extract the control and data flow. We map the detected patterns onto this information, verify the safety of any potential transformation by showing its independence of the data and control flow, and use the results of the analysis to guide subsequent source-to-source transformations.

Our approach applies to scenarios where programmers do not realize that their applications use communication patterns that correspond to collective operations and hence do not exploit these optimized communication routines. These techniques allow programmers, who do not have detailed knowledge of MPI, to apply more efficient MPI functions without having to learn or to use them explicitly. This approach also applies to very large MPI codes where manual code optimzations are impracticable and cumbersome especially due to complicated sender and receiver matching in MPI codes. Further, it can also help in the transformation of older codes, which contain hand-coded collectives to exploit machine specific communication topologies, which no longer hold on other/newer machines. In addition, our approach can enable the automatic introduction of new MPI collective functionality, which was not available during the initial design of the application. An example for this is the inclusion of non-blocking collectives in the upcoming MPI 3 standard. While this is a promising feature for a wide range of applications, application programmers would have to invest significant effort to exploit them without our automated approach. As we demonstrate, our method can identify where to apply these new features in existing

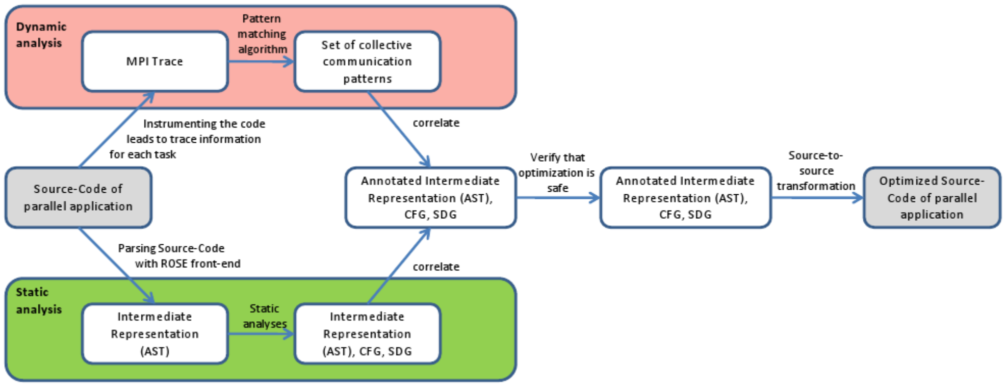
Dynamic analysis
MPI Trace
Pattern matching algorithm
Set of collective communication patterns
Instrumenting the code leads to trace information for each task
correlate
Source-Code of parallel application
Annotated Intermediate Representation (AST), CFG, SDG
Verify that optimization is safe
Annotated Intermediate Representation (AST), CFG, SDG
Source-to-source transformation
Optimized Source-Code of parallel application
Parsing Source-Code with ROSE front-end
correlate
Static analysis
Intermediate Representation (AST)
Static analyses
Intermediate Representation (AST), CFG, SDG

Figure 1. Flow diagram of the main approach

applications, such as HPL, with minimal programmer intervention.

The main contributions of our work are: (1) A formal definition of collective patterns that supports their detection for our combined dynamic and static analysis; (2) a set of static analysis algorithms that show the patterns occur independently of control or data flow and are safe to transform; (3) a set of novel transformations to introduce MPI collective operations automatically; and (4) a methodology for extending our framework to new communication or source code patterns.

II. Related Work

Several projects focus on the performance and optimization of MPI collective routines. Pješivac-Grbović et al. [1] model the performance of MPI collectives and contrast their models with experiments. STAR MPI [3] automatically finds an optimal communication topology for existing MPI collectives to match the characteristics of the application and the machine at runtime, while other projects [5], [6] have focused on optimizing collectives at compile-time. In contrast to our work, these approaches rely on the explicit use of MPI collectives and cannot exploit generic communication patterns expressed as point-to-point communication.

Other research has explored compiler-based MPI code optimizations [7], [8], or acceleration of MPI point-to-point communication [9]. Many approaches use code motion techniques to increase communication/computation overlap, an orthogonal technique that we could easily combine with our approach. Others provide several sets of optimized algorithms for each type of collective communication to replace existing MPI collective operations.

Our approach instead introduces MPI collectives into existing code that does not directly use them. We analyze the communication patterns of the application to detect repeating messaging sequences. In detail, we apply our pattern detection algorithm, which has been designed to extract arbitrary communication patterns [10], to the automated collection of collective communication patterns. Once detected, we associate these patterns with the corresponding source code and verify their generality in any application context. Thus, we combine novel static and dynamic techniques to achieve more extensive optimization, including those that require application code changes in order to exploit collective primitives efficiently.

III. Combining Static and Dynamic Analysis

As the basis for our work, we observe that certain dynamic *collective communication patterns* in a communication trace (e.g., send edges from a root task to all other tasks in the communicator) correspond to certain static *code patterns* (e.g., an if statement testing for the root task containing a loop where a message is sent to all other tasks) in the source code, especially for SPMD codes. Our approach provides an extensible set of these dynamic communication and static code patterns. When we detect such a pair, we perform three tests to determine whether the marked code regions are independent of (1) code input and (2) scale and that they (3) maintain message integrity (e.g., that a broadcast pattern always sends the same message). If the verification is successful, we replace the existing point-to-point calls with the native MPI collective operations. We provide a set of such dynamic and static patterns for several important collective MPI operations. However we show that the approach can easily be extended with additional dynamic communication and static code patterns and so is not limited to the patterns that we discuss.

While theoretically static analysis alone could achieve the results of our approach, no general static program analysis exists for accurate send-receive matching in message passing programs for arbitrary numbers of tasks [11], [12]. Further, even if it existed, it would entail significant overhead in the static analysis, likely making it infeasible for realistic programs. Thus, identifying the communication patterns requires some alternative approach.

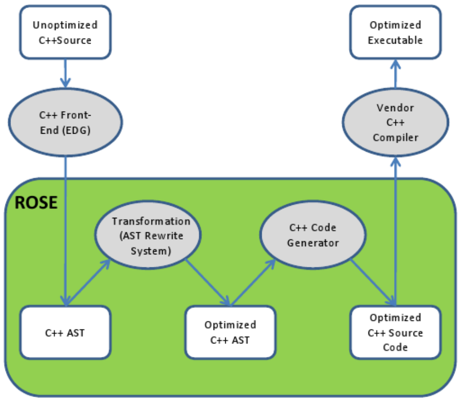

Figure 2. The ROSE compiler infrastructure

We detect the patterns from communication traces extracted during a prior execution of the application. We collect separate traces for each MPI task and condense them to a suffix tree that contains repeating sequences of communication operations. Starting from a given seed pattern, we iteratively add sequences from the different suffix trees to grow cross rank communication patterns. Constraints guide this approach to ensure patterns are compact, i.e., they do not overlap with messages outside the pattern.

Once extracted, we use the detected patterns to identify and to guide the following static analysis, which we implement in the ROSE toolkit, which generates custom source-to-source translators. ROSE provides mechanisms to translate input source code into an Intermediate Representation (IR), called the Abstract Syntax Tree (AST) [13], libraries to traverse and manipulate the information stored in the AST, and mechanisms to generate valid source code from the modified AST. The representation within the AST and the supporting data structures make exploiting knowledge of the architecture, parallel communication characteristics, and cache architecture straightforward in the specification of transformations [14], [15]. The flow diagram in Figure 2 reflects the general ROSE approach, for transforming and optimizing C++ code based on user defined abstractions and analysis steps.

IV. Classes of Collective Operations

We identify global patterns by growing them from *repeats*, task local repeating MPI communication event sequences. We define the criteria for the selection of the seed sequences, which we call *master-repeats*. The pattern detection algorithm matches repeats from other tasks (*slave-repeats*) to the master-repeat.

Figure 3 shows an example of an MPI trace in which the events in bold highlight a repeating set of MPI operations that our pattern-detection algorithm detected. This pattern is potentially equivalent to a broadcast operation using a communication structure where one process (t_4) directly sends to the others. In this example, our static analysis must verify that the pattern is a broadcast operation, i.e., it communicates the same message to all tasks, and, if so, that we can replace the associated code segments with an equivalent, but usually more efficient native MPI collective call, MPI_Bcast.

Our dynamic analysis currently detects *broadcast, scatter, gather, allgather and alltoall* as MPI collectives in a trace implemented in point-to-point operations in one of these topologies:

Star:	Each node is connected to a central node.
Ring:	Each node $r \in 0, 1, \ldots, n-1$ has a left $(r-1)$ and a right $(r+1)$ neighbors and node 0 is connected to node $n-1$.
Chain:	As Ring, but no connection between node 0 and $n-1$.
Binary Hypercube:	Each node r forms the vertex of a d-dimensional cube and is connected to d other nodes. The nodes can be addressed using a base-2 (binary) d-digit number.

A. Preliminaries and Definitions

We now provide a formal description of the local master-repeats through which we identify the corresponding global patterns from the communication trace. A defines the master-repeats and α the slave-repeats for broadcast, scatter, gather, allgather and alltoall, where n specifies the number of tasks in the communicator, i the root task and r the rank of any other task, $r \neq i \in 0, 1, \ldots, n-1$. $S[j]$ denotes an MPI_Send to task j and $R[j]$ denotes an MPI_Recv from task $j \in 0, 1, \ldots, n-1$. We denote a unit vector where all but the k^{th} bit is zero as e_k, while $d = \lceil log_2(n) \rceil$ is the dimension of a binary hypercube and $\mathbb{N}_{0,1}$=$\{0, 1\}$. The function $u(x) : \mathbb{N} \rightarrow \mathbb{N}_{0,1}^d$ returns the bit-vector representation of an integer x ($e_k = u(2^k)$), whereas $v(y) : \mathbb{N}_{0,1}^d \rightarrow \mathbb{N}$ is the inverse of u. $\bigoplus : (\mathbb{N}_{0,1}^d \times \mathbb{N}_{0,1}^d) \rightarrow \mathbb{N}_{0,1}^d$ is a bitwise logical XOR operator and $\bigotimes : (\mathbb{N}_{0,1}^d \times \mathbb{N}_{0,1}^d) \rightarrow \mathbb{N}_{0,1}^d$ is a bitwise logical AND operator, while $\%$ defines a modulo operator on integer values. Finally, $mask = u(2^d)$ and c defines a constant, $c \in 0, 1, \ldots, n-1$.

B. Defining Master- and Slave-Repeats

We define the characteristics of substrings for master- and slave-repeats for each of the collective properties defined above.

Tables I and II show formal definitions of master- and slave-repeats for broadcasts and scatters for the previously defined topologies. For example, we define the master-repeat for a broadcast or scatter for a star topology as the concatenation of send events, starting with a send event to process $(0 + c)\%n$ (Table I). Correspondingly, the slave-repeat is a single receive event from the root

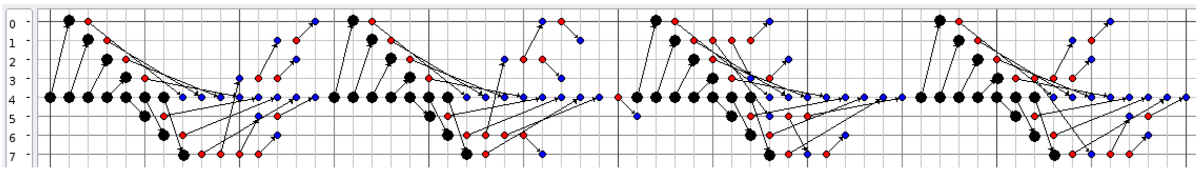

Figure 3. A detected pattern representing a broadcast operation in an MPI trace

Table I
MASTER-REPEAT FOR BROADCAST & SCATTER

Topology	Master-Repeat
Star	$A = "S[f(0)], \ldots, S[f(i-1)], S[f(i+1)], \ldots, S[f(n-1)]"$, where $f(x) = (x+c)\%n$
Ring	$A = "S[(i+1)\%n], R[(i-1)\%n]"$
Chain	$A = "S[(i+1)\%n]"$
Binary Hypercube	$A = "S[g(0)], S[g(1)], \ldots, S[g(d-1)]"$, where $g(x) = v(u(x) \bigoplus e_x)$ and $d = \lceil log_2(n) \rceil$

Table II
SLAVE-REPEAT FOR BROADCAST & SCATTER

Topology	Slave-Repeat
Star	$\alpha = "R[i]"$
Ring	$\alpha = "R[(r-1)\%n], S[(r+1)\%n]"$
Chain	$\alpha = "R[(r-1)\%n], S[(r+1)\%n]"$
Binary Hypercube	$\alpha = "h(0), h(1), \ldots, h(d-1)"$, where $h(x) = \begin{cases} R[g(x)] & \text{if } v(r \bigotimes mask) = 0 \wedge v(r \bigotimes e_x) = 0 \\ S[g(x)] & \text{if } v(r \bigotimes mask) = 0 \wedge v(r \bigotimes e_x) \neq 0 \\ \emptyset & \text{else} \end{cases}$

process (Table II). This matches the example of the broadcast pattern in Figure 3 with ($n = 8, i = 4, c = 0$). $A = "S[0], S[1], S[2], S[3], S[5], S[6], S[7]"$ is the master-repeat and the corresponding slave-repeat α is $"R[4]"$. Similarly, for a broadcast on a ring, the master task invokes MPI_Send followed by MPI_Recv and the slaves use the same operations, but in reverse order (Table II). In general, the definitions in the tables reflect the basic rules for several variations of how point-to-point operations can implement collective operations.

We note that the static analysis is different for broadcast and scatter operations. Scatters send different messages to the individual nodes, while broadcasts communicate the same data.

A gather is the reverse operation to a scatter and results in similar master- and slave-repeats, only swapping send and receive events in Tables I and II. The dynamic features of our allgathers and alltoalls are concatenations of substrings for gathers and broadcasts and can be specified analogously for each topology.

V. VERIFYING TRANSFORMATION SAFETY

Once our pattern detection algorithm finds potential communication bottlenecks in the form of collective communi-

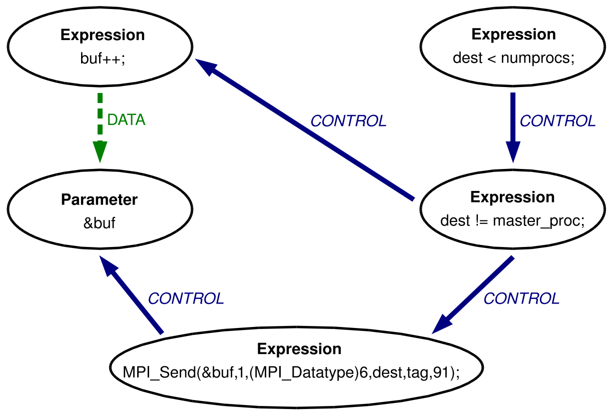

Figure 4. Simplified SDG around MPI_Send

cation implemented by point-to-point operations, we transfer the results to our static analysis component. First, we generate an AST of the application and mark the nodes representing the corresponding MPI_Send and MPI_Recv calls. We then compute the System-Dependence-Graph (SDG) [16] and the Control-Flow-Graph (CFG), which combine to describe all dependencies between nodes in the AST. Figure 4 illustrates a simplified excerpt of an SDG that ROSE generated. It shows data and control dependencies around an MPI_Send call. It represents the data-flow between the statements and expressions and shows the control-dependence edges that represent control conditions for individual statements or expressions.

Based on the information contained in the SDG and CFG, we then perform a series of static analysis steps to verify that any collective pattern considered for transformation is valid independent of any dynamic information like command-line arguments or the number of tasks involved. In particular, we have to verify three main static analysis criteria: (1) Input Independence: the detected pattern is independent of any input to the program; (2) Scale Independence: The detected pattern spans all tasks of the communicator in the program when run at any scale; and (3) Message Integrity: Messages are not changed inside the collective operation pattern. Although these three criteria apply equally to all communication patterns, the way some of these properties are verified statically in the source code depends on the

Algorithm 1 Input independence

```
Require: AST-annotated send and receive functions in dynamic pattern
Ensure: Determine input independence
1: Compute code pattern (backward traversals of SDG starting at send or receive nodes), specified by collective statement
2: Apply ROSE-query operator on code pattern to detect conditional statements and store results in list $CS$
3: if sizeof($CS$) $\geq 1$ then
4:     abort transformation
5: end if
```

topology: for example a broadcast implemented with a ring topology requires different static-analysis than a broadcast based on a star.

A. Input Independence

In order for *input independence* to hold, no conditional statements are allowed within the code pattern. To define such code patterns we look for *collective statements* that are responsible for the collective behavior and determine the boundaries of a code pattern. Collective statements are single *for*, *while* or *do-while* loops for star topologies. For rings and chains they are modulo statements and for hypercubes they are nested loops. We locate collective statements with backward traversals of the control dependence graph from the SDG starting at send or receive nodes in our dynamic pattern.

Listing 1 shows a code excerpt for the master process of a broadcast operation on a star topology. As long as the number of tasks does not exceed 41, a code transformation into MPI_Bcast would be correct; but leading to incorrect behavior otherwise. We detect such conditional statements like the one in line 5 through static analysis by applying query operators on AST subtrees. We show an example query below, where the *querySubTree* function extracts all *SgIfStmt* nodes (repesenting If-Statements in the ROSE IR) in a subtree of the AST (*subtree* indicates the root node for the code pattern subtree in the AST, e.g., a single for loop specifying the collective statement) and stores the results in a *RoseSTLContainer*, which is basically an STL list of *SgNode*s (*SgNode* represents the base class for all IR nodes).

```
RoseSTLContainer<SgNode*> list_of_ForLoops =
NodeQuery::querySubTree(subtree ,VSgIfStmt);
```

The only exceptions are if-statements in combination with MPI-Test-Functions like MPI_Test and MPI_Probe. Such Test-Functions are often used to overlap communication and computation and cause additional complexity for our static analysis. We present additional details for them in Section VII-B. Algorithm 1 summarizes the required steps for proving input independence.

```
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
2 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  if(rank == master_task) {
4   for (i=0; i<numtasks; ++i) {
      if(rank == 42) continue;
6     MPI_Send(..,i,..); }}
```

Listing 1. Checking input independence

Algorithm 2 Scale independence

```
Require: AST-annotated send and receive functions
Ensure: Determine scale independence
1: Reuse collective statements $CS$ computed in Algorithm 1 and determine virtual topology $V$
2: if ($V \in \{star, hypercube\}$) && $\neg$($for$, $while$ or $do-while$ loop of $CS$ data dependent on MPI_Comm_Size) then
3:     abort transformation
4: else
5:     if ($V \in \{chain, ring\}$) && $\neg$($modulo$ operator's right hand side operand data dependent on MPI_Comm_Size) then
6:         abort transformation
7:     end if
8: end if
```

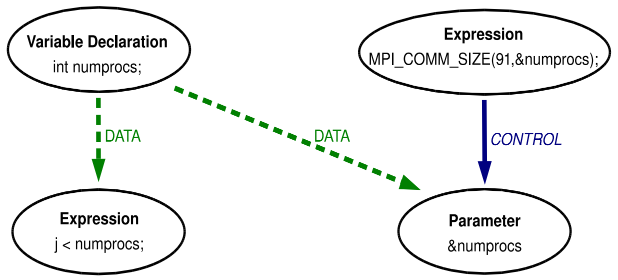

Figure 5. SDG around MPI_Comm_Size

B. Scale Independence

To test for *scale independence* we look for data dependencies of the collective statements on functions that compute the number of tasks in a communicator (i.e., MPI_Comm_Size). For star and hypercube topologies, the *for*, *while* or *do-while* loop's test statement and, for ring and chain topologies, the modulo's right hand side operand (e.g., line 8 of Listing 2, which shows a broadcast for a ring topology) must have a data dependence on MPI_Comm_Size to guarantee scale independence. Full generality of this test would require *pointer analysis* since the variable that stores the number of processes in the communicator is passed by reference to MPI_Comm_Size. Instead, we limit the scope in which the dependence occurs, conservatively rejecting transformations that more accurate analysis could determine.

```
int x, numtasks, master_task = 2;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
for (i=0; i<some_iterations; ++i) {
  if(rank == master_task) {
    MPI_Send(&x,1,MPI_INT,(rank+1)%numtasks,
              tag,MPI_COMM_WORLD);
    MPI_Recv(&x,1,MPI_INT,(rank-1)%numtasks,
              tag,MPI_COMM_WORLD,&Stat); }
  else{
    MPI_Recv(&x,1,MPI_INT,(rank-1)%numtasks,
              tag,MPI_COMM_WORLD,&Stat);
    MPI_Send(&x,1,MPI_INT,(rank+1)%numtasks,
              tag,MPI_COMM_WORLD); }             }
```

Listing 2. Broadcast on ring

```
int x, numtasks, master_task = 4;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
for (i=0; i<some_iterations; ++i) {
  if(rank == master_task){
    for (j=0; j<numtasks; ++j) {
      if( rank != master_task ) {
        MPI_Send(&x,1,MPI_INT,j,
                  tag,MPI_COMM_WORLD); }}}
  else {MPI_Recv(&x,1,MPI_INT,master_task,
                  tag,MPI_COMM_WORLD,&Stat);} }
```

Listing 3. Transformation of broadcast:before

```
  if (rank == master_task) {
      for(dest = 0; dest < numtasks; ++dest) {
          if(dest != master_task) {
              MPI_Send(&buf,1,MPI_INT,dest,
5                     tag,MPI_COMM_WORLD);
              buf++; }}}
  else{
      MPI_Recv(&buf,1,MPI_INT,master_task,
              tag,MPI_COMM_WORLD,&Stat); }
```

Listing 4. Message integrity fails: broadcast

Algorithm 3 Message integrity

Require: AST-annotated send and receive functions
Ensure: Determine message integrity

```
1: Reuse collective statements $CS$ (Algorithm 1)
2: Get message of collective function and compute message's def-use chain $D$
3: if any $d \in D$ is control dependent on $CS$ then
4:     abort transformation
5: end if
```

Figure 5 shows the simplified picture of the SDG for the code excerpt in Listing 3. It shows data- and control-dependence on the MPI_Comm_Size function. The modulo's right hand side operand $numprocs$ is passed by reference to MPI_Comm_size and, since the static analysis does not support pointer analysis, no data dependence edge from $numprocs$ to the MPI_Comm_size function can be seen. However, as stated above, this problem can be circumvented by first getting the variable declaration from $numprocs$ in the SDG, which is "*int numprocs*" in the left upper corner in Figure 5. Since the SDG gives interprocedural control and data dependence, it does not matter if the variable declaration and the broadcasting loop are in different functions of the code. Then the SDG (Figure 5) depicts that $numprocs$ is passed by reference to MPI_Comm_size as highlighted in Figure 5 by the address operator "&" applied to $numprocs$ as input parameter to MPI_Comm_size. Additionally we prove that $numprocs$ is not modified anymore between the place where its value is being set in MPI_Comm_size and the time it is used in the loop. Algorithm 2 shows the two basic steps for checking scale independence.

C. Message Integrity

Message Integrity is fulfilled if there are no modifications of the message buffer inside the collective communication statement that do not match the semantics of the collective operation[1]. First, the algorithm computes the collective communication statement in the AST and identifies the corresponding nodes in the SDG. Then the system computes the message of the collective function in the SDG and figures out all locations that illegally modify this message in the code. This is expressed by its data dependence chain in parent direction. If any of those statements of the message's data dependence chain are control dependent on the collective communication statement (i.e., improper modifications can happen before and after — but not during — the code pattern is executed), message integrity is not given and therefore code transformations have to be denied for certain collective functions. For example, messages must not be changed inside a for loop in case of a broadcast like the following example (Listing 4) demonstrates.

Figure 4 shows the simplified image of the SDG for the code excerpt in Listing 4. In detail, it is representing control and data dependency from the root process around its MPI_Send function. Despite the fact that the code in Listing 4 would generate a detectable broadcast pattern for a star-like communication topology, the static analysis will reject this pattern because of the message increment ("$buf++$") statement in line 6.

First, we identify the send statement from line 4 (Listing 4) in the SDG. It is the MPI_Send expression at the bottom of Figure 4. Starting from this node we follow the control dependence chain in parent direction (backward slicing) until the collective communication statement is

[1]This depends on the particular collective communication function. For instance, during a broadcast operation any changes to the send buffer are forbidden, whereas a scatter/gather operation can be implemented by sending a constantly increasing single value.

reached (here a for loop). The algorithm identifies this loop, because the fourth parameter (the destination parameter $dest$) of the send MPI call is data dependent on this for loop. In case of the code in Listing 4, the collective communication statement's test-function is represented in the SDG in Figure 4 in the upper right corner as the expression "$dest < numprocs$". Now, we locate the first parameter (the message to be sent) of the send function in the SDG. Note, since the message is an integer value (following data dependence chain in parent direction until variable declaration, "$int\ buf$" is reached) the dynamic pattern cannot be a scatter operation and therefore can only lead to a broadcast. It is "$\&buf$" in the SDG in Figure 4. Starting from this node, we traverse its data dependence chain in parent direction (backward slice) and check if its value is modified **inside** the collective communication statement. This is the case in line 6 of the code in Listing 4, represented by the expression "$buf++$" in the corresponding SDG. (Note, that there is a data dependence edge from the first parameter "$\&buf$" to "$buf++$"). Since the "$buf++$" node in the SDG is control dependent on the collective communication statement, we can not perform any source transformations. This control dependence can be seen in Figure 4, indicated by control dependence edges from "$buf++$" to "$dest \neq master\_proc$", which is control dependent on "$dest < numprocs$".

Other communication patterns require a similar integrity analysis, although with slight variations. The key property of a scatter is that a certain container (e.g., an array) is scattered across the nodes in the communicator. In detail, the first parameter of the MPI_Send, the message buffer, must have the form: "$send\_buf$ **AddSubOp** ($loop\_var$ **MultDivOp** $size$)", where $AddSubOp = \{+, -\}$, $MultDivOp = \{*, /\}$, $send\_buf$ is the pointer to the scattered array and $loop\_var$ is the incremented or decremented loop variable. The second MPI_Send parameter must also be $size$. For instance:

```
MPI_Send(sendbuf + i*recv_size, recv_size,
         MPI_FLOAT, i, tag, MPI_COMM_WORLD);
```

The supported dynamic pattern for allgather is composed of a gather followed by a broadcast. As a result the transformation requirements are the same as for gather combined with those for broadcast, with the additional *message integrity* rule that the message must not be changed between the gather and the broadcast.

D. Extensibility

Our approach currently covers the common patterns described previously. However, other variations are possible and we easily accommodate them by supporting user defined patterns. These extensions under our semi-automated approach require the addition of new characteristics for master- and slave-repeats to the dynamic analysis. The static code pattern also must be specified so we can apply the three previously defined safety tests.

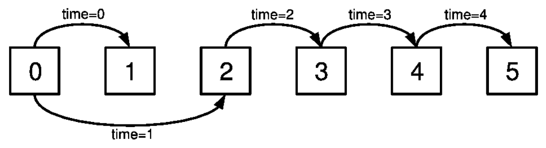

Figure 6. Increasing ring broadcast from HPL

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
int root=0, next=(rank+1)%numtasks;
int prev=(rank-1)%numtasks;
if((prev-1)%numtasks == root){partner=root;}
else { partner = prev; }
if( rank == root ) { MPI_Send(next,..);
                     MPI_Send(next + 1,..);  }
else {  MPI_Recv(partner,..);
        if(prev != root && next != root) {
          MPI_Send(next,..);                } }
```

Listing 5. Modified increasing ring in HPL

We demonstrate on a concrete example how to extend the functionality by taking a currently unsupported collective communication – a broadcast for an *increasing ring*, which is one of six broadcast algorithms used in High Performance Linpack [17]. In an increasing ring, task 0 sends two messages and process 1 only receives one message. So $0 \rightarrow 1$; $0 \rightarrow 2$; $2 \rightarrow 3$; $3 \rightarrow 4$ and so on. Figure 6 shows this communication pattern for 6 tasks. New dynamic characteristics for this communication, which have to be added to the code of the pattern detection algorithm, are:

$A =$ "$S[(i+1)\%n], S[(i+2)\%n]$" and $\alpha =$ "$R[(r-1)\%n]$" if $r = (i+1)\%n$, $\alpha =$ "$R[(r-2)\%n], S[(r+1)\%n]$" if $r = (i+2)\%n$, or otherwise $\alpha =$ "$R[(r-1)\%n], S[(r+1)\%n]$".

For example, the master repeat (A) for a modified increasing ring with 6 processes is "$S[0], S[1]$" and the slave repeat (α) for the process with rank 2 ($= r$) (third line of formula) is $\alpha =$"$R[0], S[3]$", according to the formula above.

Starting from the send and receive nodes in the AST, which we determine by dynamic analysis, we look for the code pattern for this collective class. A valid code pattern is a root task sending out two messages to its next two neighbors and the slaves receiving and sending, except two slaves do not send. Listing 5 shows pseudocode for this code pattern. If we detect the dynamic communication and static code patterns, we apply our safety tests and, if the transformation is safe, use a native MPI collective.

VI. Source Transformation

After detecting and extracting patterns from the dynamic trace we verify their safety. If this is successful, we apply a series of transformations for the static code pattern

Algorithm 4 Transformation of MPI source code

```
Require: MPI Source code, communication patterns
Ensure: Optimized parallel code
 1: Get collective pattern $M$ from dynamic analysis
 2: if $M \neq \emptyset$ then
 3:     Generate AST (ROSE front-end)
 4:     Generate CFG & SDG (ROSE mid-end)
 5:     Relate pattern information to source code
 6:     Locate code pattern through static analysis
 7:     Verify transformation safety through static anal.
 8:     if code pattern found && transformation is safe then
 9:         Transform source code (ROSE mid-end)
10:         Generate optimized code (ROSE back-end)
11:     end if
12: end if
```

then replacing it with the native MPI collectives such as *MPI_Bcast*. Finally, we generate valid C++ code from the modified AST through the ROSE rewrite mechanism.

Algorithm 4 outlines the source code transformation process, combining dynamic and static analysis for transforming point-to-point based collectives into native MPI collective operations. In the code transformation process the subtrees for the send and receive operations are cut off from the pattern in the AST and parts of their parameters for the newly generated original collective MPI functions are reused. In detail, the system "recycles" parameters of the receive event (e.g., the source of a broadcast operation) and generates a new function call expression in the AST for the original MPI function call. We must compute the root of the collective operation in some cases, such as when we find a broadcast in a chain- or ring-topology, which does not explicitly specify the master-task in the MPI_Recv call. Listing 2 shows such a ring broadcast code pattern while Listing 3 shows a simpler scenario in which the receive function already holds the $master\_task$ parameter, in which case we simply reuse it.

In case of not explicitly declared master-tasks, we exploit the SPMD nature of the code pattern to find the missing parameter guarding the code that the master task executes, which is for example contained in the *if* statement on line 5 of Listing 2. We use the CFG to identify this *if* statement. Since the send event (line 6) happens before the receive event (line 8) in the control flow, the root executes this part. We identify $rank$ as the variable that stores its rank since it is passed by reference to MPI_Comm_rank. Finally we use the variable that is compared to $rank$ as the broadcast root.

VII. Case Studies

To show its capabilities, we first apply our approach to a simple code that implements a broadcast in point-to-point operations. We then show how our approach can automatically update MPI applications to use new functionality such as nonblocking collectives.

```
int x, numtasks, master_task = 4;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
for (i=0; i<some_iterations; ++i) {
  if(rank == master_task) {
    MPI_Bcast(&x,1,(MPI_Datatype)6,
              master_task,(91)); }
  else { MPI_Bcast(&x,1,(MPI_Datatype)6,
              master_task,(91)); }       }
```

Listing 6. Transformation of broadcast:after

A. Transforming Collectives (Broadcast)

Listing 3 shows a point-to-point broadcast on a star topology. Its loop on line 6 distributes data from task t_4 to all other tasks. An execution of this application produces the trace shown in Figure 3. The detected master- and slave-repeats match the formal description of master- and slave-repeats for broadcasts (Tables I and II in Section IV-B) on a star topology. Thus, our pattern detection algorithm automatically identifies this broadcast communication pattern and marks it as a potentially inefficient collective communication.

Our safety tests succeed since the message x is not changed during the broadcast (message integrity) and the collective statements' bound variable $numtasks$ is data dependent on MPI_Comm_size (scale independence). Input independence is guaranteed since it has no conditional statements on input variables. Thus, our static analysis determines the code implements a broadcast in any execution context so we can transform it into an MPI_Bcast function. Listing 6 shows the transformed source code that allows the application to benefit from the highly tuned MPI collective routines.

B. Transforming HPL

The Linpack $N \times N$ benchmark (HPL) [17] computes the solution of a linear system of equations $Ax = b$, where A is a dense $N \times N$ matrix, and x and b are vectors of size N. HPL factorizes Matrix A in place as the product $A = LU$, where L and U are lower and upper triangular matrices. It then logically partitions both dimensions of the matrix into $NB \times NB$ blocks, which it cyclically assigns to a $P \times Q$ process grid.

Figure 7 shows an excerpt of an HPL MPI trace for 16 tasks in a $(P = 1) \times (Q = 16)$ process grid where an *increasing-ring* broadcasts the factorized panel of columns. The highlighted events form an instance of a detected broadcast pattern on a chain topology, where the root task (t_0) sends to its right neighbor (t_1) and t_j receives a message from task t_{j-1} and sends to task t_{j+1}. We detect other instances of this chain-broadcast, e.g., where task t_1 is

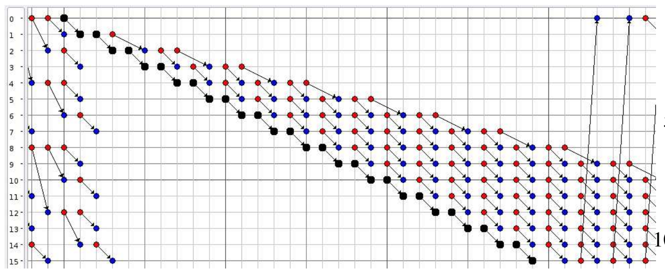

Figure 7. HPL chain broadcast pattern

```
void HPL_bcast() {
  int test = HPL_KEEP_TESTING;
  while( test == HPL_KEEP_TESTING ) {
    // Do some computation: e.g. HPL_dgemm
    (void)HPL_bcast_test( .. ,&test);   }}           5
int HPL_bcast_test( .. , int* IFLAG) {
  if( rank == root ) {
    MPI_Send(_M_BUFF,_M_COUNT,_M_TYPE,
             next_task,msgid,comm); }
  else{                                              10
    MPI_Iprobe(prev,msgid,comm,
               &go,&PANEL->status[0]);
    if( go != 0 ){
      MPI_Recv(_M_BUFF,_M_COUNT,_M_TYPE,
               recv_task,msgid,comm,                 15
               &PANEL->status[0]);
      if( next_task != root ) {
        MPI_Send(_M_BUFF,_M_COUNT,_M_TYPE,
                 next_task,msgid,comm); }}
    else{ *IFLAG=HPL_KEEP_TESTING;                   20
      return(*IFLAG);                    }}
  *IFLAG = HPL_SUCCESS; return(*IFLAG);}
```

Listing 7. Original HPL chain broadcast

the root, despite that they arise from the same send and receive statements. As with the previous example, our static analysis must verify the three criteria from Section V and, if successful, we can replace the associated AST nodes with an equivalent MPI_Bcast.

We must resolve the additional *if* statement (line 17 in Listing 7) in the slave's portion of the code pattern before we can perform the safety tests. The MPI_Iprobe on line 11, which tests for the message sent by lines 8 or 18, causes this additional complexity. This *if* statement does not alter the broadcast semantics since the outer *while* loop on line 3 ensures that the MPI expressions inside it execute once per task. If the message has arrived, the true branch (i.e., the receive and send) executes and the loop terminates. Otherwise the application performs computation that is independent of the message and then executes the MPI_Iprobe. Our data dependence analysis detects these characteristics through interprocedural analysis that the SDG supports. We easily prove *message integrity* since the message $\_M\_BUFF$ is not altered between the slave's receive and send. *Input independence* clearly holds since no inappropriate conditional statements occur within the code pattern (the *if* statement on line 13 refers to the MPI_Iprobe and the one on line 17 is part of a chain broadcast code pattern). The code is *scale independent* since a data dependence exists from $next\_task, recv\_task$ to MPI_Comm_size.

However, unlike the previous simple example, this transformation decreases the performance of the application because HPL significantly overlaps computation with communication. Thus, a blocking MPI_Bcast decreases this overlap substantially, which results in the performance loss. The next subsection shows how we can avoid this problem.

C. Automatically Updating to MPI 3.0

The communication patterns in HPL implement nonblocking collective operations, which are not part of the current MPI standard. However, such constructs are proposed for the upcoming MPI 3.0 standard and *libNBC* portably implements a first functional prototype (although not yet fully optimized) of nonblocking collective operations on top of MPI-1 [18], [19]. While nonblocking collective operations could mitigate pseudo-synchronization effects and hide latency costs, properly applying them to existing real-world applications is non-trivial. Their use often requires significant restructuring to exploit communication/computation overlap fully [20]. This requirement confronts the programmer with yet more complexity in the optimization process. Our approach, however, can automatically include nonblocking collectives and combine with code motion techniques to provide the necessary overlap. We demonstrate this potential by transforming HPL with a nonblocking broadcast (NBC_Ibcast) that preserves the carefully constructed communication/computation overlap.

The transformed code preserves the overlapping nature of the HPL_bcast_test function. We added global variables and test functions (NBC_Test similarly to MPI_Iprobe) that test if the nonblocking broadcast has finished. The transformation process takes 4.5 seconds on a standard PC with a 2.8GHz CPU and 2GB RAM running Linux.

Our approach of combining static and dynamic analysis vastly simplifies the analysis needed to detect such transformation candidates. We only need the SDG. Creating the SDG dominates static analysis overhead (about 90%). Since we only require the SDG for the source files that contain MPI calls in the dynamic pattern, the analysis is considerably sped up. The complexity of SDG construction grows exponentially with the size of the source code. Creating the SDG for all HPL files that contain MPI communication (as would be necessary if we did not have dynamic analysis to limit its scope) takes 195 seconds. Generation of our "semantically sliced" SDG takes only 3.9 seconds, an improvement of a factor of 50.

Early runtime experiments show only marginal performance benefits with *libNBC*. The gains are limited because HPL sends large messages and the libNBC implementation does not pipeline packets from an individual message. However, we can expect this and other optimizations of the implementation of nonblocking collectives once they are included in the standard. At that point, we expect our transformation will provide significant benefits even to a highly optimized benchmark like HPL.

VIII. Conclusions

This paper presents novel dynamic and static program analyses that support algorithms to transform source code of parallel scientific applications automatically. In particular, we focus on optimizing MPI point-to-point operations that correspond to collective operations, which is often critical to overall application performance. We detect collective communication patterns in runtime traces, apply three tests to verify that these collective patterns exist independent of application context, and then provide transformations that replace them with equivalent MPI collective routines. Our work closes an important gap in existing frameworks for automated MPI optimization, which has previously mostly focused on optimizing existing MPI collective routines or providing communication/computation overlap through code motion.

We demonstrated our approach on the HPL benchmark as well as simple examples. We also demonstrated with HPL how our approach can transparently update a legacy code to use new MPI features like the recently agreed upon nonblocking collectives, which will be part of the MPI 3.0 standard. We also demonstrated that combining dynamic and static analysis provides a significant performance advantage to the analysis, speeding up the static analysis time by a factor of 50.

References

[1] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of MPI collective operations," *Cluster Computing*, vol. 10, no. 2, pp. 127–143, 2007.

[2] R. Rabenseifner, "Automatic MPI Counter Profiling of All Users: First Results on a CRAY T3E 900-512," in *Proceedings of the Message Passing Interface Developer's and User's Conference (MPIDC'99), Atlanta*, March 1999, pp. 77–85.

[3] A. Faraj, X. Yuan, and D. Lowenthal, "STAR-MPI: self tuned adaptive routines for MPI collective operations," in *ICS '06: Proceedings of the 20th Annual International Conference on Supercomputing*. Cairns, Queensland, Australia: ACM, 2006, pp. 199–208.

[4] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra, "Automatically tuned collective communications," in *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*. Dallas, TX, USA: IEEE Computer Society, 2000, p. 3.

[5] A. Faraj and X. Yuan, "Automatic generation and tuning of MPI collective communication routines," in *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*. Montreal, Canada: ACM, 2005, pp. 393–402.

[6] A. Karwande, X. Yuan, and D. K. Lowenthal, "CC–MPI: A compiled communication capable MPI prototype for ethernet switched clusters," *SIGPLAN Not.*, vol. 38, no. 10, pp. 95–106, 2003.

[7] A. Danalis, L. Pollock, M. Swany, and J. Cavazos, "MPI-aware compiler optimizations for improving communication-computation overlap," in *ICS '09: Proceedings of the 23rd international conference on Supercomputing*. Yorktown Heights, NY, USA: ACM, 2009, pp. 316–325.

[8] C. Hu, Y. Shao, J. Wang, and J. Li, "Automatic Transformation for Overlapping Communication and Computation," in *NPC '08: Proceedings of the IFIP International Conference on Network and Parallel Computing*. Shanghai, China: Springer-Verlag, 2008, pp. 210–220.

[9] M. Lauria and A. Chien, "MPI-FM: high performance MPI on workstation clusters," *J. Parallel Distrib. Comput.*, vol. 40, no. 1, pp. 4–18, 1997.

[10] R. Preissl, T. Köckerbauer, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan, "Detecting Patterns in MPI Communication Traces," in *ICPP*, 2008, pp. 230–237.

[11] G. Bronevetsky, "Communication-Sensitive Static Dataflow for Parallel Message Passing Applications," *IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 1–12, 2009.

[12] S. Shao, A. Jones, and R. Melhem, "A compiler-based communication analysis approach for multiprocessor systems," *IEEE International Parallel & Distributed Processing Symposium*, p. 65, 2006.

[13] M. Schordan and D. J. Quinlan, "A source-to-source architecture for user-defined optimizations," in *JMLC '03: Joint Modular Languages Conference*, ser. LNCS 2789. Springer-Verlag, 2003, pp. 214–223.

[14] T. Panas, D. J. Quinlan, and R. Vuduc, "Tool Support for Inspecting the Code Quality of HPC Applications," in *SE-HPC '07: Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing Applications*. Minneapolis, MS, USA: IEEE Computer Society, 2007, p. 2.

[15] D. J. Quinlan, "ROSE: Compiler Support for Object-Oriented Frameworks," *Parallel Processing Letters*, vol. 10, no. 2/3, pp. 215–226, 2000.

[16] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *SIGPLAN Not.*, vol. 39, no. 4, pp. 229–243, 2004.

[17] "HPL: A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, http://www.netlib.org/benchmark/hpl/," Sept. 2008.

[18] T. Hoefler and A. Lumsdaine, "Design, Implementation, and Usage of LibNBC," http://www.unixer.de/publications/, Aug. 2006, Open Systems Lab, Indiana University, IN, USA.

[19] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for MPI," in *SC*, 2007, p. 52.

[20] T. Hoefler, P. Gottschling, and A. Lumsdaine, "Leveraging non-blocking collective communication in high-performance applications," in *SPAA*, 2008, pp. 113–115.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.