ParFuse: Parallel and Compositional Analysis of Message Passing Programs

Sriram Aananthakrishnan[1], Greg Bronevetsky[2], Mark Baranowski[1], and Ganesh Gopalakrishnan[1]

[1] University of Utah
{sriram,baranows,ganesh}@cs.utah.edu
[2] Google Inc.
bronevet@google.com

Abstract. Static analysis discovers provable true properties about behaviors of programs that are useful in optimization, debugging and verification. Sequential static analysis techniques fail to interpret the message passing semantics of the MPI and lack the ability to optimize or check the message passing behaviors of MPI programs. In this paper, we introduce an abstraction for approximating the message passing behaviors of MPI programs that is more precise than prior work and is applicable to a wide variety of applications. Our approach builds on the compositional paradigm where we transparently extend MPI support to sequential analyses through composition with our MPI analyses. This is the first framework where the data flow analysis is carried out in parallel on a cluster, with the message-carried data flow facts for refining inter-process data flow analysis states. We detail ParFuse – a framework that supports such parallel and compositional analysis of MPI programs, report its scalability and detail the prospects of extending our work for more powerful analyses.

1 Introduction

HPC systems have become increasingly complex as we step into the exascale computing era. In parallel, MPI has also evolved, introducing sophisticated communication primitives for interprocess communication. Debugging and performance tuning of message passing programs have become notoriously difficult. With the growing complexity of writing message passing programs, tools to assist developers are crucially needed. While many dynamic and runtime tools exist to assist MPI programmers, only a handful of static analysis based tools exist in comparison. Static analysis of MPI programs can discover provably true properties about the communication behaviors of the MPI programs which are useful in optimization, error detection and verification. For instance, compilers can replace point-to-point operations in a neighborhood communication pattern with their optimized collective counterparts [11] if the MPI program's communication topology can be determined.

Many standard dataflow analyses such as constant propagation are MPI agnostic i.e., they do not model the effects of dataflow due to MPI communication,

losing precision at the call sites of MPI operations and thereby missing the opportunity to apply program optimizations. Static analysis of MPI programs require abstractions for modeling the communication behaviors where the abstraction must provide an interpretation for MPI operations and compute the possible message matches. This task is challenging requiring composition of multiple static analyses.

Prior work on analyzing MPI programs have focused on a non-compositional approach. The message passing semantics are modeled by constructing a communication graph [19, 20, 2] and the analysis associates special transfer functions for each MPI operation to interpret the dataflow information along the communication edges. Adopting a new dataflow analysis for MPI programs under this setting requires implementing the special transfer functions corresponding to each MPI operation. In this paper, we build on the compositional principles of the Fuse[3] framework where we implement a suite of analyses for modeling the MPI message passing semantics. Our approach allows any dataflow analyses to be composed with MPI analyses which transparently adds MPI support for the MPI-agnostic analyses.

In this paper, we offer the first static analysis method with the following features:

- We introduce specific abstractions for MPI operations which enables us to reach a useful level of accuracy that covers many real applications.
- Our abstractions for MPI operations are built on top of the Fuse framework where MPI-agnostic static analyses are leveraged with MPI support through composition with our MPI analyses.
- Our analysis is carried out in parallel on a cluster to ameliorate the cost when analyzing an MPI program with N processes. We provide an evaluation of the scalability of our approach.
- Visualization of possible communication matches as an automatically generated "dot graph" built using our compositional infrastructure for analyzing MPI programs.

The rest of the paper provides background on compositional analysis and prior work in Section 2, our abstractions for MPI semantics in Section 3, MPI analyses that realizes our abstraction in Section 4, our parallel and compositional ParFuse framework in Section 5 and the results in Section 6. Related work and concluding remarks follow.

2 Background

2.1 Compositional Analysis

In a prior project [3], we introduced the Fuse compositional framework that simplifies composition of static analyses through a data structure called Abstract Transition System (ATS). ATSs are graphs where the nodes correspond to different possible code execution paths and edges represent transitions from one program state to another. Static analyses can be executed on ATSs and compute constraints (e.g. dataflow facts) on reachable program executions, which are stored as annotations on each ATS node. The ATS organizes the constraints

on reachable executions using sets of program state components (memory locations, values or operations). This allows analyses to portably communicate the constraints on reachable executions as set constraints on state components to other analyses, which we denote as "abstract objects". While the abstract objects are opaque (their individual values may be infinitely many), its implementations must include standard set operations such as overlaps, must-equals, equal-sets, subset etc. This enables other analyses to compare two abstract objects and make complex inferences based on them without knowing how they were computed. The graph representation for the ATS makes it easier to transparently introduce path pruning (by eliminating nodes that correspond to impossible execution paths), path-sensitivity or context-sensitivity (e.g. multiple copies of a function's body for each code location from which the function may be called). The structure of the graph is made available via a standard graph interface and Fuse uses it to execute additional analyses with the added precision by associating constraints on the modified graph.

Fuse enables analysis interactions through a novel query interface which allows analyses to prove new constraints. The interactions between the analyses are organized as a client-server architecture where clients are static analyses asking questions and servers are static analyses providing answers to client's questions. Client queries are either a graph query or a set query where the graph queries (GetATSInit, GetATSFin) are used for traversing the graph and set queries (GetMemLoc, GetValue, GetCodeLoc) are used accessing the constraints at an ATS node. To access the set constraints at an ATS node, the clients provide a program segment and ask for the set of memory locations, values, or operations denoted by the program segment. The server provides an approximate interpretation of the program segment and returns abstract objects for the set query. The interactions are orchestrated by a composer entity which forwards the queries from clients to the servers and returns the abstract objects from the servers back to the clients.

Illustration Consider the source code in Fig. 1 requiring composition of multiple static analyses. The analyses constant propagation 1(b), unreachable code elimination 1(c), points-to analysis 1(d) and constant propagation 1(e) interacts using the Fuse query interface to determine the value of the expression $*p + 5$. Constant propagation 1(b) determines the outcome of the branch condition as true. Unreachable code elimination 1(c) queries constant propagation for GetValue($arr[1] == 3$). Constant propagation responds with an abstract value object $\{True\}$ which allows unreachable code analysis to eliminate the infeasible path. Points-to traverses the modified graph and computes the constraint $p \rightarrow arr[0]$. Constant propagation 1(e) queries points-to for GetMemLoc($*p$) using which it computes the value of $*p + 5$.

Key Advantages Fuse allows for a configurable program analyses where the developer picks the static analyses to be applied on a given program. The analysis composition is described as a composition command. The Fuse query interface allows analyses to communicate constraints in an API-agnostic way i.e., without being aware of analysis specific API such as LLVM's Alias Analysis interface

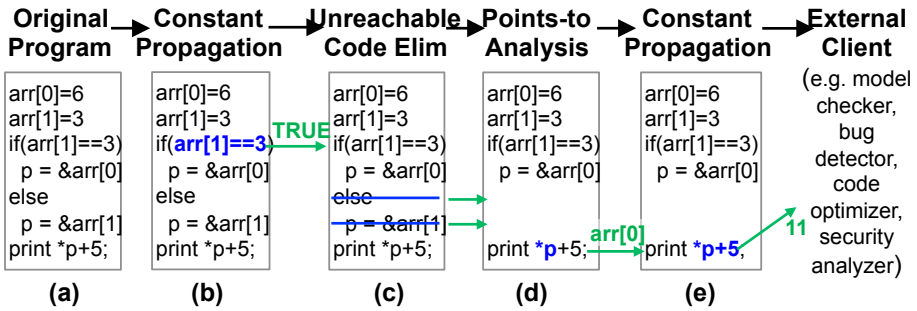


Fig. 1: Compositional Analysis by Fuse

[13]. Fuse simplifies analysis composition and allows modular abstractions to be introduced and flexibly composed with other analyses. We will leverage this capability in this paper to create a set of new analyses that model MPI semantics and compose our MPI analyses with traditional analyses that model non-MPI aspects of a program's behavior. This enables traditional static analyses to accurately analyze a wide range of properties (e.g. optimization potential or memory safety) of MPI applications.

2.2 Prior Work : Dataflow Analysis of MPI Programs

The fundamental challenge in reasoning about MPI programs is identifying the communication topology of the MPI program i.e., statically matching the send-receive operations. While this problem is undecidable in general, analyses compute approximations for it. The computed approximation must be sound (i.e., it must connect each pair of send and receive operations that may possibly match) but does not need to be complete (i.e some of the matched operations may not actually match in a real execution). Abstracting the communication topology requires: (1) an abstraction for the MPI operations and (2) a matching of the send abstractions with the receive abstractions. One simple abstraction for the communication topology is to group all the send operations into one equivalence class and all the receive operations into another equivalence class and match the two equivalence classes. While sound, this simple abstraction is imprecise for practical purposes.

MPI operations can be grouped into equivalence classes based on the static code location. Strout et al [20] use this abstraction to construct the MPI-ICFG where the matchings are computed by (i) grouping all the send operations from a send statement into an equivalence class (ii) grouping all the receive operations from a receive statement into an equivalence class (iii) connecting the send and receive equivalence classes. MPI-ICFG extends the interprocedural CFG by adding communication edges between the send and receive CFG nodes and the dataflow analysis is performed by propagating dataflow facts over the communication edges. The matchings are further refined using tags, datatypes and simple path constraints. This approach has two drawbacks. First, this approach uses a single CFG for modeling the message passing behaviors and consequently, the abstraction for MPI operations groups the MPI operations issued by different

processes executing the same path into a single equivalence class. For instance, consider the following code snippet

```
while(true) {
if(rank % 2 == 0)  MPI_Send(buf,.. rank+1);
else MPI_Recv(buf,..rank-1);
}
```

MPI-ICFG for the code snippet groups the send operations of all even processes into one equivalence class and the receive operations of all odd processes into another and connects the two equivalence classes. While sound, this abstraction allows communication between process 0 and process 3 which never happens in the original program. Furthermore, when the target expressions of MPI operations and path constraints are complex(left-neighbor, right-neighbor expressions), refinement of the send-receive matchings is cumbersome. Second, this approach ignores the matches-before ordering of MPI matching semantics, thereby losing opportunities for potential refinement.

Bronevetsky [2] constructed a parallel control-flow graph (pCFG) which improves the matching precision by grouping processes into equivalence classes and the equivalence classes were split at communication points or branch conditions and merged whenever they were identical. Message passing semantics are simulated by performing the analysis on the pCFG. To precisely match MPI operations in pCFG, the analysis would first block on corresponding MPI operations and the symbolic constraints on the target expression of a send must isomorphically match the symbolic constraints on the target expression of a receive operation. While scalable, this approach makes matching difficult when complex abstractions are used to describe the equivalence classes and target expressions evaluating to multiple values.

3 Approximating MPI Semantics

Our key insight is that computing an approximation of the communication topology with a reasonable precision on an unbounded number of processes is expensive and cumbersome. In our approach, we relax the unbounded constraint and fix the number of processes and compute an approximation for a fixed number of processes. This means that the program must be analyzed separately for each number of processes the user wants to run with; this can be done as a final compilation pass at job load-time.

Abstracting MPI Operations Our approach analyzes a concurrent MPI program with N processes using a cross-product of the ATSs given by $A_{T_1} \times A_{T_2} \times \cdots \times A_{T_N}$ where we associate an analysis instance for each process. For abstracting the MPI operations, we group MPI operations issued from an ATS node of a process into an equivalence class. Our abstraction differentiates the MPI operations issued by different processes, in different locations in the code, which allows ParFuse to compute more precise matchings than previous approaches. Furthermore, our abstraction allows ParFuse to compute process-sensitive value approximations (i.e., specific to each process) for the buffers of the MPI operations.

MPI Matching The challenge in matching the abstractions for MPI operations i.e., their equivalence classes, is that they must be matched following the out-of-order matching semantics of the MPI. Blocking operations are matched in the program order i.e., the order in which they are issued by the program. However, non-blocking operations are matched out of order i.e., two non-blocking operations to two different process are matched in any order. But two non-blocking operations to the same process are matched in the program order. MPI enforces this by the non-overtaking rule. One way to formalize the out-of-order matching of MPI is through matches-before relations. Vakkalanka et al [21] introduce intra matches-before relations (within a process) between the MPI operations issued by a process where the matches-before relations are due to the MPI matching semantics. The intra matches-before ordering between the operations is summarized as follows.

- Two blocking or non-blocking MPI point-to-point operations are matches-before ordered if they are send/receive to the same process and two operations are unordered if they are send/receive to different processes (non-overtaking rule).
- The non-blocking point-to-point operations are ordered before their respective `MPI_Wait` operations.
- MPI-specific strong-ordering points such as Barrier and Wait are matches-before ordered with any MPI operations that follow in program order.

Explicitly matching the equivalence classes of MPI operations following the matches-before ordering is cumbersome in practice. *We simplify matching by delegating the task to the MPI runtime.* In our approach, when a dataflow analysis reaches the ATS node of a send or a receive equivalence class it issues the operation to the MPI runtime where they are matched and exchange dataflow facts as the message payload.

While our approach simplifies MPI matching, it imposes three restrictions: First, we require that the matches-before ordering must be exactly determinable at compile time. Second, we require that the MPI operations are deterministic as the non-deterministic MPI operations have many possible matching choices that are not explored by the MPI runtime. Third, we require the divergent paths of the MPI processes where MPI send/receive operations are potentially issued to be loop-free. While these restrictions may seem onerous, we believe that composable static analysis of many MPI programs can be achieved under these restrictions, and that the data flow facts obtained under these restrictions can prove to be useful, while guaranteeing soundness. In particular, all of our MPI benchmarks yielded useful data flow facts under these restrictions. Furthermore, by introducing new MPI analyses (i.e., improving the MPI abstractions), our framework allows MPI-agnostic analyses to be MPI-aware on a larger set of applications. By fixing the number of processes and using the MPI runtime for matching provides ParFuse an unique opportunity towards building a parallel dataflow analysis framework for MPI programs where the framework is deployed as an MPI application.

Novelty Our approach improves upon the prior work where our abstraction for MPI operations allows ParFuse to compute more precise matchings. We differ from other approaches in matching the MPI abstractions where we delegate the matching to the MPI runtime. By performing the matching on the fly we do not require a priori construction of a communication graph for dataflow analysis. We realize our abstractions by implementing MPI specific analyses in the ParFuse framework. Our method allows analysis of each process to be carried out independently in parallel allowing ParFuse to scale better. Lastly, by building on the compositional principles of Fuse framework our work enables compositional reasoning of MPI programs.

4 MPI Analyses in ParFuse

Our approach for approximating MPI semantics is based on the following key ideas. First, we relax the unbounded process constraint by fixing the number of processes for the analysis. Second, we associate the MPI operations issued from an ATS node into a group. Third, we match the send-receive groups using the MPI runtime and exchange dataflow facts as message payload in-lieu of actual messages. We realize these novel ideas by modularly introducing MPI specific analyses into analysis composition using Fuse's compositional principles and transparently extending MPI support to existing MPI-agnostic analyses.

MPI Context Sensitivity (MCC) The role of MCC is to implement our abstraction for MPI operations by replacing the context-insensitive single copy of the ATS node for an MPI function body (empty stub) with multiple copies creating one copy for each call site. MCC operates on an input ATS and emits a MPI context sensitive ATS as its output. Observe that the ATS node is specific to each process and context of MPI operations at two different processes are not equal. The successors of MCC operate on the MPI context sensitive ATS allowing them to interpret the message passing semantics due to MPI operations issued from the same ATS node.

MPI Value (MV) MPI value provides semantic interpretation of MPI specific variables `rank` (the pid of the MPI process) and `size` (the total number of MPI processes). The values of `rank` and `size` are assigned by the MPI runtime when the program executes the functions `MPI_Comm_rank` and `MPI_Comm_size` respectively. The transfer function of MV semantically interprets the two MPI operations `MPI_Comm_rank` and `MPI_Comm_size` using MPI_COMM_WORLD as the argument and assigns positive integer constants to the variables `rank` and `size` as assigned by the MPI runtime. Analyses such as constant propagation when composed with MV, can infer new information based on the values computed by MV.

MPI Communication (MCO) MPI communication analysis provides semantic interpretations for the MPI communication operations such as `MPI_Send`, `MPI_Recv` by executing the operations. The message payload is the dataflow facts corresponding to the buffer of the MPI operations. MCO traverses the ATS of a previously completed analysis and at ATS nodes of MPI communication operations queries a prior analysis for the set of values denoted by the buffer using

the Fuse query interface function GetValue. The abstract value object obtained from a prior analysis is serialized using the boost serialization API [1]. The MCO executes the `MPI_Send` operation to transmit a serialized representation of the abstract value object as the message payload.

The envelope information of the MPI operations such as `target` and `tag` must be known to execute the MPI operations. As such, the execution of MCO must be preceded by a value analysis, such as constant propagation, which can compute this information. Then MCO can obtain the values for `target` and `tag` by the value analysis by calling GetValue on these variables. The MCO analysis requires that the values of the expressions `target` and `tag` evaluate to integer constants and aborts if the values are unknown. The restriction that the matches-before ordering of the MPI operations be exactly determinable at static time ensures that the values of the expressions exactly `target` and `tag` evaluate to integer constants. With the values for `target`, `tag` and `*buf` obtained from a prior analysis, MCO transmits the analysis information to the MPI runtime by executing the MPI operations.

MCO of the receiveing process deserializes the received information and caches the abstract value object at the call site of the `MPI_Recv` operation. The value approximation computed by a dataflow analysis is moved from the MCO of a sending process to the MCO of the receiving process. The portable query interface makes it possible for ParFuse to transparently add a dataflow analysis into the analysis composition and MCO propagates the dataflow facts from one process to another through the MPI runtime.

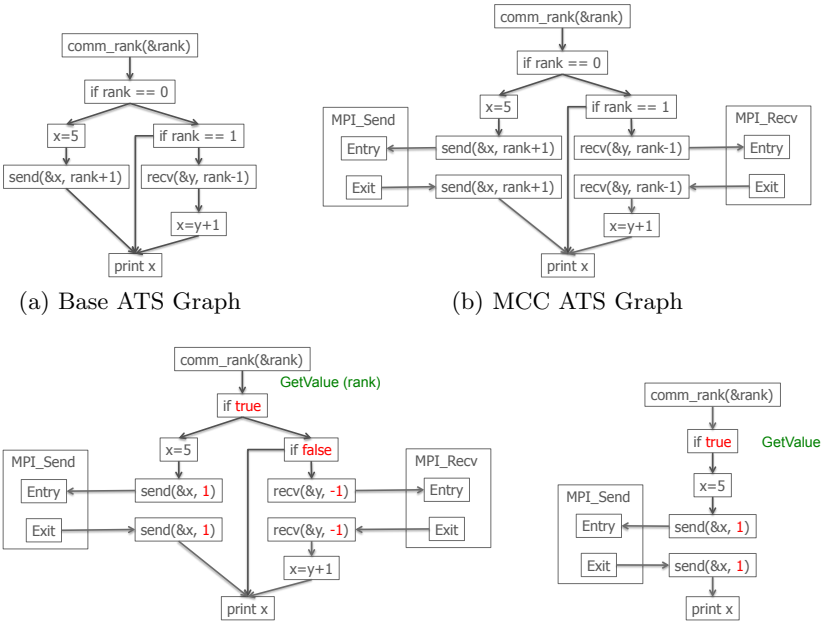
5 ParFuse Framework

We realize our methods for analyzing MPI programs in the ParFuse framework. ParFuse creates N instances of the Fuse compositional analysis framework, one for each ATS graph of a process. Each Fuse instance F_i executes an identical composition command containing a list of analyses that are composed using sequential composition where the analyses are executed one after the other. The ParFuse framework with N Fuse instances is deployed itself as an MPI program where each MPI process is a Fuse instance.

5.1 Analysis Composition Recipe

The standard dataflow analyses such as constant propagation(CP), points-to analysis(PT), unreachable code elimination(UC), calling context sensitivity(CCS), array analysis(ARR) have been observed to be useful to compose with our MPI analyses and will be the focus of our experiments, although any standard dataflow analyses can be composed in the ParFuse framework. First, these analyses benefit from MPI semantics provided by the MPI analyses. Second, these analyses are also instrumental in static determination of the matches-before ordering required by the MPI analyses. We illustrate this using a simple example shown in Fig. 2a. Let CC denote the composition command and we will add analyses to CC based on the demands of the MPI program. For notational convenience, we differentiate two instances of an analysis appearing in CC using subscripts.

For instance, CP_1 is the first instance of constant propagation and CP_2 is the second instance. Fig 2a shows the base ATS graph computed by the syntactic analysis(SYN) that transforms the source code to an ATS and provides syntactic constraints for memory, values and code locations. MPI analysis begins with the



(a) Base ATS Graph (b) MCC ATS Graph

(c) CP ATS Graph (d) UC ATS Graph

Fig. 2: Analysis Composition Recipe: Illustration

abstraction of MPI operations and we introduce MPI Context Sensitivity(MCC) into analysis composition which assigns a unique context to MPI operations based on the ATS node.

$$\mathsf{CC} = \mathsf{SEQ}(\mathsf{SYN}, \mathsf{MCC})$$

The ATS graph extended by MCC calling-site context for each MPI operation is shown in Fig. 2b. The ATS graph constructed by MCC is identical for both process 0 and process 1. For matching the send-receive groupings using the MPI runtime, determining the value of the target expressions $rank + 1$ and $rank - 1$ of the send and receive operations is critical. We will extend the composition command CC with points-to (rank is passed using pointers to `MPI_Comm_rank`), constant propagation (to propagate initial constants from MPI headers to MPI operations), MPI value (which interprets `MPI_Comm_rank`) and another constant propagation (propagate the rank value to target expressions) to determine the value of the target expressions of the send and the receive operation. Unreachable

code elimination (UC) is then added to prune infeasible paths.

$$\mathsf{CC} = \mathsf{SEQ}(\mathsf{SYN}, \mathsf{MCC}, \mathsf{PT}, \mathsf{CP}_1, \mathsf{MV}, \mathsf{CP}_2, \mathsf{UC})$$

The ATS graph for process 0 after CP_2 is shown in Fig. 2c and the ATS graph after UC is shown in Fig. 2d.

With the value of target expressions known, we can now introduce MPI communication analysis for matching the send-receive groupings using the MPI runtime and propagating the dataflow fact $\{x\} \rightarrow 5$ from the send call site of process 0 to the receive call site of process 1. This is illustrated in Fig. 3 The points-to composed earlier disambiguates the points-to relations at MPI

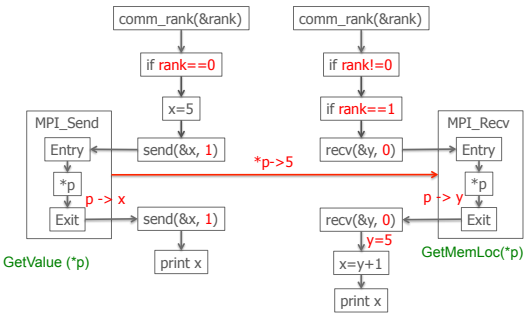


Fig. 3: MPI Runtime Matching Using MCO

call sites. MCO of process 0 queries CP_2 of process 0 for the values of x and propagates the value from the sender (rank=0) to the receiver(rank=1). The received value object is cached by the MCO of process 1. By adding another instance of constant propagation after MCO, the received value is propagated to the rest of the program.

$$\mathsf{CC} = \mathsf{SEQ}(\mathsf{SYN}, \mathsf{MCC}, \mathsf{PT}, \mathsf{CP}_1, \mathsf{MV}, \mathsf{CP}_2, \mathsf{UC}, \mathsf{MCO}, \mathsf{CP}_3) \quad (1)$$

The analysis composition CC described above is the basic recipe for analyzing MPI programs with message passing behaviors. The Fuse query interface allows for transparent exchange of dataflow facts between the analyses and the MPI communication analysis(MCO) communicates the dataflow facts through MPI message passing operations.

5.2 Illustration: Configurable Analysis of MPI Programs

We demonstrate the flexibility of our approach by proving a message passing dependent property shown in Fig. 4 that requires a non-trivial composition of standard dataflow and MPI analyses. ParFuse proves the property with two Fuse instances using the following analysis composition.

$$\mathsf{CC} = \mathsf{SEQ}(\mathsf{SYN}, \mathsf{MCC}, \mathsf{PT}, \mathsf{CP}_1, \mathsf{MV}, \mathsf{CP}_2, \mathsf{UC}_1, \mathsf{MCO}_1, \mathsf{CP}_3, \mathsf{UC}_2, \mathsf{CP}_4, \mathsf{MCO}_2, \mathsf{CP}_5)$$

```
if(rank == 0) {
  x = 2;
  MPI_Send(&x, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
  MPI_Recv(&z, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
}
else if(rank == 1) {
  MPI_Recv(&y, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
  if(y==2) z = 3;
  else z = 4;
  z = z+2;
  MPI_Send(&z, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD);
}
assert(z == 5);
```

Fig. 4: Configurable Program Analysis Example

The analysis composition consists of 13 instances of dataflow analyses which are composed sequentially one after the other. The two Fuse instances are executed independently of each other where MCO_1 propagates the value of x from the sender to the receiver. CP_3 propagates the received value to the branch condition, using which UC_2 eliminates the infeasible path. CP_4 on the receiver side computes precise value for z which is propagated back to the sender using MCO_2. Finally, the newly received value is propagated to the assert statement by CP_5. The compositional reasoning of ParFuse simplifies the task of proving the message passing dependent property which is otherwise cumbersome when using the existing non-compositional static analysis techniques for MPI programs. ParFuse makes it possible to configure program analyses to target the complexity of the program and the property to be proven. ParFuse also makes it easy to add new MPI analyses implementing different abstractions for MPI operations with varying cost/accuracy tradeoffs.

6 Experimental Results

We implemented the ParFuse framework in the ROSE [18] compiler infrastructure where our current implementation provides semantic interpretations for the following MPI operations: `MPI_Comm_rank`, `MPI_Comm_size`, `MPI_Barrier`, `MPI_Bcast`, `MPI_Send`, `MPI_Recv` and `MPI_Reduce`. Our goal is to evaluate the performance of realistic compositions of analyses that include our MPI analyses. Instead of timing the executions of MPI analyses, we will pick a concrete analysis task, compose a variety of standard dataflow analyses with MPI analyses to accomplish this task and measure the performance of the analysis execution for varying process counts. Two factors determine the choice of our analysis composition: (i) analysis composition required to accomplish the concrete analysis task (ii) analysis composition required to resolve the send-receive matching unambiguously. The communication topology of an MPI program is a useful property to be known statically with many applications such as debugging, overlapping

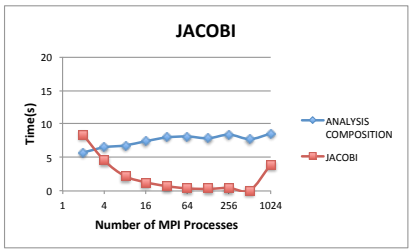
the computation with communication, optimal process placement etc. For the concrete analysis task, we will synthesize the communication topology of the MPI program as a DOT [10] graph. For this, we will compose MPI Dot Value(MDV) (a visualization tool) with our MPI analyses. MDV assigns unique id to the call sites of MPI send operations. The MPI Communication analysis(MCO) employs the Fuse API GetValue to obtain the unique id as a value object from MDV and transmits the value object to the matching receive call sites. The MDV at a receiving process employs the Fuse API GetValue and queries MCO to update the receive call sites with the received information. The communication graph in the DOT language is then synthesized by adding edges between send and receive ATS nodes using the received information. MDV also exemplifies the versatility of the ParFuse framework where non-dataflow facts such as unique ids are exchanged through our compositional principles.

We chose the following MPI programs: (i) Jacobi [16] iteration solving the Laplacian equation in two dimensions (ii) Heat [6] equation solver solving the time dependent heat equation in one dimension (iii) 2D Diffusion [9] solver solving the diffusion equation (iv) Prime [5] counting parallelized using MPI (v) Quadrature [4] approximating an integral using the quadrature rule for our study. The programs are of varying complexity in their source code requiring different analysis composition to unambiguously resolve the send-receive matching. Table 1 summarizes the analysis composition required for each benchmark to synthesize the communication topology as the DOT graph. The analyses are

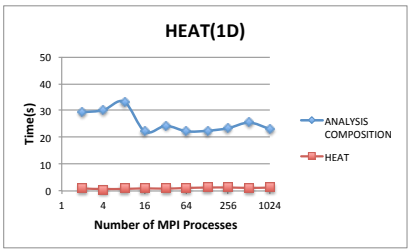
Table 1: Analysis Composition Summary

Benchmark	Analysis Composition
Jacobi	$\mathsf{SEQ}(\mathsf{SYN}, \mathsf{MCC}, \mathsf{PT}_1, \mathsf{CP}_1, \mathsf{MV}, \mathsf{CP}_2, \mathsf{UC}, \mathsf{ARR}, \mathsf{PT}_2, \mathsf{MDV}_1, \mathsf{MCO}, \mathsf{MDV}_2)$
Heat	$\mathsf{SEQ}(\mathsf{SYN}, \mathsf{CCS}, \mathsf{MCC}, \mathsf{CP}_1, \mathsf{ARR}_1, \mathsf{CP}_2, \mathsf{PT}_1, \mathsf{MV}, \mathsf{CP}_3, \mathsf{UC}, \mathsf{CP}_4, \mathsf{ARR}_2, \mathsf{PT}_2, \mathsf{MDV}_1, \mathsf{MCO}, \mathsf{MDV}_2)$
2D Diffusion	$\mathsf{SEQ}(\mathsf{SYN}, \mathsf{CCS}, \mathsf{MCC}, \mathsf{CP}_1, \mathsf{ARR}_1, \mathsf{CP}_2, \mathsf{PT}_1, \mathsf{MV}, \mathsf{CP}_3, \mathsf{UC}, \mathsf{CP}_4, \mathsf{ARR}_2, \mathsf{PT}_2, \mathsf{MDV}_1, \mathsf{MCO}, \mathsf{MDV}_2)$
Prime	$\mathsf{SEQ}(\mathsf{SYN}, \mathsf{CCS}, \mathsf{MCC}, \mathsf{CP}_1, \mathsf{PT}, \mathsf{MV}, \mathsf{CP}_2, \mathsf{UC}, \mathsf{MDV}_1, \mathsf{MCO}, \mathsf{MDV}_2)$
Quadrature	$\mathsf{SEQ}(\mathsf{SYN}, \mathsf{CCS}, \mathsf{MCC}, \mathsf{CP}_1, \mathsf{PT}, \mathsf{MV}, \mathsf{CP}_2, \mathsf{UC}, \mathsf{MDV}_1, \mathsf{MCO}, \mathsf{MDV}_2)$

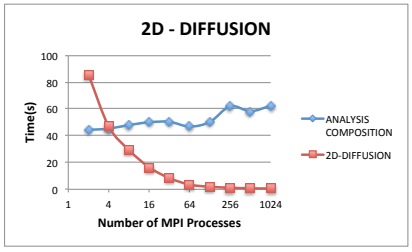
repeatedly applied as the reapplication discovers new information. For instance, to determine the memory locations denoted by the expression `arr[maxn/size]`, where `size` is assigned by `MPI_Comm_size`, constant propagation (CP) must be reapplied after MPI value (MV). The array analysis (ARR) composed after the CP queries CP for the value of the index expressions and consequently, determines the set of memory locations denoted by the the expression `arr[maxn/size]`. We evaluated the performance of our analysis composition with varying process counts up to 1024. The experiments were performed on a cluster with over 290 nodes(5104 cores, 32GB memory per node). The nodes are Intel Xeon (Sandybridge/Ivybridge E5-2670 and Haswell) processors and are connected through the Mellanox FDR Infiniband interconnect. Figure 5 shows the plots comparing



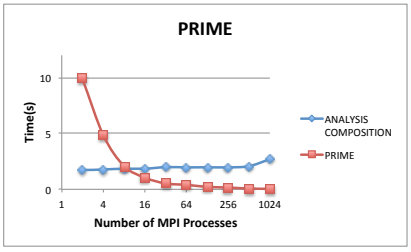
(a) Jacobi



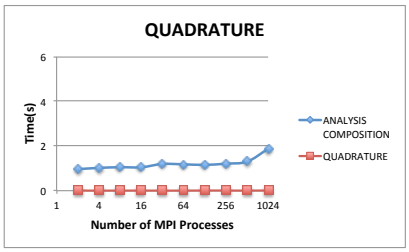
(b) Heat 1D



(c) 2D-Diffusion



(d) Prime



(e) Quadrature

Fig. 5: Scalability Evaluation

the average execution time (wall time) of the application and the analysis composition for each benchmark. The input problem size for the applications Jacobi, Prime and 2D Diffusion was not changed and with increasing process count they exhibit strong scaling whereas the input problem size for applications Heat and Quadrature was increased proportionally to the number of processes and they exhibit weak scaling. Figures 5a to 5e show a weak scaling for our analysis composition. Our results show that our approach scales linearly with increasing process counts. The challenge lies in picking the suite of analyses for disambiguating the communication and carrying out the analysis task for proving properties. Our current method is partial where the analyses are manually picked based on the complexity in the source code (arrays, pointers, mpi variables etc.). We repeatedly apply the analyses until the necessary information for disambiguating the communication is determined. We can overcome this challenge i.e., the phase ordering problem [7] by performing a tight composition which is computationally expensive and learning based approaches [12] that learns the characteristics of

the code being optimized and decides the best ordering of the analyses. Tight composition [14] evades the phase ordering problem by discovering all the information in one phase. Our preliminary implementation of tight composition reveals that this effort merits further investigation.

7 Related Work

In Section 2.2, we summarized prior work on dataflow analysis of MPI programs. In this section, we will summarize non-dataflow static techniques for analyzing MPI programs and dataflow analysis techniques for non-MPI message passing programs. McPherson et al [15] employed a tree based data structure for understanding the call sites of the MPI operations. The tree based structure allowed them to compute the value of target expressions when they involve `rank` and `size` on demand. They used a bit vector for a process sensitive computation of the target expressions of the MPI operations. Similar to our approach, they bound the number of processes and determine the values of target expressions and the message payload size at the call sites of the MPI operations. Their approach did not however match the send receive operations and simulate the message passing behaviors. Droste et al [8] implemented static checks purely based on the AST of the program. While the tool implements many useful checks based on the ATS, it was able to match MPI operations only when the target expression is trivial (constants) and the other arguments are exactly the same. Their technique solely relied on the AST producing sub-optimal results when matching point-to-point MPI operations. Reif [17] introduced a monotone lattice theoretic dataflow framework for communicating concurrent processes. Similar to our approach, Reif bounded the number of processes. The matching however was computed explicitly considering the semantics of the message passing operations. The framework was monolithic and was applied on a simpler message passing model than MPI.

8 Concluding Remarks

This paper presents a compositional approach towards building a dataflow analysis framework for analyzing MPI programs. Our approach builds on the compositional principles of the Fuse framework where abstractions for message passing operations are modularly introduced, by adding MPI specific analyses into analysis composition. We implemented a specific abstraction for MPI operations that allowed us to compute a more precise matching of the MPI operations than previous approaches. We adopted a simple solution for matching MPI abstractions by delegating it to the MPI runtime where our analysis is not burdened with simulating the complex matching semantics of MPI. Our compositional approach provides a mechanism to extend sequential dataflow analyses to work with MPI programs. Standard dataflow analyses can be transparently added into the analysis composition with the MPI analyses where the MPI analyses handles the task of abstracting message passing semantics. Our design choice of fixing the number of processes provided us a unique opportunity for carrying out the analysis of each process independently of each other, allowing the analyses to be

executed in parallel on a cluster and help our techniques scale for a large number of processes. The framework is also first in its kind where the dataflow facts are exchanged as message payload in lieu of actual messages.

Acknowledgments. This research was supported in part by NSF ACI 1148127, CCF 1439002 and DOE grant "Static Analysis using ROSE".

References

1. BOOST Team. *Boost Serialization API*, 2004.
2. G. Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. In *CGO*, 2009.
3. G. Bronevetsky, M. Burke, S. Aananthakrishnan, J. Zhao, and V. Sarkar. Compositional dataflow via abstract transition systems. Technical report, LLNL, 2013.
4. J. Burkardt. Quadrature using MPI. `http://people.sc.fsu.edu/~jburkardt/c_src/quad_mpi/quad_mpi.html`, 2010.
5. J. Burkardt. Couting Primes using MPI. `https://people.sc.fsu.edu/~jburkardt/c_src/prime_mpi/prime_mpi.html`, 2011.
6. J. Burkardt. Heat Equation solver in MPI-C. `http://people.sc.fsu.edu/~jburkardt/c_src/heat_mpi/heat_mpi.html`, 2011.
7. K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *SC*, 2002.
8. A. Droste, M. Kuhn, and T. Ludwig. MPI-Checker: Static Analysis for MPI. In *LLVM-HPC*, 2015.
9. Formal Verification Group at University of Utah. 2D Diffusion Equation Solver in MPI-C. `http://formalverification.cs.utah.edu/MPI_Tests/general_tests/small_tests/2ddiff.c`, 2009.
10. E. Gansner, E. Koutsofios, and S. North. *Drawing Graphs with DOT*, 2006.
11. T. Hoefler and T. Schneider. Runtime Detection and Optimization of Collective Communication Patterns. In *PACT*, 2012.
12. S. Kulkarni and J. Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. In *OOPSLA*, 2012.
13. C. Lattner. LLVM Alias Analysis Infrastructure. `http://llvm.org/docs/AliasAnalysis.html`.
14. S. Lerner, D. Grove, and C. Chambers. Composing Dataflow Analyses and Transformations. In *POPL*, 2002.
15. V. McPherson, Andrew J.and Nagarajan and M. Cintra. Static approximation of mpi communication graphs for optimized process placement. In *LCPC*, 2015.
16. MCS, Argonne National Laboratory. Simple Jacobi Iteration in C. `http://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiexmpl/src/jacobi/C/main.html`, 2000.
17. J. H. Reif. Data flow analysis of communicating processes. In *POPL*, 1979.
18. ROSE Compiler Team. *ROSE User Manual: A Tool for Building Source-to-Source Translators*.
19. D. Shires, L. Pollock, and S. Sprenkle. Program flow graph construction for static analysis of MPI programs. In *PDPTA*, 1999.
20. M. M. Strout, B. Kreaseck, and P. D. Hovland. Data-flow analysis for MPI programs. In *ICPP*, 2006.
21. S. Vakkalanka, A. Vo, G. Gopalakrishnan, and R. M. Kirby. Reduced execution semantics of MPI: from theory to practice. In *FM*, 2009.