Data-Flow Analysis for MPI Programs

Michelle Mills Strout
Colorado State University
Fort Collins, CO U.S.
mstrout@cs.colostate.edu

Barbara Kreaseck
La Sierra University
Riverside, CA U.S.
kreaseck@lasierra.edu

Paul D. Hovland
Argonne National Laboratory
Argonne, IL U.S.
hovland@mcs.anl.gov

Abstract

Message passing via MPI is widely used in single-program, multiple-data (SPMD) parallel programs. Existing data-flow frameworks do not model the semantics of message-passing SPMD programs, which can result in less precise and even incorrect analysis results. We present a data-flow analysis framework for performing interprocedural analysis of message-passing SPMD programs. The framework is based on the MPI-ICFG representation, which is an interprocedural control-flow graph (ICFG) augmented with communication edges between possible send and receive pairs and partial context sensitivity.

We show how to formulate nonseparable data-flow analyses within our framework using reaching constants as a canonical example. We also formulate and provide experimental results for the nonseparable analysis, activity analysis. Activity analysis is a domain-specific analysis used to reduce the computation and storage requirements for automatically differentiated MPI programs. Automatic differentiation is important for application domains such as climate modeling, electronic device simulation, oil reservoir simulation, medical treatment planning and computational economics to name a few. Our experimental results show that using the MPI-ICFG data-flow analysis framework improves the precision of activity analysis and as a result significantly reduces memory requirements for the automatically differentiated versions of a set of parallel benchmarks, including some of the NAS Parallel Benchmarks.

Key Words: *MPI, data-flow analysis, activity analysis, SPMD, MPI-ICFG*

1 Introduction

Message passing via MPI is widely used in parallel programs executing under the single-program, multiple-data (SPMD) model. MPI is a standard interface for message-passing parallel programs [30] written in C, C++, or Fortran that supports point-to-point communications (messages) and collective operations (broadcast, gather-scatter, reductions). Programs written in MPI typically employ single-program, multiple-data (SPMD) parallelism, with branches based on process rank used to achieve multiple instruction streams. MPI is ubiquitous, with vendor-supplied and open source implementations [12, 13, 5, 37] on essentially every parallel platform.

To support program analysis and understanding of MPI programs, there is a need for a data-flow analysis framework that models the semantics of message-passing, SPMD programs. Specifically, the communication operations induce data-flow from "sent" variables to "received" variables. This flow of data and the semantics associated with SPMD affects the precision and in some cases the correctness of nonseparable data-flow analyses. We define a nonseparable data-flow analysis as a data-flow analysis where the data-flow value for an individual construct (e.g., a variable in reaching constants) can not be determined disjointly from the data-flow values of the other constructs being analyzed. Analyses such as reaching definitions and liveness that are typically referred to as "bitvector" analyses are *separable* and do not require a special data-flow analysis framework for message-passing, SPMD programs. For example, reaching definitions do not flow between a send a receive since the send and receive may be in different processes, and the variable that receives the sent value is defined at the receive statement. Reaching constants (i.e. the analysis-only version of constant propagation) is the canonical example of a nonseparable data-flow analysis; the constant values of some variables can directly affect the constant values of other variables. For MPI programs, if all possible sends for a particular receive send the same constant then the received variable is equivalent to that constant. Handling the semantics of communication within SPMD programs is also needed in tools for program understanding, such as static slicing or chopping [2, 33] and program verification [36]; tools for finding security bugs, such as those that perform trust analysis [15]; and tools for program transformation, including performance and power optimizations such as bitwidth analysis [38] and automatic

```
begin program          (0)
  x = 0                (1)
  z = 2                (2)
  b = 7                (3)
  if (rank == 0) then  (4)
    x = x + 1          (5)
    b = x * 3          (6)
    send(x)            (7)
  else                 (8)
    receive(y)         (9)
    z = b * y          (10)
  endif                (11)
  f = reduce(SUM,z)    (12)
end program            (13)
```

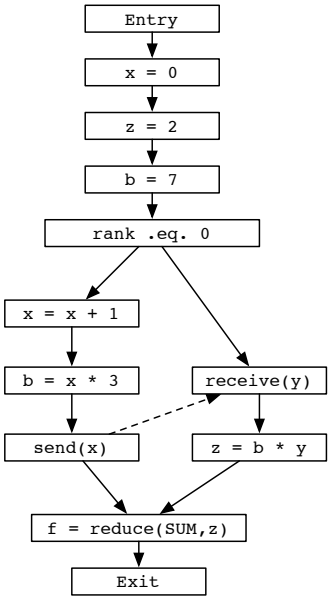

Figure 1. A small SPMD program and the corresponding MPI-CFG.

differentiation [4], which requires activity analysis.

Reaching constants is an example where modeling the semantics of message passing can improve precision, but is not necessary for correctness. Figure 1 presents a situation where the precision of reaching constants analysis improves with the use of communication semantics. The variable `y` will be assigned the constant value 1 due to the send of `x` and the corresponding receive into `y`. We found that constants typically are shared in SPMD programs without communicating them; therefore, performing reaching constants with communication and SPMD semantics is useful mainly for illustrative purposes.

Data-flow analysis that fails to model the SPMD nature of MPI programs may be incorrect. Again consider the simple program in Figure 1. If one attempts to take a forward slice to identify all statements influenced by the assignment `x = 0` in statement 1, using an analysis framework that does not consider the SPMD nature of the program, an erroneous result will be obtained. The framework will identify statements 1, 5, 6, and 7 as the only statements in the slice, when in fact statements 1, 5, 6, 7, 9, 10, and 12 should be in the slice. This situation cannot be remedied by changing only the behavioral model used for the communication library.

Despite the importance of SPMD data-flow analysis, we are unaware of any research that specifies how to perform data-flow analysis of message-passing, SPMD programs. Shires et al. [35] developed an extension to the control-flow graph representation called the MPI-CFG. The MPI-CFG represents the semantics of message passing by including communication edges between message-passing procedure calls. Figure 1 contains an example MPI-CFG with control flow edges represented as arrows with solid lines and a communication edge represented with dashed lines.

In Shires et al. [35], the authors argue that the MPI-CFG may be used as the basis for data-flow analysis of MPI programs, but the specification of such analysis is not provided. Simply treating communication edges as control-flow edges does not accurately model SPMD semantics. We also show that simple extensions to existing data-flow analysis frameworks are not effective for the analysis of message-passing, SPMD programs. To solve these problems, we present a method for performing data-flow analysis on SPMD MPI programs that propagates modified data-flow information over the communication edges in the MPI-CFG and our interprocedural extension, the MPI-ICFG. Our main contributions are as follows:

- the development of a data-flow analysis framework for the MPI-CFG [35] and our interprocedural extension the partially context-sensitive MPI-ICFG,
- the formulation of an important data-flow problem within our data-flow framework, and
- some experimental results that show significant precision improvements are possible.

This work was initially motivated by the need to perform activity analysis on MPI programs. Activity Analysis is a nonseparable data-flow anlaysis used in the context of automatic differentiation. Automatic Differentiation (AD) is code transformation that generates code for computing the sensitivies of inputs with respect to outputs for codes that simulate physical processes. Specifically, AD generates a program F' based on a program F, where F' computes the derivatives of a subset of F's outputs (the dependent variables) with respect to a subset of F's inputs (the independent variables). Such derivatives are useful for many application domains, such as climate modeling, electronic device simulation, oil reservoir simulation, medical treatment

planning and computational economics and more generally in applications that require partial differential equation solvers, numerical optimization, and numerical integration. Automatic differentiation works by mechanically applying the chain rule of differential calculus to the statements in a program. In the absence of activity analysis, a conservative strategy is to differentiate all statements and compute (and store) derivatives for all variables. However, because the independent variables are a subset of the input variables and dependent variables are a subset of the output variables, one can often tell *a priori* through static analysis that a variable either does not contribute to the derivatives of the dependent variables (is not useful) or has a zero derivative with respect to the independent variables (does not vary). Such variables are termed *inactive*, or *passive*, and need not have their derivatives computed. This approach can lead to substantial savings in time and memory [3].

The data-flow framework we have developed for activity analysis can be used more generally for any nonseparable data-flow analysis. Section 2 provides an overview of our MPI data-flow analysis framework and shows that simple extensions to existing data-flow analysis frameworks are insufficient. Section 3 more formally specifies a method for converting data-flow analysis problems to operate on the MPI-CFG. Section 4 extends the MPI-CFG to an MPI-ICFG for interprocedural analysis. Section 5 provides experimental results showing that when activity analysis is performed on the MPI-ICFG, it can reduce the space requirements for automatic differentiated code significantly in some benchmarks.

2 Handling the Semantics of Message-Passing SPMD Programs

Naive extensions to existing data-flow analysis frameworks that attempt to model the semantics of message-passing, SPMD programs are either incorrect, not scalable, or not as precise as our framework for data-flow analysis over the MPI-ICFG. The key insight is that data-flow information must be propagated over communication edges differently than it is propagated over control-flow edges. We use the forward phase of activity analysis to illustrate the shortcomings of other approaches.

Activity analysis necessitates a forward data-flow analysis that determines the set of variables that depend on selected inputs (the independents) *Vary*, and a backward data-flow analysis that determines the set of variables needed for the computation of selected outputs (the dependents) *Useful*. Variables in the intersection of *Vary* and *Useful* at a particular point in a program are *active*. A more complete description of activity analysis can be found in [31, 17]. For example, in Figure 1 assume that we want to create the derivative program that computes the derivative of `f` with respect to `x`. Considering message passing and SPMD semantics, the forward analysis *should* determine that the variables `x`, `y`, `z`, `b`, and `f` depend on the input `x`. The backward analysis *should* determine that variables `x`, `y`, `b`, and `z` are needed for the computation of `f`. A correct analysis should determine that at least the variables `x`, `y`, `z`, and `f` are active.

A naive and unrealistic use of a typical data-flow analysis framework results in an incomplete set of active variables, and therefore incorrect results. For the example in Figure 1, the relationship between the send of `x` and the receive of `y` is not modeled with a control-flow graph; therefore, the forward analysis determines that only the variables `x` and `b` depend on the variable `x`. The backward analysis finds that only `f`, `z`, and `y` are needed to compute `f`. The final intersection incorrectly concludes that there are no active variables within this program.

The Odyssée AD tool extends the typical data-flow analysis framework by modeling sends and receives as writes to and reads from global variables [8]. This models the communication that can occur, but not the fact that in SPMD execution model multiple processes may be executing the same program. In Figure 1, we can model the communication semantics by assigning `x` to a global variable `mpi_buff` and then copying the value of `mpi_buff` into `y` at the receive statement. However, since normal data-flow analysis assumes that the program could go down either side of the branch, but not both at the same time, activity analysis will simply conclude that `mpi_buff` is active and still incorrectly leave out other variables.

In our data-flow analysis framework, the data-flow information that is propagated over the communication edges is different from the information that is propagated over the control-flow edges. In Figure 1, there is a communication edge between the send of `x` and the receive of `y`. For the forward analysis, which determines which variables depend on the input `x`, the data-flow information propagated over the control-flow edges is the set of variables that $vary$ at that particular point in the program. For the communication edge, a boolean value is propagated. The value is true if the variable being sent (e.g. `x`) was in IN_{vary} for the send node, and false otherwise. The boolean value on the communication edge is used to determine whether the received variable should be included in the OUT_{vary} set for the receive node. The details of the framework are described in Section 3.

Other approaches correctly model the semantics, but are not as scalable or as precise as a data-flow analysis framework with communication edges. One strategy is to copy the control-flow graph for each process, provide each process with its own variable namespace, model communication with communication edges between separate control-flow graphs, and propagate data-flow information over com-

munication edges. This approach is similar to approaches used in data-flow frameworks for programs with explicit parallelism [32], and it provides accurate results, but is not scalable. An improvement on this approach is to analyze using only two copies of the control-flow graph (an idea also used within the context of performing cycle detection [24]). If the communication edges go between the two control-flow graphs, then the semantics of disjoint memory spaces is properly modeled, and overly conservative results are avoided. Our approach requires only one copy of the control-flow graph and provides results with equivalent precision.

In our experimental results, we compare activity analysis performed over an interprocedural control-flow graph (ICFG) [25] to activity analysis performed over an MPI-ICFG. Activity analysis can be solved correctly on an ICFG by using some global assumptions, specifically that all sends and receives write to and read from a global variable and that the global variable is an interesting input and output for the derivative code (i.e., initially put into the vary and useful sets). The global assumptions force all variables being sent that are vary to be active and all variables being received that are useful to be active. Similar global assumptions can be used to handle communication and SPMD semantics in other nonseparable analyses. As an example, trust analysis [15], which is used to detect security bugs in programs, can use the conservative assumption that the global variable modeling communication between sends and receives is untrusted; therefore, any variable that is received is assumed untrusted.

3 Intraprocedural Data-flow Analysis on the MPI-CFG

The MPI-CFG [35] represents message-passing between statements within the same procedure as communication edges. We present data-flow analysis for the MPI-CFG by first reviewing the data-flow analysis problem formulation for reaching constants on a typical control-flow graph and then extending that formulation for data-flow analysis over an MPI-CFG.

Intraprocedural data-flow analysis is formulated over a control-flow graph, which represents a procedure with a node for each statement[1] and edges between statements indicating possible control flow. Formally, a control-flow graph is specified as $CFG = \{V, E\}$, where V is the set of nodes in the graph, and E is the set of edges with $(n_1, n_2) \in E$ indicating that control can flow between the statement in node n_1 and the statement in node n_2. Each node n has a set of predecessors $pred(n)$ and a set of successors $succ(n)$, such that $(n_1, n_2) \in E$ implies that $n_1 \in pred(n_2)$ and $n_2 \in succ(n_1)$.

[1] This can be generalized to basic blocks.

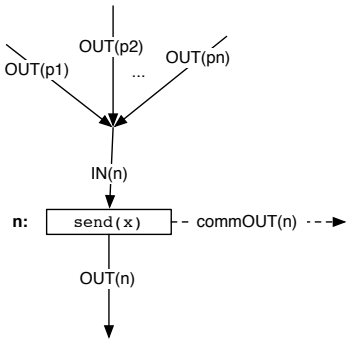

Figure 2. Control-flow edges and communication edges incident on a send node.

Data-flow analysis involves assigning $IN(n)$ and $OUT(n)$ sets to each node n in the control-flow graph CFG. The IN and OUT sets contain either program entities such as variables and statements or program entities paired with information from a partially ordered set referred to as the lattice. Iterative data-flow analysis recalculates the IN and OUT sets until convergence occurs. When two or more control paths in the CFG merge, a meet operation occurs between the lattice values for a particular program entity. For a forward data-flow analysis, the result of a meet operation is the $IN(n)$ set for the node n that directly succeeds the merging control-flow paths. In a forward analysis the $OUT(n)$ set for the node n is calculated by applying a transfer function $f_s(IN(n))$ to the $IN(n)$ set, which depends on the semantics of the statement s within n.

Reaching constants is a canonical example of a nonseparable, forward data-flow analysis. Each variable v is paired with a lattice value c_v. The possible constant lattice values are top $\top$, which indicates that no information is known about the variable; bottom $\bot$, which indicates the variable is not constant; or a constant value c, which indicates the variable holds the value c. Before performing the analysis, every IN and OUT set is inititialized with a pair $\langle v, c_v \rangle$ for each variable v. The IN set at the entry of the program is initialized with $\langle v, \bot \rangle$ and all other sets are initialized with $\langle v, \top \rangle$. The meet operation $\sqcap$ for reaching constants determines a lattice value for each variable when two OUT sets are merged. The result of the meet operation $\langle v, c_1 \rangle \sqcap \langle v, c_2 \rangle$ is $\langle v, c_r \rangle$, where c_r is as follows: if c_1 equals c_2, then c_r is c_1; if c_1 equals $\top$, then c_r is c_2; if c_2 equals $\top$, then c_r is c_1; otherwise, c_r is $\bot$. At an assignment statement, the transfer function evaluates the right-hand-side of the statement to a constant value c or $\bot$ and then pairs that resulting lattice value with the left-hand-side variable in the OUT set for

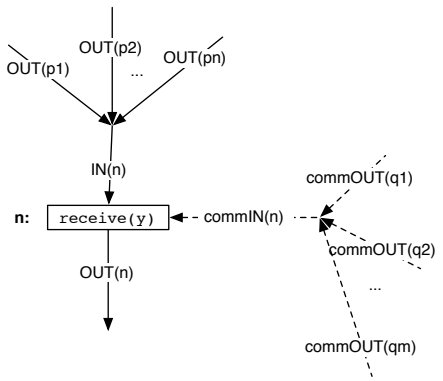

Figure 3. Control-flow edges and communication edges incident on a receive node.

the statement. When performing reaching constants on the MPI-ICFG in Figure 1, the variable `x` will be paired with the constant value 2 in the OUT set for statement `x=x+1`.

An MPI-CFG is specified as $CFG_{MPI} = \{V, E, C\}$, where V is the set of nodes in the graph, E is the set of control-flow edges, and C is the set of communication edges in the graph. Extending any forward, nonseparable data-flow analysis for operation over the MPI-CFG involves defining the communication transfer function f_{comm} that calculates the lattice value to propagate over outgoing communication edges based on the $IN(n)$ set for a send node and the variable being sent (see Figure 2). For reaching constants, the communication transfer function is $commOUT(n) = f_{comm}(IN(n)) = \{c_x | \langle x, c_x \rangle \in IN(n)\}$, where n is the node containing the statement `send(x)` and c_x is the lattice value assigned to the variable x in the $IN(n)$ data-flow set for the send node. When performing reaching constants on the MPI-ICFG in Figure 1, when performing reaching constants the lattice value 2 will be propagated over the communication edge between `send(x)` and `receive(y)`.

The transfer function for the receive statement must be defined so that it uses the lattice value propagated over all incoming communication edges as input. Assume that an MPI-CFG has been constructed such that there are communication edges between send and receive statements that conservatively estimate possible communications (see Figure 3). For each receive statement, we denote the set of possible sends nodes identified by the incoming communication edges as $commpred(n)$. In Figure 1, the `receive(y)` node only has the `send(x)` node in its $commpreds(n)$ set. For reaching constants, the transfer function for the receive node can be defined as $OUT(n) = (IN(n) - \{\langle y, c_y \rangle\}) \cup \{\langle y, \sqcap_{q \in commpred(n)} f_{comm}(IN(q)) \rangle\}$. When performing reaching constants on the MPI-ICFG in Figure 1, the OUT set for the node containing the `receive(y)` statement will include the following set of variables paired with lattice values: $\{\langle x, 0 \rangle, \langle z, 2 \rangle, \langle b, 7 \rangle, \langle f, \perp \rangle, \langle y, 2 \rangle\}$.

The approach used to define the transfer functions and communication transfer function for reaching constants can be used for other nonseparable data-flow analyses as well. Activity analysis is the analysis we implement for our experimental results. One phase of activity analysis is useful analysis, which is a backward data-flow analysis to determine the set of variables that are useful when computing a subset of output variables. If the variable is useful at the exit of a statement, then that variable is in the $OUT(n)$ set. The meet operator is set union. To initialize the analysis, all output variables of interest are inserted into the $IN(EXIT)$ set. The $OUT(n)$ set for node n is calculated by performing the meet operation between all the $IN(m)$ sets where m is in $succs(n)$. The transfer function for useful analysis calculates the $IN(n)$ set by making any variables that are being defined in a nodes statement be useful, if there are variables being used in the statement in a differentiable way. Note that the used variables might be arrays and that in the context of activity analysis, the variable(s) being defined in a statement do not depend on any of the variables used to index such arrays.

Extending useful analysis for the MPI-CFG necessitates passing a boolean value over the communication edge from the receive node to the send node to indicate whether the received variable is useful and therefore the sent variable is also useful. The communication transfer function applied to `receive(y)` is $commIN(n) = f_{comm}(OUT(n)) = \{true \mid y \in OUT(n)\}$. When performing useful analysis on the MPI-CFG in Figure 1, if the variable `f` is put in $OUT(Exit)$, then a true value will be propagated from the node containing the `receive(y)` statement to the node containing the `send(x)` statement. Also, the variable `x` will be in the IN set for the node containing the `send(x)` statement.

4 Interprocedural Data-Flow Analysis of SPMD MPI Programs

Most MPI programs do not have all of their MPI calls in one procedure; therefore, interprocedural data-flow analysis is necessary. We generate an interprocedural control-flow graph (ICFG) [25], augment the ICFG with communication edges that can cross procedure boundaries to generate the MPI-ICFG, and provide context-sensitivity for certain MPI calls and their callers. We chose an ICFG versus the typical call graph with a CFG per procedure, because the communication edges would result in the separate CFGs interacting with each other within the data-flow analysis in ways not

captured by the call graph relationships.

4.1 Construction of the MPI-ICFG

The construction of the MPI-ICFG applies to programs that use MPI. The MPI standard specifies interfaces for C, C++, and Fortran. In our experiments we perform analysis of Fortran programs. We build the MPI-ICFG by first constructing an ICFG and then adding communication edges between possible send/ isend and receive/ireceive pairs, among all calls to broadcast, and among all calls to reduce. We perform an interprocedural reaching constants analysis and perform a matching using the MPI semantics to reduce the number of communication edges that are conservatively necessary. For broadcast and reduce, the root parameters must match if they statically evaluate to constants. For send and receive pairs, the tag and communicator must match if they statically evaluate to constants. Additional heuristics for reducing the number of communication edges are described in [35], but were not used in our experiments.

The precision of data-flow analysis over the ICFG is affected by the precision of the alias analysis, the lack of context-sensitivity in the ICFG, and the global assumptions made at MPI calls. The precision of data-flow analysis over the MPI-ICFG is affected by the same things, but it has better precision in terms of the MPI calls as long as there is less than full connectivity in the communication edges between sends, receives, reduce, and broadcast operations.

The precision benefits of propagating data over the communication edges can be lost when there is not enough context-sensitivity in the call path involving the MPI sends and receives. We provide partial context-sensitivity within the ICFG and the MPI-ICFG by cloning the ICFG nodes for the MPI send and receive stub routines for each call site. In Table 1 this is referred to as clone level zero. For some of the benchmarks, the send and receive calls are occur within layers of wrapper routines, which are in turn called in multiple locations. Clone levels greater than zero indicate the number of levels in the call graph away from MPI send and receive that routines are marked for cloning in the ICFG. In our experimental results, we used the lowest level of cloning that experienced the best possible precision on the ICFG. In a practical implementation, the necessary level of cloning could be determined by inspecting the call graph to determine the wrapper depth around MPI sends and receives.

4.2 Complexity of Data-Flow Analysis

The depth of the MPI-ICFG multiplied by the number of variables provides an upper bound on the number of passes required for convergence. The depth is difficult to calculate because the MPI-ICFG is generally irreducible due to the communication edges. It can be shown that computing the depth of an irreducible graph is NP-complete [29]. Nonetheless, our experimental results show that the convergence is comparable to the more conservative analysis over the ICFG (see Table 1).

4.3 Implementation of Data-Flow Analysis

Data-flow analysis frameworks for CFG's are typically implemented so that only the transfer and meet operations must be specified [9, 16, 40]. Data-flow analysis over ICFGs also requires a specification of how information is mapped from the caller to the callee, and vice versa. These same operations must be specified for data flow over an MPI-ICFG. The only new methods needed for analysis over an MPI-ICFG are the communication transfer function, the transfer functions for send and receive statements, and a meet operation for the values propagated over the communication edges.

5 Experimental Results

5.1 Methodology

We implemented the construction of the MPI-ICFG, the data-flow framework for an MPI-ICFG, and activity analysis using the OpenAnalysis toolkit [39] coupled with the Open64/SL compiler infrastructure [34] and used the data-flow analysis framework to apply activity analysis to various benchmarks. The NAS parallel benchmarks [1], labeled NASPB, were obtained from http://www.nas.nasa.gov/Software/NPB/; the benchmark labeled SOR is an implementation of successive overrelaxation developed by one of the authors [18]; and the benchmark labeled Biostat is a parallelized version of a biostatistical analysis function provided by D. Spiegelman [19]. Sweep3d [26] is a benchmark code derived from a real application, 3D Discrete Ordinates Neutron Transport, which solves a geometry neutron transport problem. Table 1 summarizes our benchmarks.

Each benchmark is a unique combination of source, context routine (shown in parentheses), and independent and dependent variables. For example, LU-1 refers to the source `NASPB LU` with context routine `rhs`, independent variable `frct` and dependent variable `rsd`. As described in Section 2, activity analysis requires selecting a subset of the input variables to a procedure within the program as the independent variables and a subset of the output variables as the dependent variables. For each benchmark, we selected at least one reasonable set of independent and dependent variables and a context routine. The context routine is a subroutine within the program that it makes sense to automatically differentiate. The ICFG and MPI-ICFG contain the routines that are called either directly or indirectly by

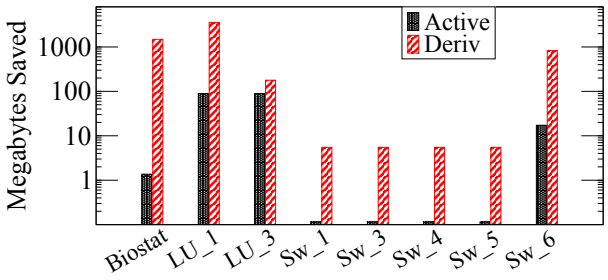

Figure 4. Activity Analysis Results: Number of megabytes saved per benchmark from MPI-ICFG over ICFG activity analysis for both the Active set and within the Derivative code.

the context subroutine. The Biostat and SOR problems had previously been differentiated using the ADIFOR automatic differentiation tool and appropriate independent and dependent variables were known. The NAS Parallel Benchmarks considered primarily solve a linear system. Therefore, we selected as independent variables one or more of the scalar quantities used to compute the righthand-side (rhs) vector or the rhs vector itself. We selected as the dependent variable either the rhs vector, the solution vector, or the residual of the solution vector. For the context routine, we selected either the subroutine used to form the rhs vector or the subroutine used to set up and solve the linear system.

We performed activity analyses over ICFGs and MPI-ICFGs on all the benchmarks. When using the ICFG, the benchmarks were augmented with a global variable to model possible communication between all send and receive pairs, and the global variable was declared both independent and dependent within the context routine. We recorded the number of iterations for convergance as well as the number of active bytes found by each analysis. The decrease in number of active bytes is given as a percentage and is calculated by subtracting the MPI-ICFG-Active-Bytes from the ICFG-Active-Bytes and dividing by the ICFG-Active-Bytes. Using the number of active bytes and the number of independent variables, it is possible to determine the storage that will be needed within the derivative code using the formula $\text{DerivBytes} = (\text{number of independents}) * (\text{ActiveBytes})$. This is because in the derivative code, it will be necessary to maintain the derivative of each active variable or array element (typically a double) with respect to each independent variable.

5.2 Effect on Activity Analysis

Each analysis determines an active symbol list and the size in bytes of active symbols. Figure 4 shows the possible storage savings when activity analysis is performed over the MPI-ICFG versus the ICFG. Storage savings only occur for eight of the benchmarks, but the amount of storage saved for those benchmarks is significant. Table 1 details the differences between activity analysis over the ICFG and analysis over the MPI-ICFG on all of the benchmarks.

The most dramatic difference is seen in Biostat and Sweep3D. In the Biostat problem, using the MPI-ICFG allows us to determine that a large data array (in this small test problem, an array of approximately 300,000 floating-point values) is not active and therefore does not need derivatives [18]. For this small example, the resulting memory savings would be approximately 1.5 gigabytes; for the real problem, the savings would be hundreds of gigabytes [19, 3]. In addition to the space savings, there would be significant time savings, since otherwise all of this useless data would need to be broadcast from the root processor to all other processors. The savings seen in the LU set of benchmarks is also due to a large array being categorized as inactive.

These results are specific to activity analysis and do not easily generalize to other nonseparable data-flow analyses. For example, the trust analysis described in [15] will experience precision improvements on the MPI-ICFG whenever trusted information is sent and received. Trust analysis is not relevant to the benchmarks used in these experiments, because the programs do not receive any input from untrusted sockets. The important conclusion to draw from the activity analysis experiments is that there are situations where a data-flow analysis that explicitly handles communication edges can derive more precise results.

5.3 Convergence

The column labeled Iter in Table 1 shows that the number of iterations over the MPI-ICFG is slightly larger than the number of iterations over the ICFG. The worst-case number of iterations is the depth of the graph multiplied by the number of variables. Since the MPI-ICFG includes communication edges, it will always have a depth that is greater than or equal to the depth of the ICFG. The actual number of iterations for both the ICFG and the MPI-ICFG do not show worst-case behavior.

6 Related Work

The work most closely related to the data-flow framework presented in this paper is that of Reif and Smolka [32]. They formalize a data-flow framework for a collection of communicating processes. The flow of data over what they refer to as communication channels is similar to the flow of data over the communication edges in our framework. The main difference is that they do not handle single-program,

Table 1. Number of iterations, number of active bytes and number of DerivBytes for ICFG and MPI-ICFG Activity analyses.

Bench-mark	Source	Clone-level	IND	DEP	Analysis	Iter	Active Bytes	# of Indeps	Deriv Bytes	% Decrease
Biostat	Spiegelman: Biostat (lglik3)	0	xmle	xlogl	ICFG	12	1441632	1089	1569937248	
					MPI-ICFG	12	9016		9818424	99.37%
SOR	Hovland: SOR (mainsor)	0	omega	resid	ICFG	13	3038136	1	3038136	
					MPI-ICFG	17	3030104		3030104	0.26%
CG	NASPB: CG (conj_grad)	0	x	z	ICFG	14	240048	1	240048	
					MPI-ICFG	18	240048		240048	0.00%
LU-1	NASPB: LU (rhs)	1	frct	rsd	ICFG	18	187194472	40	7487778880	
					MPI-ICFG	19	93636000		3745440000	49.98%
LU-2	NASPB: LU (ssor)	2	omega	rsd	ICFG	23	145901208	1	145901208	
					MPI-ICFG	30	145901168		145901168	0.00%
LU-3	NASPB: LU (rhs)	1	tx1,tx2	rsd	ICFG	18	140376488	2	280752976	
					MPI-ICFG	18	46818016		93636032	66.65%
MG-1	NASPB: MG (mg3P)	3	r	u	ICFG	16	647487912	1	647487912	
					MPI-ICFG	18	647487896		647487896	0.00%
MG-2	NASPB: MG (psinv)	1	c	u	ICFG	16	16908656	4	67634624	
					MPI-ICFG	17	16908640		67634560	0.00%
Sw-1	ASCI: Sweep3d (sweep)	2	w	flux	ICFG	24	18120784	48	869797632	
					MPI-ICFG	23	18000048		864002304	0.67%
Sw-3	ASCI: Sweep3d (sweep)	2	w	leakage	ICFG	23	120984	48	5807232	
					MPI-ICFG	25	248		11904	99.80%
Sw-4	ASCI: Sweep3d (sweep)	2	weta	leakage	ICFG	23	120840	48	5800320	
					MPI-ICFG	25	104		4992	99.91%
Sw-5	ASCI: Sweep3d (sweep)	2	w	flux, leakage	ICFG	22	121032	48	5809536	
					MPI-ICFG	22	296		14208	99.76%
Sw-6	ASCI: Sweep3d (sweep)	2	weta	flux, leakage	ICFG	22	18120840	48	869800320	
					MPI-ICFG	22	104		4992	99.99%

multiple data (SPMD) semantics. In their framework, each process is modeled with a separate control-flow graph.

Data-flow frameworks have been developed for various types of shared-memory parallelism [7, 14, 20, 22, 23, 21, 27]. Other research [28, 10] provides a model for data-flow analysis of concurrent programs with Ada-style rendezvous. We are aware of only Krishnamurthy and Yelick [24] looking at analysis for single-program, multiple-data programs. They present methods for performing interference dependence analysis of SPMD programs with a global shared address space by making two copies of the control-flow graph.

Several automatic differentiation tools support MPI, but activity analysis is typically limited. ADIFOR 3.0 forces all floating-point variables passed as an argument to an MPI call to be active [6]. TAMC and TAF allow the user to specify activity information for library calls [11], including calls to MPI. The programmer must have a deep understanding of the context of MPI calls or make the conservative assumption that all variables communicated via MPI are active. Odyssée and Tapenade employ a model that induces a dependence from all sent variables to all received variables through assignment to/from a single global variable [8]. This model ignores the SPMD nature of MPI programs and thus may fail if a branch on rank occurs prior to communication and outside of any loops.

7 Conclusions

MPI programs are a significant portion of all parallel programs. We describe a data-flow analysis framework for the MPI-CFG [35] and the MPI-ICFG. The key feature of these intermediate representations is that they model the communication between MPI calls and therefore the associated data-flow analysis is capable of modeling communication and SPMD semantics. Data-flow analysis that propagates information over communication edges is relevant to non-separable analyses such as reaching constants, activity analysis, slicing, bitwidth analysis [38], and trust analysis [15]. Our experimental results show that activity analysis per-

formed over the MPI-ICFG has a convergence rate comparable to a conservative analysis over the ICFG and that using the MPI-ICFG data-flow analysis framework improves the precision of activity analysis. Although we focus on MPI programs, the MPI-ICFG and the associated data-flow analysis framework are applicable to any SPMD parallel program that uses messages for communication.

8 Acknowledgments

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-Eng-38. We would like to thank Nathan Tallent and Luis Ramos for their programming efforts that contributed to the results for this paper. We would also like to thank Gail Pieper for proofreading several revisions.

References

[1] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, Nasa Ames Research Center, Moffet Field, CA, 1995.

[2] D. W. Binkley and K. B. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[3] C. Bischof, A. Carle, P. Khademi, and A. Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

[4] C. H. Bischof, P. D. Hovland, and B. Norris. Implementation of automatic differentiation tools. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 98–107, January 2002.

[5] G. Burns, R. Daoud, and J. Vaigl. LAM: An open cluster environment for MPI. In *Proceedings of the Supercomputing Symposium*, pages 379–386, 1994.

[6] A. Carle. *Automatic Differentiation*, pages 701–719. Morgan Kaufmann Publishers, 2003.

[7] J. Cheng. Dependence analysis of parallel and distributed programs and its applications, 1997.

[8] P. Dutto and C. Faure. Extension of Odyssée to the MPI library: The direct mode. Rapport de Recherche 3715, INRIA, 1999.

[9] M. B. Dwyer and L. A. Clarke. A flexible architecture for building data flow analyzers. In *Proceedings of the 18th International Conference on Software Engineering*, pages 554–564. IEEE Computer Society Press, 1996.

[10] M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, and G. Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Transactions Software Engineering Methodology*, 13(4):359–430, 2004.

[11] R. Giering and T. Kaminski. Applying TAF to generate efficient derivative code of Fortran 77-95 programs. In *Proceedings of GAMM 2002*, 2002.

[12] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[13] W. D. Gropp and E. Lusk. User's guide for `mpich`, a portable implementation of MPI. Technical Report ANL-96/6, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[14] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 159–168, 1993.

[15] S. Guyer and C. Lin. Client-driven pointer analysis. In *Static Analysis Symposium (SAS)*, 2003.

[16] M. W. Hall, J. M. Mellor-Crummey, A. Carle, and R. G. Rodriguez. FIAT: A framework for interprocedural analysis and transfomation. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 522–545, 1994.

[17] L. Hascoet, U. Naumann, and V. Pascual. "to be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8), 2004.

[18] P. Hovland. *Automatic Differentiation of Parallel Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[19] P. Hovland, C. Bischof, D. Spiegelman, and M. Casella. Efficient derivative codes through automatic differentiation and interface contraction: An application in biostatistics. *SIAM Journal on Scientific Computing*, 18(4):1056–1066, 1997.

[20] J. Knoop. Constant propagation in explicitly parallel programs. In *Proceedings of Euro-Par, LNCS 1470*. Springer, 1998.

[21] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):268–299, 1996.

[22] J. Krinke. Static slicing of threaded programs. In *Proceedings ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 35–42, 1998.

[23] J. Krinke. Context-sensitive slicing of concurrent programs. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2003.

[24] A. Krishnamurthy and K. A. Yelick. Optimizing parallel SPMD programs. In *Languages and Compilers for Parallel Computing*, pages 331–345, 1994.

[25] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[26] Lawrence Livermore, Los Alamos, and Sandia National Laboratories. The accelerated strategic computing initiative (ASCI) sweep3d benchmark code, 1995. http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html.

[27] J. Lee, D. A. Padua, and S. P. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1999.

[28] D. Long and L. A. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *TAV4: Proceedings of the Symposium on Testing, Analysis, and Verification*, pages 21–35, New York, 1991. ACM Press.

[29] R. McConnell. Personal communication, 2005.

[30] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.

[31] U. Naumann. Reducing the memory requirement in reverse mode automatic differentiation by solving TBR flow equations. In P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Computational Science – ICCS 2002, Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II*, volume 2330 of *LNCS*, pages 1039–1048. Springer, 2002.

[32] J. H. Reif and S. A. Smolka. Data flow analysis of distributed communicating processes. *International of Journal Parallel Programming*, 19(1):1–30, 1990.

[33] T. Reps and G. Rosay. Precise interprocedural chopping. *ACM SIGSOFT Software Engineering Notes*, 20(4):41–52, 1995.

[34] Rice University. Open64 project. http://www.hipersoft.rice.edu/open64/.

[35] D. Shires, L. Pollock, and S. Sprenkle. Program flow graph construction for static analysis of mpi programs. In *International Conference on Parallel and Distributed Processing Techniques and Applications, (PDPTA 99)*, June 1999.

[36] S. Siegel. Personal communication, 2004.

[37] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings of the 10th European PVM/MPI Users' Group Meeting*, volume 2840 of *LNCS*, pages 379–387. Springer-Verlag, 2003.

[38] M. Stephenson, J. Babb, and S. Amarasinghe. Bidwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 108–120, 2000.

[39] M. M. Strout, J. Mellor-Crummey, and P. Hovland. Representation-independent program analysis. In *Proceedings of the The sixth ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, September 5-6 2005.

[40] S. W. Tjiang and J. L. Hennessy. Sharlit–a tool for building optimizers. In *The ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1992.