Data-Flow Analysis for MPI Programs

Michelle Mills Strout
Colorado State University
1873 Campus Delivery
Fort Collins, CO 80523
mstrout@cs.colostate.edu

Barbara Kreaseck
La Sierra University
4500 Riverwalk Parkway
Riverside, CA 92515
kreaseck@lasierra.edu

Paul Hovland
Argonne National Laboratory
9700 S. Cass Ave.
Argonne, IL 60439
hovland@mcs.anl.gov

ABSTRACT

Message passing via MPI is widely used in single-program, multiple-data (SPMD) parallel programs. Data-flow analysis frameworks that respect the semantics of message-passing SPMD programs are needed to obtain more accurate and in some cases correct analysis results for such programs. We qualitatively evaluate various approaches for performing data-flow analysis on SPMD MPI programs and present a method for performing interprocedural data-flow analysis on the MPI-ICFG representation. The MPI-ICFG is an interprocedural control-flow graph (ICFG) augmented with communication edges between possible send and receive pairs.

We discuss in detail two analyses that potentially benefit from propagating information over communication edges: reaching constants and activity analysis. Constants can be shared in SPMD programs without communicating them; therefore, performing reaching constants over the MPI-ICFG is useful mainly for illustrative purposes. Activity analysis is a domain-specific analysis used to reduce the computation and storage requirements for automatically differentiated MPI programs. Our experimental results show that activity analysis performed over the MPI-ICFG has a convergence rate comparable to a more conservative version of the analysis performed on an ICFG. Also, using the MPI-ICFG data-flow analysis framework improves the precision of activity analysis and significantly reduces memory requirements for the automatically differentiated versions of some 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 [28] 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, 6, 34] on essentially every parallel platform.

In order to support program analysis and understanding, data-flow analysis frameworks are needed that recognize the semantics of MPI. Specifically, the communication operations induce data-flow and data dependencies from "sent" variables to "received" variables. This flow of data affects the precision and in some cases the correctness of nonseparable data-flow analyses. Reaching constants, is one example of a nonseparable data-flow analysis; 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 also is needed in tools for program understanding, such as static slicing or chopping [2, 30] and program verification [33]; tools for finding security bugs, such as those that perform trust analysis [15]; and tools for program transformation, including performance and power optimization such as bitwidth analysis [35] and automatic differentiation [4, 5], 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. We found that constants typically are shared in SPMD programs without communicating them; therefore, performing reaching constants over the MPI-ICFG is useful mainly for illustrative purposes. 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`.

Data-flow analysis that fails to take into account 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,

```
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)
```

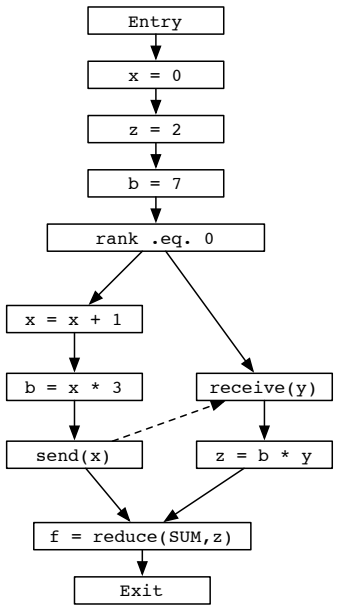

Figure 1: A small SPMD program (left) and the corresponding MPI-CFG. Analysis that ignores the SPMD nature may be incorrect. For example, reaching constants using the communication edges in the MPI-CFG will be more accurate.

and 12 should be in the slice. This situation cannot be remedied by changing only the behavioral model used for the communication library.

Little work appears to have been done on analysis of SPMD message-passing programs. Typical data-flow analysis treats calls to a message-passing library like any other function or procedure call. Without the semantics of communication relationships between MPI function calls, data-flow analysis must treat MPI function calls as they would any other function. Since an MPI receive assigns to its input parameter via a read from a buffer, nothing will be statically assumed about the contents of that buffer in existing data-flow analysis frameworks. For example, it is possible to indicate via a behavioral model or stub function that an MPI_RECV defines the buffer receiving the data, but within the context of a data-flow analysis characteristics of the buffer will not always be semantically correct due to SPMD semantics.

Shires et al. [32] developed an extension to the control flow graph representation called the MPI-CFG. The MPI-CFG represents the semantics of MPI by including communication edges between message-passing procedure calls. Unlike most other models for concurrent programs, the MPI-CFG requires only one control-flow graph. If it were necessary to represent each process explicitly, the analysis would not be scalable to realistic computations involving thousands of processes. 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 this paper, 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 extension, the MPI-ICFG. Section 2 qualitatively compares augmenting data-flow analysis using propagation over communication edges with other analysis methods. Section 3 introduces 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, discusses the convergence rate of data-flow analysis over the MPI-ICFG, and describes a data-flow analysis framework to implement such analysis. Section 5 provides experimental results that indicate 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. METHODS FOR ANALYSIS OF MPI PROGRAMS

Other approaches to performing data-flow analysis on SPMD MPI programs are typically incorrect, not as accurate as ours, or not scalable. We use the forward phase of activity analysis to illustrate the shortcomings of other approaches.

Activity Analysis is used in the context of automatic differentiation. Automatic differentiation is a technique for generating 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 algorithms in scientific computing, including those for solution of differential equations, minimization of nonlinear functions, and uncertainty quantification. 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 dependent 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].

Activity analysis involves a forward analysis that determines the set of variables that depend on selected inputs (the independents) OUT_{vary} and a backward analysis that determines the set of variables needed for the computation

of selected outputs (the dependents) IN_{useful}. The vary and useful results are flow sensitive. Variables in the intersection of OUT_{vary} and IN_{useful} at a particular point in a program are active and require additional data structures and code for the calculation of derivatives. For example, in Figure 1 assume that we want to create the derivative program that computes the derivative of `f` with respect to `x`. 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`. The active variables are `x`, `y`, `z`, and `f`. The variable `b` is not active because at no point in the program is it both vary and useful. On the left branch `b` is vary, and on the right branch `b` is useful. A more complete description of activity analysis can be found in [29, 17].

Applying activity analysis to an MPI program without recognizing the communication and SPMD semantics results in an incomplete set of active variables. For the example in Figure 1, if the relationship between the send of `x` and the receive of `y` is unknown, then the forward analysis will determine 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.

One possible solution is to model sends and receives as writes to and reads from global variables. 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 copy 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. To fix this situation, we could add a fictitious outer loop around the program to simulate the SPMD paradigm, but then the results may be overly conservative, because data-flow information associated with other variables ends up being carried by the loop as well.

A second strategy for fixing the approach where sends and receives are modeled with writes to and reads from a global variable is to treat the the communication edges in the MPI-CFG as if they were normal control-flow edges. This approach introduces spurious activity because the communication edges represent communication between processes in disjoint memory spaces. In Figure 1, if all of the variables in the vary set were propagated over the communication edge then the variable `b` would end up vary and useful before the statement `z = b * y` and therefore unnecessarily active.

A third strategy is to copy the control-flow graph for each process, provide each process with its own variable namespace, model communication with global shared variables, and propagate data-flow information over communication edges. This approach provides accurate results, but is not scalable.

A fourth strategy involves analyzing the program 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 ICFG 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 and received to be active if even one variable being sent is vary and one being received is useful. The main contributor to loss of precision using the MPI-CFG model is communication nodes that have more than one incoming or outgoing communication edge. As the results in Section 5 show, however, in some cases data-flow analysis over the MPI-ICFG can result in significant precision improvements.

3. INTRAPROCEDURAL DATA-FLOW ANALYSIS ON THE MPI-CFG

The MPI-CFG [32] 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 to data-flow analysis over an MPI-CFG.

Reaching constants is a well-known forward data-flow analysis. Each variable is paired with a value from the constant lattice, where top $\top$ indicates that no information is known about the variable, bottom $\bot$ indicates the variable is not constant, and a constant value c indicates that the variable holds the value c. Before performing the analysis, the $<$variable, lattice value$>$ pair for each variable v is initialized to $< v, \top >$ everywhere except at the entry of the program where all variables are initialized to $\bot$.

For data-flow analysis, a control-flow graph represents a procedure with a node for each statement[1] and edges between statements indicating possible control flow. Each statement s has a set of predecessors $pred(s)$ and a set of successors $succ(s)$. The $IN(s)$ set contains the $<$variable, lattice value$>$ pairs, one for each variable, that are valid at the entry of statement s. The $OUT(s)$ set contains those pairs that are valid upon exiting statement s.

Forward data-flow analysis occurs by applying a transfer function $f_s(IN(s))$ that computes the $OUT(s)$ set for the statement s based on the $IN(s)$ set and the semantics of the statement with respect to the specific data-flow analysis. The transfer functions for reaching constants are specified in Table 1.

When two or more control paths in the CFG merge, a meet operation occurs between the lattice values for a vari-

[1]This can be generalized to basic blocks.

Table 1: Transfer functions for reaching constants.

Statement s	Transfer Function $f_s(IN(s))$
$x \leftarrow y$	$(IN(s) - \{<x, c_x>\}) \cup \{<x, c_y> \mid <y, c_y> \in IN(s)\}$
$x \leftarrow y$ op z	$(IN(s) - \{<x, c_x>\}) \cup \{<x, c_y \text{ op } c_z> \mid <y, c_y> \in IN(s) \text{ and } <z, c_z> \in IN(s)\}$
any other stmt	$IN(s) - \{<x, c_x> \mid x \text{ is defined in the statement }\}$
Added transfer functions for data-flow analysis over communication edges	
$send(x)$	$IN(s)$
$receive(y)$	$(IN(s) - \{<y, c_y>\}) \cup \{<y, \sqcap_{q \in commpred(s)} f_{comm}(IN(q))>\}$

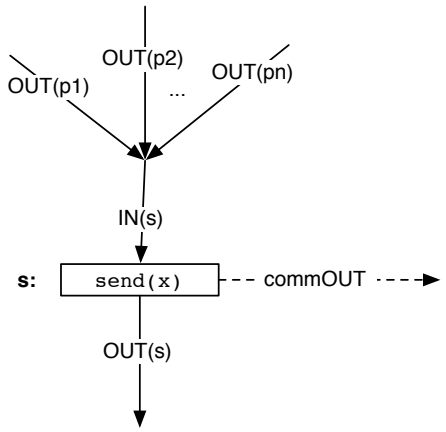

Figure 2: Control-flow edges and communication edges incident on a send node.

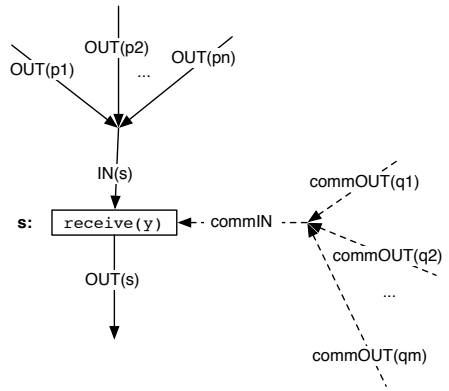

Figure 3: Control-flow edges and communication edges incident on a receive node.

able; if $<v, c_1>$ is valid along one control path and $<v, c_2>$ is valid along another path, then $<v, c_1 \sqcap c_2>$ is valid at the merge. For reaching constants, if c_1 equals c_2, then $c_1 \sqcap c_2$ is c_1. If c_1 or c_2 is equal to $\perp$, then the result of the meet operation is also $\perp$. If c_1 equals $\top$, then the result of the meet operation is c_2, and vice versa. The $IN(s)$ set for a statement is calculated by performing the pairwise meet over all the $<$ variable, lattice value $>$ pairs valid upon exiting predecessor statements, $IN(s) = \sqcap_{p \in preds(s)} f_p(OUT(p))$.

Extending reaching constants analysis to the MPI-CFG involves defining the communication transfer function f_{comm} that calculates the lattice value to propagate over communication edges based on the $IN(s)$ set for a send statement and the variable being sent. For reaching constants, the communication transfer function is

$$commOUT = f_{comm}(IN(s)) = \{c_x \mid <x, c_x> \in IN(s)\},$$

where s is the send statement `send(x)` and c_x is the lattice value assigned to the variable x in the IN data-flow set for the send statement (see Figure 2).

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 indentified by the incoming communication edges as $commpred(s)$. In Figure 1, the `receive(y)` statement only has the `send(x)` statement in its $commpreds(s)$ set. For reaching constants, the transfer function for the receive statement is defined in Table 1.

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 example we implement for our experimental results. One phase of activity analysis involves a backward data-flow analysis to determine what variables are useful when computing a particular output variable.

Useful analysis is a bitvector analysis; each variable is either useful or not. If the variable is useful at the exit of a statement, then that variable is in the $OUT(s)$ set. The same is true for the $IN(s)$ set. The meet operator is set union. In order to initialize the analysis, all output variables of interest are put into the $IN(EXIT)$ set. The transfer function shown in Table 2 has been generalized for any statement where some variable x is defined as a function of the variables y_1 through y_n. Notice that the used variables might be arrays and that in the context of activity analysis, x does not depend on any of the variables used to index such arrays.

Extending useful analysis for the MPI-CFG involves pass-

Table 2: Transfer functions for useful analysis.

Statement s	Transfer Function $f_s(IN(s))$
$x \leftarrow g(y_1, ..., y_m, y_{m+1}[*], ..., y_n[*])$	$(OUT(s) - \{x\}) \cup \{y_1, ..., y_m, y_{m+1}, y_n\}$
Added transfer functions for data-flow analysis over communication edges	
$send(x)$	$OUT(s) \cup \{x \mid (\sqcap_{r \in commsuccs(r)} f_{comm}(OUT(r))) == true\}$
$receive(y)$	$OUT(s)$

ing 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 $commIN = f_{comm}(OUT(s)) = \{true \mid y \in OUT(s)\}$.

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] and augment the ICFG with communication edges that can cross procedure boundaries to generate the MPI-ICFG. We specify the worst-case complexity for iterative data-flow analysis over the MPI-ICFG and describe the general functions that must be implemented within an MPI-ICFG data-flow analysis framework.

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; therefore, we describe the construction of the MPI-ICFG using the Fortran syntax. The relevant MPI routines and their semantics are listed in Table 3.

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 intraprocedural 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 evaluate to constants. For send and receive pairs, the tag and communicator must match if they evaluate to constants. Additional heuristics for reducing the number of communication edges are described in [32] but are not within the scope of this paper.

The MPI-ICFG also modifies how calls to MPI routines are represented. The MPI_RECEIVE, MPI_IRECEIVE, MPI_SEND, and MPI_ISEND routines are modeled with stub functions. To ensure context sensitivity for the MPI routines, the MPI-ICFG connects a unique copy of the ICFG nodes for each stub to the call and return nodes at the call-site.

4.2 Convergence of Data-Flow Analysis

To demonstrate convergence, we reduce data-flow analysis over the MPI-ICFG to normal data-flow analysis over a modified ICFG. For forward data-flow analyses, each communication edge can be converted to two control-flow edges with a node in between them. The node must include assignments that kill data-flow information for all the variables except the one being sent and must include a statement modeling a copy between the variable being sent and the variable receiving the value. The variable being received must also be set to top $\top$ before the receive statement. For the example in Figure 1, the communication edge would be replaced as shown in Figure 4.

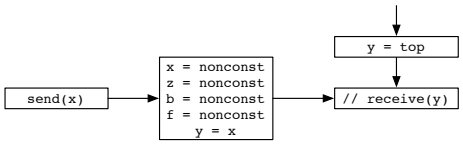

Figure 4: Converting the communication edge in Figure 1 to control-flow edges and modeling the change of address space.

The depth of the MPI-ICFG multiplied by the number of variables provides an upper bound on the number of passes required for convergence. However, the depth is difficult to calculate because the MPI-ICFG is in general irreducible as a result of the communication edges. Nonetheless, our experimental results show that the convergence is reasonable (see Table 4).

4.3 Implementation of Data-Flow Analysis

The next issue is how to implement data-flow analysis over an MPI-ICFG. Data-flow analysis frameworks for CFG's are typically implemented so that only the transfer and meet operations must be specified [10, 16, 37]. Data-flow analysis over ICFGs also requires some 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 send and receive transfer functions, and a meet operation for the communication values.

5. EXPERIMENTAL RESULTS

5.1 Methodology

We implemented the creation of the MPI-ICFG, the data-flow framework for an MPI-ICFG, and activity analysis using the OpenAnalysis toolkit [36] coupled with the Open64/SL compiler infrastructure [31] and used the data-flow analysis framework to apply activity analysis to various benchmarks. The NAS parallel benchmarks [1], labeled NASPB, were ob-

Table 3: MPI routines that we recognized as possible sources and destinations of communication edges when building our MPI-ICFG.

MPI_SEND (sendbuf, count, datatype, dest, tag, comm) A message consisting of *count* number of *datatype* values at starting address *sendbuf* is being sent to the process with the *dest* rank in the *comm* group and is marked with the message identifier *tag*.
MPI_ISEND (sendbuf, count, datatype, dest, tag, comm, request) A message consisting of *count* number of *datatype* values at starting address *sendbuf* is being sent to the process with the *dest* rank in the *comm* group and is marked with the message identifier *tag*. The status of this nonblocking send may be queried using the *request* handle.
MPI_RECV (recvbuf, count, datatype, source, tag, comm, status) A message consisting of *count* number of *datatype* values and marked with the identifier *tag* is to be received from the process with the *source* rank in the *comm* group and stored at the starting address *recvbuf*. The status of this blocking receive may be queried using the *status* handle.
MPI_IRECV (recvbuf, count, datatype, source, tag, comm, request) A message consisting of *count* number of *datatype* values and marked with the identifier *tag* is to be received from the process with the *source* rank in the *comm* group and stored at the starting address *recvbuf*. The status of this nonblocking receive may be queried using the *request* handle.
MPI_BCAST (buf, count, datatype, root, comm) A message consisting of *count* number of *datatype* values at starting address *buf* is being sent from the process with the *root* rank in the *comm* group to all processes within that group. It is copied into the starting address *buf* at each process.
MPI_REDUCE (sendbuf, recvbuf, count, datatype, op, root, comm) Each process within the *comm* group will submit *count* numbers of *datatype* values at starting address *sendbuf* to be reduced element-wise using the operation *op* into a single collection of *count* numbers of *datatype* values. This collection will be stored at the starting address *recvbuf* at the process with the *root* rank in the *comm* group.
MPI_ALLREDUCE (sendbuf, recvbuf, count, datatype, op, comm) Each process within the *comm* group will submit *count* numbers of *datatype* values at starting address *sendbuf* to be reduced element-wise using the operation *op* into a single collection of *count* numbers of *datatype* values which is then copied into the starting address *recvbuf* at each process within the *comm* group.

tained 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].

Table 4 includes definitions of our benchmarks. Each benchmark is a unique combination of source, context routine, 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 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 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 the global variable within the appropriate MPI routines 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. From the number of active bytes and the number of independent variables it is possible to estimate the storage that will be needed within the derivative code using the following formula.

$$\text{DerivBytes} = (\text{number of independents}) * (\text{ActiveBytes})$$

5.2 Effect on Activity Analysis

Each analysis determines an active symbol list and its size in bytes. Figure 5 shows the possible storage savings when activity analysis is performed over the MPI-ICFG versus the ICFG. Storage savings only occur for three of the bench-

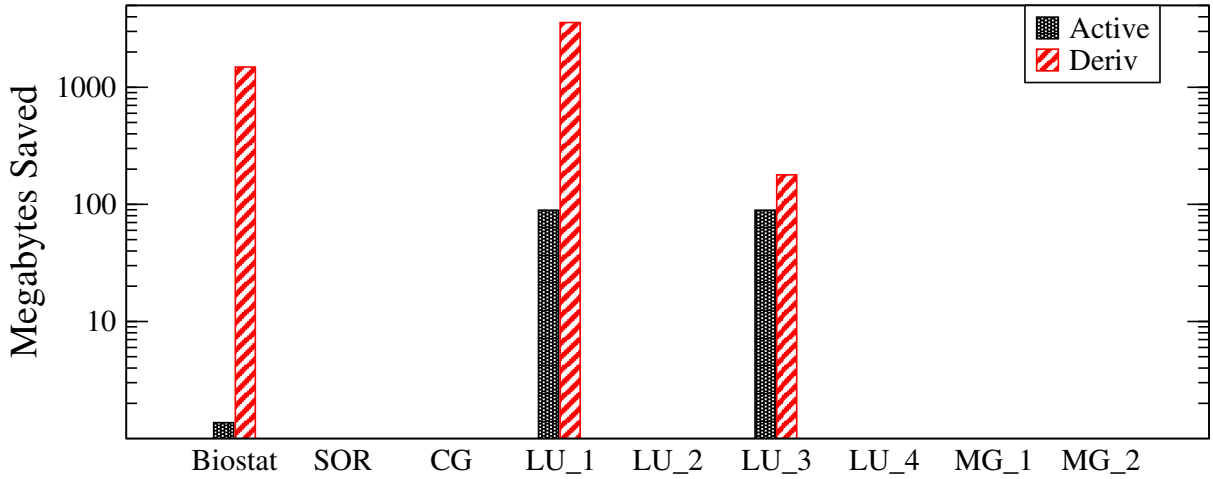

Figure 5: Activity Analysis Results: Number of megabytes saved per benchmark from MPI-ICFG over ICFG activity analysis for both the Active set and within the Derivitive code.

Table 4: Differences in number of iterations, number of active bytes and number of DerivBytes between ICFG and MPI-ICFG Activity analyses.

B'mark	Source	Context	IND	DEP	Analysis	Iter	Active Bytes	# of Indeps	Deriv Bytes	% Decrease
Biostat	Spiegelman: Biostat	lglik3	xmle	xlogl	ICFG MPI-ICFG	12 13	1432920 296	1089	1560449880 322344	99.98%
SOR	Hovland: SOR	mainsor	omega	resid	ICFG MPI-ICFG	13 17	3042176 3030096	1	3042176 3030096	0.40%
CG	NASPB: CG	conj_grad	x	z	ICFG MPI-ICFG	16 18	3600184 3600180	1	3600184 3600180	0.00%
LU-1	NASPB: LU	rhs	frct	rsd	ICFG MPI-ICFG	19 19	187194480 93636000	40	7487779200 3745440000	49.98%
LU-2	NASPB: LU	ssor	omega	rsd	ICFG MPI-ICFG	23 26	145901220 145901172	1	145901220 145901172	0.00%
LU-3	NASPB: LU	rhs	tx1,tx2	rsd	ICFG MPI-ICFG	19 19	140376496 46818016	2	280752992 93636032	66.65%
MG-1	NASPB: MG	mg3P	r	u	ICFG MPI-ICFG	17 18	16983252 16983228	1	16983252 16983228	0.00%
MG-2	NASPB: MG	mg3P	c	u	ICFG MPI-ICFG	19 25	16908572 16908548	4	67634288 67634192	0.00%

marks, but the amount of storage saved for those benchmarks is significant. Table 4 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—a 99.98% decrease in the number of active bytes with MPI-ICFG over ICFG. 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 LU-1 and LU-3 benchmarks also see dramatic reductions in active bytes and space requirements for the derivative code. The other benchmarks experience disappointing reductions. These results are probably due to precision losses within the analysis. The activity analysis over the MPI-ICFG experiences precision loss due to the may alias analysis being used, due to the context-insensitivity in the MPI-ICFG for all routines except the MPI communication routines, and due to spurious communication edges. Future work includes implementing must alias analysis within OpenAnalysis, looking at representations other than the ICFG as the basis for data-flow analysis over MPI programs so as to increase context-sensitivity, and implementing heuristics for reducing the number of communication edges adjacent to each send and receive.

5.3 Convergence

The column labeled Iter in Table 4 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

Data-flow frameworks have been developed for various types of shared-memory parallelism [8, 14, 20, 22, 23, 21, 26]. Long and Clarke [27] provide 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. It appears unlikely, however, that this approach can be extended to message-passing programs without a shared address space.

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 [7]. 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 [9]. 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. In summary, without user intervention, existing automatic differentiation tools lose precision by forcing all communicated variables to be active or by assuming communication edges between all sends and receives.

7. CONCLUSIONS

MPI programs are a significant portion of all parallel programs. We describe a data-flow analysis framework for the MPI-CFG [32] 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 nonseparable analyses such as reaching constants, activity analysis, bitwidth analysis [35], and trust analysis [15]. Our experimental results show that activity analysis performed over the MPI-ICFG has a reasonable convergence rate and that using the MPI-ICFG data-flow analysis framework provides correctness and 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 and Gail Pieper for proofreading several revisions.

9. 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 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Portland, Oregon, January 14–15, 2002*, pages 98–107, New York, 2002. ACM Press.

[5] C. H. Bischof, P. D. Hovland, and B. Norris. Implementation of automatic differentiation tools. *Higher-Order and Symbolic Computation*, 2005. Revised version of [4], to appear.

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

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

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

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

[10] M. B. Dwyer and L. A. Clarke. A flexible architecture for building data flow analyzers. In *ICSE '96: Proceedings of the 18th International Conference on Software Engineering*, pages 554–564, Washington, DC, USA, 1996. IEEE Computer Society.

[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. ACM Press, 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, London, 1994. Springer-Verlag.

[17] L. Hascoet, U. Naumann, and V. Pascual. TBR analysis in reverse mode automatic differentiation. *Future Generation Computer Systems*, 2005. To appear.

[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'98, LNCS 1470*, pages 445–455. 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 *Proc. ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42, Montreal, Canada, 1998. ACM SIGPLAN Notices 33(7).

[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 1992 Conference on Programming Language Design and Implementation*, pages 235–248. ACM Press, 1992.

[26] 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*, pages 1–12. ACM Press, 1999.

[27] 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.

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

[29] 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.

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

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

[32] 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.

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

[34] 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.

[35] M. Stephenson, J. Babb, and S. Amarasinghe. Bidwidth analysis with application to silicon compilation. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 108–120, New York, NY, USA, 2000. ACM Press.

[36] 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.

[37] S. W. K. Tjiang and J. L. Hennessy. Sharlit–a tool for building optimizers. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 82–93, New York, NY, USA, 1992. ACM Press.