# Depth Analysis of MPI Programs

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

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

Paul Hovland
Argonne National Laboratory
Argonne, IL, USA
hovland@mcs.anl.gov

## Abstract

Data-flow analyses that include some model of the data-flow between MPI sends and receives result in improved precision in the analysis results. One issue that arises with performing data-flow analyses on MPI programs is that the interprocedural control-flow graph ICFG is often irreducible due to call and return edges, and the MPI-ICFG adds further irreducibility due to communication edges. To provide an upper bound for iterative data-flow analysis complexity, one needs to have a general idea as to the depth of common flow graphs. Unfortunately, computing the depth of an irreducible graph is NP-complete.

In this paper, we compare the depth-based iteration bounds for several MPI benchmarks with the actual number of iterations required for two data-flow analyses. We are able to compute the depth despite the worst-case exponential complexity of the depth-analysis algorithm by first reducing the reducible parts of the flow graph and then explore paths in the reduced graph. Our results show that on average the reduced graphs are 80% smaller than the original graphs and have 90% fewer cycle-free paths, resulting in a 10x faster algorithm. We also observe that although the number of iterations over the flow graph is bounded by the lattice height multiplied by the graph depth, the graph depth is clearly the dominating factor and provides a close approximation to the complexity of iterative data-flow analysis over MPI programs.

***Keywords*** MPI, graph depth, MPI-ICFG, data-flow analysis

## 1. Introduction

Many parallel programs are written using MPI (Message Passing Interface) [23]. With the push for exascale computing, the expectation is that MPI will still be the main parallelization mechanism between nodes on a machine [17]. Therefore, techniques for statically and dynamically analyzing MPI programs will become ever more crucial.

Researchers have developed techniques for performing data-flow analysis on MPI programs [7, 26, 28]. Data-flow analysis is the basis for bug-finding analyses [18], analyses that find security issues such as buffer overflow [14], and performance optimizations such as those based on activity analysis in the context of automatic differentiation [6]. Our previous research involving data-flow analysis for MPI programs has shown that extending the data-flow analysis so that information flows over communication edges between MPI sends, receives, and reduction primitives enables more precise analysis.

A complication is that the modified interprocedural control-flow graphs (dubbed MPI-ICFGs) used in MPI-aware data-flow analysis are frequently irreducible. Many techniques for improving the efficiency of data-flow analysis, such as interval analysis [4, 8] and path compression [29], depend on the control-flow graph being reducible. This is not unreasonable as most programs are reducible [1, 3]. It has also been shown that techniques such as node splitting can make a control-flow graph reducible [1, 3, 10]. Unfortunately, these techniques do not generally apply to MPI-ICFGs.

This paper makes the following contributions:

- Description of why MPI-ICFGs are frequently irreducible.
- Explanation of why techniques such as node splitting are not applicable.
- An algorithm for computing the depth of a graph that leverages polynomial time algorithms for computing the reducible parts of the graph and therefore enables analyzing larger graphs more efficiently even though it is still worst-case exponential.
- Depth results using that algorithm for a collection of MPI benchmark programs, including several NPB benchmarks and small applications, along with a comparison of the depth-based upper bound with the actual data-flow analysis complexity.

In this paper we explore the algorithmic complexity of performing data-flow analysis on MPI programs. Section 2 reviews the MPI-ICFG and a data-flow analysis framework for the MPI-ICFG. Section 3 reviews the concept of depth and its relationship with the algorithmic complexity of iterative data-flow analysis. Section 4 describes why MPI-ICFGs are irreducible and presents an algorithm for computing the depth of an irreducible graph. Section 5 shows that although the depth analysis algorithm is worst-case exponential in the size of the graph, reducing the graph as much as possible enables us to present depth analysis results on a number of benchmarks. Section 6 concludes.

## 2. Data-Flow Analysis for MPI Programs

Data-flow analysis for MPI programs must capture the possible data flow between sending and receiving MPI calls, as well as the traditional data flow along control-flow paths. Most MPI programs do not have all of their MPI calls in one procedure; therefore, interprocedural data-flow analysis is necessary. This section reviews the MPI-ICFG, which is used to represent MPI programs. This section also reviews a data-flow analysis framework for the MPI-ICFG [28].



*AMP '10* 6 June 2010, Toronto, Ontario, Canada

### 2.1 Intraprocedural Data-Flow Analysis using the CFG

First we review data-flow analysis for a single procedure, which is also called intraprocedural data-flow analysis. Intraprocedural data-flow analysis is formulated over a control-flow graph (CFG), 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 $n_2 \in succ(n_1)$.

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 such 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. Additionally 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. The transfer function relies on the semantics of the statement $s$ within $n$ and the data-flow analysis being performed.

Reaching constants is a canonical example of a nonseparable, forward data-flow analysis. A *nonseparable analysis* is one where the data-flow value associated with one entity like a variable is dependent on data-flow values associated with other entities. We are interested in nonseparable analyses, because their precision can improve when data-flow information is propagated between an MPI send and receive.

In reaching constants, each variable $v$ is paired with a value $c_v$. The possible constant 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 initialized 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 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 value with the left-hand-side variable in the $OUT$ set for the statement. In Figure 1, the variable `x` will be paired with the constant value `1` in the $OUT$ set for statement `x = x + 1`.

### 2.2 Interprocedural Data-Flow Analysis using the ICFG

For interprocedural data-flow analysis, we generate an interprocedural control-flow graph (ICFG) [21]. The ICFG includes control-flow edges between procedure calls and the control-flow graphs for the called procedures. The ICFG can be constructed as follows: (1) construct a control-flow graph for each procedure, (2) split each control-flow graph node containing a procedure call into a *call* node and a *return* node, (3) add an edge from the call node to the control-flow graph entry node of the called procedure, and (4) add an edge from the control-flow graph exit node of the called procedure to the return node. Data-flow analysis frameworks for control-flow graphs are typically implemented so that only the transfer and meet operations must be specified [11, 15, 29]. Data-flow analysis over ICFGs also requires a specification of how information is mapped from the caller to the callee, and vice versa.

[1] This can be generalized to basic blocks.

```
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)
```

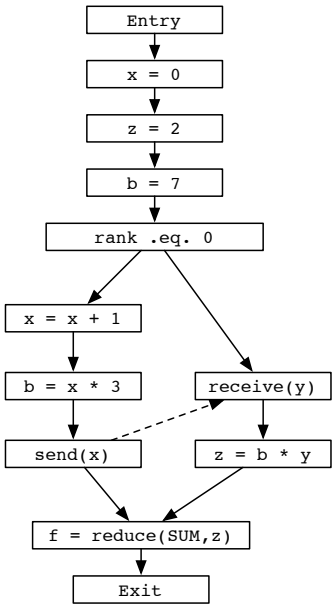


**Figure 1.** A small SPMD program and the corresponding simplified MPI-ICFG.

### 2.3 Interprocedural Data-Flow Analysis using the MPI-ICFG

For data-flow analysis of MPI programs, we use an MPI-ICFG, which augments an ICFG with communication edges between possible send and receive pairs, among all calls to broadcast, and among all calls to reduce. A *communication* edge connects the call node of the sending call to the return node of the receiving call. 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, we eliminate communication edges where the root parameters statically evaluate to different constants. For send and receive pairs, we eliminate communication edges where the tag or communicator do not match.

The communication edges in an MPI-ICFG are not control-flow edges. Only the data specified in the sending call flows along a communication edge. Accordingly, a data-flow analysis on MPI

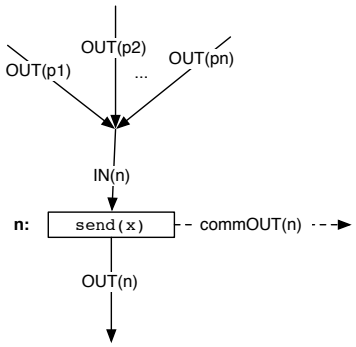


**Figure 2.** Control-flow edges and communication edges incident on a send node.

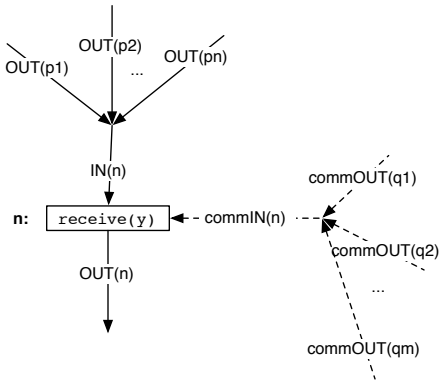


**Figure 3.** Control-flow edges and communication edges incident on a receive node.

programs should allow only a value for the variable being sent/received to be transferred across a communication edge.

An MPI-ICFG is specified as $ICFG_{MPI} = \{V, E, C\}$, where $V$ is the set of nodes in the graph, $E$ is the set of control-flow edges and call and return edges, and $C$ is the set of communication edges in the graph. Extending any forward, nonseparable data-flow analysis for operation over the MPI-ICFG involves defining the communication transfer function $f_{comm}$ that calculates the 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 value assigned to the variable $x$ in the $IN(n)$ data-flow set for the send node. In Figure 1, the value `1` will be propagated over the communication edge between `send(x)` and `receive(y)`. Note that for simplicity, the example in Figure 1 does not include any call and return edges. Examples with call and return edges are provided in Section 4.2.

The transfer function for the receive statement must be defined so that it uses the value propagated over all incoming communication edges as input. Assume that an MPI-ICFG 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 send nodes identified by the incoming communication edges as $commpred(n)$. In Figure 1, the `receive(y)` node has only the `send(x)` node in its $commpreds(n)$ set. For reaching constants, the transfer function for the receive node can be defined as

$$\begin{aligned} OUT(n) \quad = \quad & (IN(n) - \{\langle y, c_y \rangle\}) \\ & \cup \{\langle y, \sqcap_{q \in commpred(n)} f_{comm}(IN(q)) \rangle\}, \end{aligned}$$

where $\sqcap$ is the symbol for the meet operation and the variable $y$ is being set to the meet of all the values that result from applying the communication transfer function to $IN(q)$ set for all source nodes $q$ of the incoming communication edges. In Figure 1, the $OUT$ set for the node containing the `receive(y)` statement will include the following set of variables paired with values: $\{\langle x, 0 \rangle, \langle z, 2 \rangle, \langle b, 7 \rangle, \langle f, \bot \rangle, \langle y, 1 \rangle\}$.

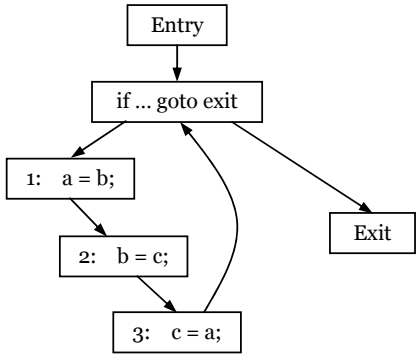


**Figure 4.** Control-flow graph with three statements.

## 3. Data-Flow Analysis, Complexity, and Depth

The possible values being propagated during data-flow analysis can be organized into a lattice, or partially ordered set. The algorithmic complexity of iterative data-flow analysis is determined based on lattice height and flow graph depth [2]. We use a nonseparable, forward data-flow analysis called vary analysis [16]. The example program in Figure 4 helps illustrate these concepts.

### 3.1 Vary Analysis

Vary analysis is a domain-specific forward data-flow analysis used in some implementations of automatic differentiation [13], which is a type of program transformation that transforms a subprogram that computes a mathematical function into a subprogram that computes the (partial) derivatives of that function. Vary analysis seeks to identify the set of variables that depend on the subset of input variables designated as independent variables for purposes of differentiation. This enables the automatic differentiation tool to avoid allocating space and performing intermediate computations for variables that can be statically proven to have zero derivatives. A more complete description of vary analysis can be found in [16].

For the example in Figure 4, assume that we apply vary analysis with the set of inputs designated as independent containing only the variable `b`. Table 1 shows the IN/OUT sets for this example. The assignment of `a` depends on the initial value of the variable `b`, and therefore `a` varies after statement 1. Since the variable `b` is reassigned in statement 2, the variable `b` will not vary after statements 2 and 3.

**Table 1.** Results of vary analysis with b as independent over the CFG in Figure 4.

| | | | After Iteration | | |
|---|---|---|---|---|---|
| Node | Set | Orig. | #1 | #2 | #3 |
| Entry | IN | {b} | {b} | {b} | {b} |
| | OUT | {b} | {b} | {b} | {b} |
| If... | IN | {} | {b} | {ab} | {ab} |
| | OUT | {} | {b} | {ab} | {ab} |
| 1:a=b | IN | {} | {b} | {ab} | {ab} |
| | OUT | {} | {ab} | {ab} | {ab} |
| 2:b=c | IN | {} | {ab} | {ab} | {ab} |
| | OUT | {} | {a} | {a} | {a} |
| 3:c=d | IN | {} | {a} | {a} | {a} |
| | OUT | {} | {a} | {a} | {a} |
| Exit | IN | {} | {b} | {ab} | {ab} |
| | OUT | {} | {b} | {ab} | {ab} |

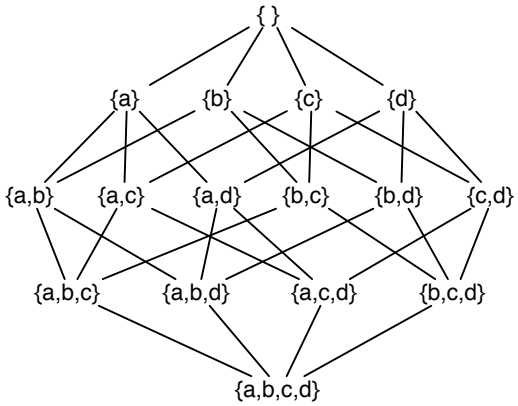


**Figure 5.** Lattice for vary analysis and program in Figure 4.

**Table 2.** Results of vary analysis with d as independent over the CFG in Figure 4.

| | | | After Iteration | | | | |
|---|---|---|---|---|---|---|---|
| Node | Set | Orig. | #1 | #2 | #3 | #4 | #5 |
| Entry | IN | {d} | {d} | {d} | {d} | {d} | {d} |
| | OUT | {d} | {d} | {d} | {d} | {d} | {d} |
| If... | IN | {} | {d} | {cd} | {bcd} | {abcd} | {abcd} |
| | OUT | {} | {d} | {cd} | {bcd} | {abcd} | {abcd} |
| 1:a=b | IN | {} | {d} | {cd} | {bcd} | {abcd} | {abcd} |
| | OUT | {} | {d} | {cd} | {abcd} | {abcd} | {abcd} |
| 2:b=c | IN | {} | {d} | {cd} | {abcd} | {abcd} | {abcd} |
| | OUT | {} | {d} | {bcd} | {abcd} | {abcd} | {abcd} |
| 3:c=d | IN | {} | {d} | {bcd} | {abcd} | {abcd} | {abcd} |
| | OUT | {} | {cd} | {bcd} | {abcd} | {abcd} | {abcd} |
| Exit | IN | {} | {d} | {cd} | {bcd} | {abcd} | {abcd} |
| | OUT | {} | {d} | {cd} | {bcd} | {abcd} | {abcd} |

### 3.2 Complexity of Iterative Data-Flow Analysis

We can use iterative data-flow analysis to solve vary. The data-flow analysis equations for the vary analysis are as follows:

$$IN[n] = \bigcup_{p \in pred(n)} OUT[p]$$

$$OUT[n] = \{x \mid x \in defs[n] \text{ and } (IN[n] \cap uses[n]) \neq \emptyset\}$$

In the above data-flow analysis equations, the union $\bigcup$ over all predecessor $OUT$ sets is the meet operation for the vary analysis. The expression that computes the $OUT$ set based on the $IN$ set is the transfer function.

An iterative data-flow analysis algorithm visits each node in the control-flow graph, or in the case of MPI programs the MPI-ICFG, and computes the $IN$ and $OUT$ sets for the nodes until all $IN$ and $OUT$ sets converge.

The upper-bound on the number of visits per node is $O(hd)$, where $h$ is the lattice height and $d$ is the depth of the graph.

### 3.3 Lattice Height

The *lattice* is a partially-ordered set of data-flow values. For vary analysis, the domain of possible data-flow values is the powerset of the set of all variables in the program, $2^V$. For the example program in Figure 4, the lattice is shown in Figure 5. The height of a powerset lattice is the size of the original set (e.g., $|V|$).

The height of the lattice affects the complexity because all of the $IN$ and $OUT$ sets excluding possibly those for the entry and exit nodes will start at the top value in the lattice and will stop when reaching the bottom value.

Figure 4 provides an example where the full height of the lattice is traversed. Assume that instead of variable b being in the independent variables that the variable d is in the initial $IN$ set for the entry node. Table 2 shows the contents of all IN/OUT sets for this modified example. On the first iteration of the data-flow analysis, the variable c will join the vary set in the $OUT$ set for statement 3. The second iteration over the nodes will propagate the set including variable d and c into the top of the loop and will result in b becoming vary after statement 2. The third iteration will propagate the set {bcd} into the top of the loop and will result in a becoming vary after statement 1. A fourth iteration will propagate the set {abcd} into the loop as well as the exit node. The final iteration will check convergence.

### 3.4 Depth of a Graph

The *depth* of a graph is the maximal number of retreating edges on any cycle-free path [2]. A *retreating edge* is an edge that is *not* a tree edge within a depth-first spanning tree of the graph and that targets a node that is an ancestor in the DFST. A *cycle-free path* is defined as a list of nodes in a graph $(n_0, n_1, ..., n_k)$ such that for each $n_i$ and $n_{i+1}$ in the list there is an edge $(n_i, n_{i+1})$ in the graph. Also, each of the $n_j$ must be unique.

The depth of the graph also affects the complexity of an iterative data-flow analysis algorithm in that each node could be visited lattice height multiplied by depth times, $O(hd)$. In Figure 4, the depth of the graph is 1. The path $(3, 1, 2)$ is a cycle-free path containing the one retreating edge $3 \rightarrow 1$.

### 3.5 Reducible Graphs

In a structured program (i.e., one built with structured control-flow constructs), the depth of the control-flow graph is equivalent to the maximum loop depth. It is also possible to determine the depth of a

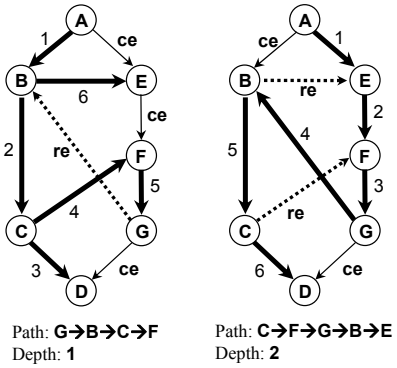


**Figure 6.** Different DFSTs on the same graph may produce different depths. The number beside the tree edges indicates order of inclusion into the DFST. The provided paths indicate the same graph is able to have more than one depth based on the DFST.

structured program's control-flow graph in $O(N+E)$ time because such a control-flow graph is reducible [12].

A *reducible* graph is one where all retreating edges in any depth-first spanning tree (DFST) are edges between a node and one of the node's dominators. A *dominator* for node X is a node Y that must be in any path from the entry of the flow graph to the node X. When a graph is reducible, it is possible to apply T1 and T2 transformations [30] so as to reduce the whole graph to one node. A T1 transformation removes self loops, and a T2 transformation collapses a child node into a parent node if the child only has edges from one parent node.

T1 and T2 transformations are useful because they enable the generation of data-flow summaries for a whole procedure and thus significantly reduce the cost of iterative data-flow analysis [4, 24]. T1 and T2 transformations can also be used to compute the depth of a reducible flow graph.

## 4. The Depth of Irreducible Graphs

The problem with MPI-ICFGs is that the communication edges often break the structure of the program and cause the MPI-ICFG to become irreducible. Using a reduction from the directed Hamiltonian cycle problem, Fong and Ullman [12] show that determining the depth for irreducible graphs is NP-complete. In this section, we describe some characteristics of irreducible graphs, explain in detail why MPI-ICFGs are irreducible, and then present a worst-case exponential algorithm for computing the depth of arbitrary graphs.

### 4.1 Irreducibility

All directed graphs can be categorized as being either reducible or irreducible. Therefore, any graph that is not reducible is irreducible. Irreducibility of a graphs means that computing the graph's depth is an NP-complete problem and therefore any such computation will have exponential worst-case complexity. Another difficulty that irreducible graphs introduce is that they can have a different set of retreating edges based on the depth-first spanning tree (DFST). Different sets of retreating edges can result in different depths for the same graph.

Figure 6 shows two DFSTs over the same graph that result in different depths. The bold edges are tree edges, and the number beside each tree edge indicates the order in which it was added to the DFST. The dashed edges are retreating edges. The rest are cross edges. The DFST on the left starts with the edge (A → B), and the DFST on the right starts with the edge (A → E). After the starting edge is selected, all other edges are visited in a depth-first manner;

```
subroutine posPow(a,n,ans)    (0)
  ans = 0                     (1)
  call foo(ans,a)             (2)
  do k=2, n, 1                (3)
    call foo(next,ans)        (4)
    ans = next                (5)
  end do                      (6)
end subroutine                (7)
subroutine foo(y,x)           (8)
  y = 2 * x                   (9)
end subroutine               (10)
```

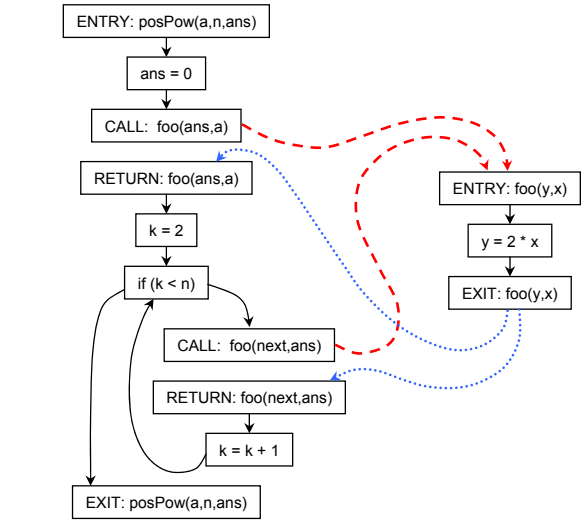


**Figure 7.** Irreducibility induced by ICFG.

outgoing edges are visited in alphabetical order. The DFST on the left contains only one retreating edge, so its depth can be at most 1. The DFST on the right has two retreating edges, so its depth can be at most 2. Indeed, the cycle-free paths displayed in Figure 6 for each DFST confirm those depths.

To work around this issue in our computation of the depth of MPI-ICFGs, we ensure that the traversal order being used in the depth analysis is based on the same DFST as that used in the data-flow analysis.

### 4.2 Irreducibility of MPI-ICFGs

Irreducibility may be introduced into the MPI-ICFG graph in two ways. First, the MPI-ICFG is based on an ICFG, where we have one interconnected graph for the entire application program. When there are multiple calls to the same subroutine, there will be multiple edges from the various call nodes to the entry of the called subroutine. An example of how this introduces irreducibility is seen in Figure 7. This example contains two calls to the same subroutine. One call is within a loop; the other call is outside the loop. Figure 7 also shows the ICFG for this example where each call node now has an outgoing edge to the entry node of the called subroutine and each return node has an in-coming edge from the exit node of the called subroutine. The edge from `foo`'s exit node to the in-loop return node constitutes an edge entering the middle of the loop from outside the loop, since there exists a path from the first call, through `foo`, into the loop. This situation with two entries into a loop is irreducible.

A second way that irreducibility may be introduced into the MPI-ICFG is through the use of communication edges. Communication edges connect the call nodes of communication sending calls to the return nodes of compatible communication receiving calls. An example of how this introduces irreducibility is seen in

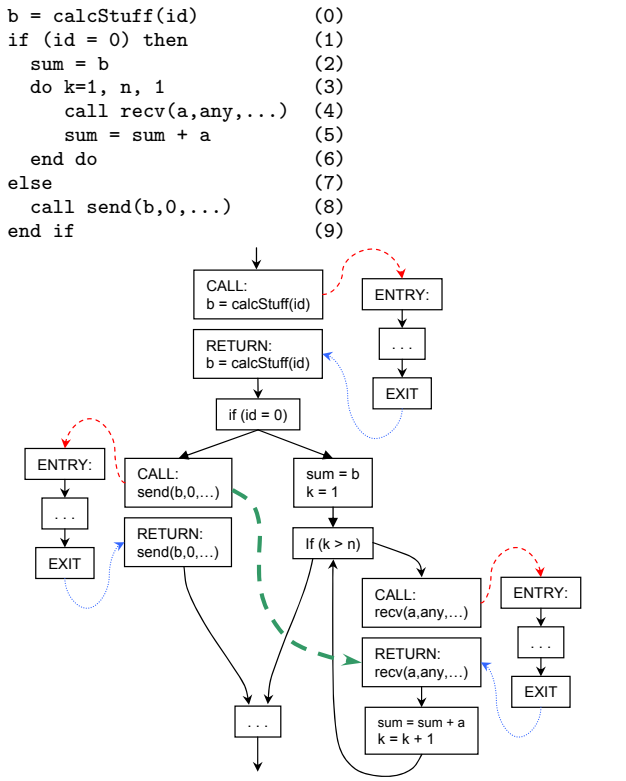


**Figure 8.** Irreducibility induced by MPI-ICFG.

Figure 8. One branch of the if-statement contains a call to send(); the other branch contains a loop with a call to recv(). A communication from the send call node to the receive return node represents an edge entering the middle of a loop, which renders the graph irreducible.

### 4.3 Why Not Use Node Splitting?

*Node splitting* [1, 3, 10] is an analysis technique for making an irreducible graph reducible. Node splitting causes the duplication of one of the regions involved in an irreducible loop within the region graph. If the irreducible loop is due to a goto statement within one procedure, then the amount of duplication might not be that significant. Since irreducibility is introduced in MPI-ICFGs due to a communication edge leaving one procedure and entering a loop in another procedure, we hypothesize that the amount of duplication that would be needed to perform node splitting might be on the order of full procedures and therefore become possibly exponential in the size of the original flow graph [9].

### 4.4 Algorithm for Computing Depth of Irreducible Graphs

Fong and Ullman [12] present an algorithm that computes the depth of a reducible graph in $O(N + E)$ time by using T1 and T2 reductions. A *T1 reduction* removes a self-edge from a node, and a *T2 reduction* collapses a node into a parent node when the node only has one parent node. The result of their algorithm is the largest number of retreating edges on any path ending at each node. The depth is then the maximum of all those values.

We use their algorithm as a starting point on an irreducible graph so as to reduce the structured parts of the graph as much as possible. We present our algorithm in Figures 9, 10, and 11.

Figure 9 shows the variables that are used throughout the depth computation. $Y_n$ is the maximum depth of any reducible path that

```
// Y_n is the max depth for any reducible path
//     that ends at node n.

// X_{n,m} is the the max depth for any reducible path p
//         that ends at node n and p does not intersect the
//         path from the region's root to the node m.

// R is the set of retreating edges in the
//     reduced graph G'.

// C_n is the color for node n.

// PATH is a list of nodes for the current
//         cycle-free path under investigation.
```

**Figure 9.** Variables used in the T1T2ComputeDepth, computeReducibleDepth, and maxDepthPath algorithms.

```
Algorithm T1T2ComputeDepth( G(V,E), entry )
    for all v ∈ V initialize Y_v = 0

    for all v ∈ V and w ∈ V initialize X_{v,w} = -∞

    initialize R = ∅

    // Reduce the graph while computing reducible depths
    // and determining a set of retreating edges.
    // Computes all Y_n, X_{n,m}, and R.
    G' = computeReducibleDepth(G, entry)

    // S is the set of all retreating edges in G' and
    // all outgoing edges for nodes that are the end of
    // a path with some reducible depth.
    S = R ∪ {(v,w) | Y_v > 0}

    depth = max_{e∈S} maxDepthPath(G', e)

    return depth
```

**Figure 10.** The T1T2 algorithm for computing the depth of an irreducible graph.

ends at node $n$. Any irreducible edges exiting $n$ could then begin a path containing irreducible retreating edges. A path through the root of a region to an exit node of a region will have no retreating edges.

Figure 10 shows pseudocode for the T1T2Algorithm for computing the depth of an irreducible graph. The main idea is to use T1 and T2 reductions to compute the reducible depth as first described by Fong and Ullman [12] and then use the more direct approach of investigating all cycle-free paths starting from retreating edges or from the end of reducible paths with depth to determine the full depth of the graph. Searching all possible cycle-free paths even from a subset of starting edges can result in a worst-case algorithm complexity that is exponential in the size of the graph.

Figure 11 details the algorithm for computing the depth of the reducible portions of the flow graph. The results of the depth-first traversal in the computeReducibleDepth algorithm are (1) the $Y_n$ values will be set for every node, thus indicating the greatest number of retreating edges on any reducible path ending at node n, and (2) the variable $R$ will contain a set of retreating edges that remain in the reduced graph.

Figure 12 shows the maxDepthPath algorithm, which will find a maximum-depth path out of all of the paths that start at a given edge and return that depth. We compute the maximum depth by finding the largest depth on an irreducible path starting from an edge and adding in the reducible depth for the source node in the starting edge.

```
Algorithm computeReducibleDepth(G(V,E), v)
    C_v = grey

    // Depth-first traversal and find retreating edges.
    for each outgoing edge (v,w) from v
        if C_w ≠ grey then // tree edge
            G' = computeReducibleDepth(G,w)

        else if C_w = grey then // retreating edge
            R = R ∪ (v,w)

    // Apply T1 and T2 reductions as much as possible.
    while reductions are still possible

        // Check for T1 applicability.
        if there is a self loop (s,t) for region(v) then

            // First update the maximal reducible depth
            // ending at any node in the region.
            for each node n in this region

                // The max depth path ending at n either
                // stays the same or is the path that ends
                // at node s, goes through the self loop,
                // and traverses tree edges to end at n.
                Y_n = max(Y_n, 1 + X_{s,n})

            Remove the self loop from R and E'.

        // Check for T2 applicability.
        if S = region(v) only has a single
        predecessor region P then

            // All pairs of nodes in regions P and S.
            for each n ∈ P and for each m ∈ S
              // All nodes at frontier of P that have
              // outgoing edges into S.
              for each node p ∈ P such that (p, root(S)) ∈ E'

                // The largest depth path to n will be in
                // region P and any path to m in region
                // S must go through one of the p nodes.
                X_{n,m} = max ( X_{n,m}, X_{n,p} )

                // A node m in region S might have a
                // larger depth path that ends at a p
                // node and then traverses S tree edges.
                X_{m,n} = max ( Y_m, X_{p,n} )

            for all the nodes in S
              for each node p ∈ P with an edge to S
                Y_n = max ( Y_n, Y_p )

            Merge regions P and S and
            remove edges from P to S.

    // Return the reduced graph where each region is a
    // node and edges go between regions.
    return G'(V', E')
```

**Figure 11.** The computeReducibleDepth algorithm uses T1 and T2 reductions to reduce as much of the graph as possible and to compute the maximum reducible depth that ends at each node. This is an implementation of the algorithm presented by Fong and Ullman [12].

```
Algorithm maxDepthPath(G'(V',E'), (v,w), path)
    // For first edge, need to initialize the path.
    if PATH is empty then
        Put v at the end of the path

        // Initialize maxDepth based on the max depth
        // reducible path that ends at v.
        maxDepth = Y_v

    // Place the target of the edge on the current path.
    Put w at end of PATH.

    // Find the max depth for any path that starts at w.
    maxDepth = 0
    for each edge (w,t)
        if t is not in the path
            depth = maxDepthPath(G', (w,t))
            if (depth > maxDepth) maxDepth = depth

    if (v,w) is a retreating edge in R
        maxDepth = maxDepth + 1

    // Modify current path being investigated.
    Take w off the path.

    return maxDepth
```

**Figure 12.** The maxDepthPath algorithm adds the maximum depth for any reducible path that ends at the first node in the path to the maximum depth of any path in the irreducible graph.

## 5. Depth Analysis Results

In this section, we compute the depth of a number of MPI benchmarks. We use the depth to calculate the worst-case number of iterations and compare those figures to actual numbers of iterations for analyses on both the ICFG and the MPI-ICFG. We present our methodology and discuss our results below.

### 5.1 Methodology

We implemented two algorithms to find the depth of a graph. The Direct algorithm explores all cycle-free paths starting from a retreating edge in the depth-first-spanning tree (DFST) of the data-flow graph. For reducible graphs, all DFSTs of the graph have the same depth. However, this situation is not true for irreducible graphs, as illustrated in Section 4.1. Thus, to find the true maximum depth of an irreducible graph, one would need to investigate all DFSTs of the graph. We limit our depth results to the DFSTs used in the data analysis runs, since those directly affect the actual number of iterations. Specifically, we ensure that the depth computed by our depth analysis uses the same DFST as that used by the reverse-post ordering in the iterative data-flow analysis.

The T1T2Algorithm described in Section 4.4 uses T1 and T2 reductions to reduce the number of regions and inter-region edges, memoizing the reducible depth in each region, before traversing paths in the reduced graph. This algorithm also starts from the same DFST as that used in the actual data analysis runs.

We implemented the construction of the MPI-ICFG, the data-flow framework for an MPI-ICFG, and a handful of analyses using the OpenAnalysis toolkit [27] coupled with the Open64/SL compiler infrastructure [25]. We used the data-flow analysis framework to apply two data-flow analyses (reaching constants and vary) to various benchmarks. The NAS Parallel Benchmarks [5], 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 [19]; and the benchmark labeled Biostat is a parallelized version of a biostatistical analysis function provided by D. Spiegelman [20]. Sweep3d [22] is a benchmark code derived from a real application,

**Table 3.** Benchmark Description: Number of communication calls, nodes, edges and communication edges.

| Benchmark | Source | # of Sends | # of Recvs | # of Nodes | # of Edges | Comm Edges |
|---|---|---|---|---|---|---|
| Biostat | Spiegelman: Biostat (lglik3) | 5 | 3 | 75 | 90 | 5 |
| CG | NASPB: CG (conj_grad) | 8 | 8 | 154 | 200 | 64 |
| LU_1 | NASPB: LU (rhs) | 4 | 4 | 309 | 402 | 4 |
| LU_2 | NASPB: LU (ssor) | 8 | 8 | 589 | 760 | 16 |
| MG_2 | NASPB: MG (psinv) | 6 | 1 | 372 | 479 | 6 |
| SOR | Hovland: SOR (mainsor) | 17 | 15 | 283 | 362 | 87 |
| Sw_1 | ASCI: Sweep3d (sweep) | 1 | 1 | 359 | 467 | 1 |

3D Discrete Ordinates Neutron Transport, which solves a neutron transport problem. Table 3 summarizes our benchmarks. Column 2 indicates the source of the benchmark along with the procedure of interest at the top of the call graph. Columns 3 and 4 characterize the type and amount of MPI-communication calls in the benchmark that contributed to the communication edges listed in the last column. Columns 5 and 6 list the number of nodes and non-communication edges in the MPI-ICFG and ICFG graphs for each benchmark.

### 5.2 Results

In this section, we first present our results on the performance of both the T1T2Algorithm and the Direct algorithm, with special emphasis on the achieved potential of the T1T2Algorithm over the Direct algorithm. We then discuss our depth results and compare the depth-defined upper-bound number of iterations to actual numbers of data-flow analysis iterations.

#### 5.2.1 Algorithm Evaluation

We calculated the depth of both the ICFG and MPI-ICFG for each benchmark using the Direct algorithm and then the T1T2Algorithm, for a total of four calculations per benchmark. In Figure 13, we show the number of cycle-free paths investigated by either algorithm versus the size of the graph (nodes + edges or regions + edges, as appropriate). The size of the graph is shown along the horizontal axis, and the number of paths is shown along the log-scale vertical axis. The upper-bound complexity of the Direct algorithm is exponential – on the order of $E^{N+1}$. The time to traverse all paths prohibits the direct calculation of the depth on large graphs. Our benchmarks are medium-sized, but in some cases the calculations still took a while. The two marks in the far upper-right in Figure 13 represent the two Direct calculations on LU_2, our largest benchmark. Each calculation took more than 14 days to complete on a 2.6 GHz Intel Core 2 duo processor with 2GB 1067 MHz DDR3 memory running the Mac OS X (v. 10.5.8) operating system.

The T1T2Algorithm uses T1 and T2 reductions to reduce the number of regions and inter-region edges before traversing paths to calculate the graph depth. Figure 14 shows the reduction in the graph size for each benchmark for both the ICFG and MPI-ICFG graphs. On average, the T1T2Algorithm was able to reduce the graph size by 80% for MPI-ICFGs and by 86% for ICFGs, before traversing inter-region edges. This algorithm is still worst-case exponential; however, it ultimately traverses fewer paths than does the Direct algorithm, enabling depth calculation on larger graphs.

As a last measure of the T1T2Algorithm, we compared the number of cycle-free paths traversed by both algorithms. In Figure 15, we display the number of paths traversed by the T1T2Algorithm as a percentage of the number of paths traversed by the Direct algorithm on the benchmark MPI-ICFGs. For MG_2, the T1T2Algorithm traversed as few as 3.5% of the number of paths traversed by the Direct algorithm. On average, the T1T2Algorithm traverses only 10% as many paths on the MPI-ICFGs and only 7% as many on the ICFGs as does the Direct algorithm.

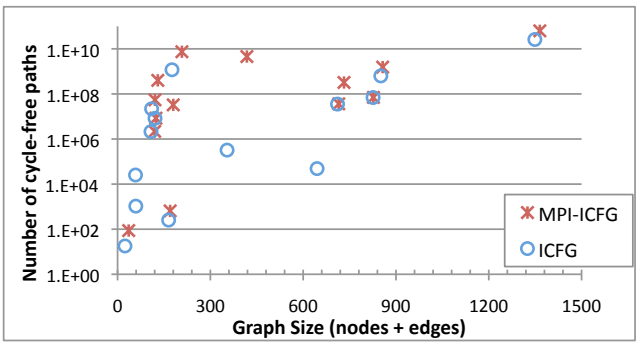


**Figure 13.** Graph size vs. number of paths traversed – includes output from both algorithms over ICFG and MPI-ICFG graphs

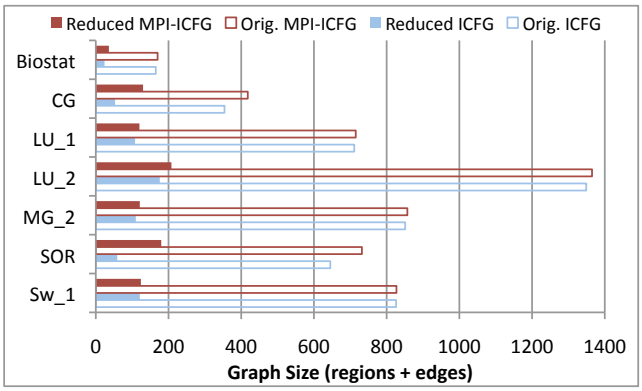


**Figure 14.** Reduction in graph size (regions + edges) by T1T2 algorithm.

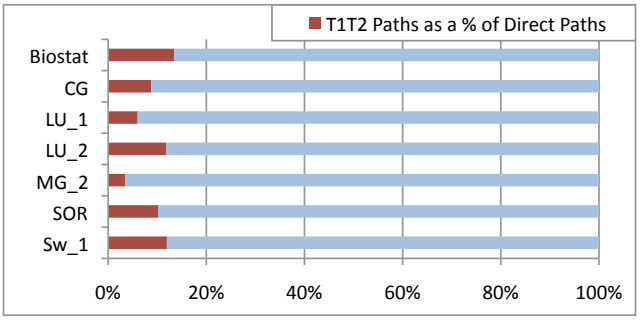


**Figure 15.** Number of paths traversed by the T1T2Algorithm displayed as a percentage of those traversed by the Direct algorithm on the benchmark MPI-ICFGs.

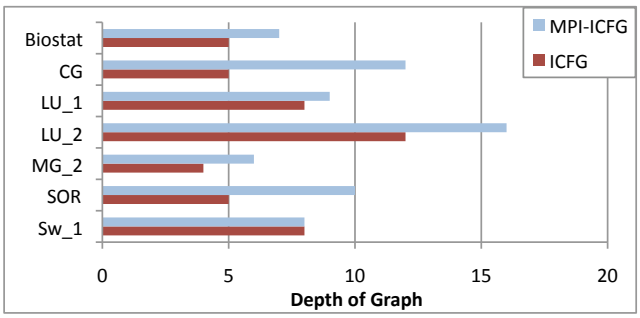


**Figure 16.** Graph depth of the ICFG and additional depth due to communication edges in the MPI-ICFG.

#### 5.2.2 Depth and Data-Flow Analysis Results

We calculated the depth of both the MPI-ICFG and ICFG for each benchmark. The results are shown in Figure 16. As expected, the MPI-ICFG is at least as deep as the ICFG for each benchmark. The depths ranged from a low of 4 in the ICFG of MG_2 to a high of 16 in the MPI-ICFG of LU_2. For Sw_1, with only one communication edge, the depth is the same for both graphs. CG and SOR are the benchmarks with the highest number of communication edges: 64 and 87, respectively. For both of these, the depth of the MPI-ICFG is at least twice that of the ICFG.

Let's take a closer look at the LU benchmarks. We first present a detailed discussion of the source of the depth of LU_1 and then a brief description of LU_2 components that contribute to its depth. LU_1 is a subset of LU_2 and all of the MPI-calls in LU_1 are within the `exchange_3` subroutine. `exchange_3` performs either north-south communication or east-west communication depending upon a parameter value. For north-south communication, there are six consecutive if-statements that guard MPI calls in the following order: receive, send, wait, receive, send, wait. Four of the six if-statements contain double-nested loops. The east-west communication is similar. Each send and receive has a unique tag indicating direction (e.g. from_north), which results in exactly two communication edges in the north-south communication and two in the east-west. The code is written such that an if-statement containing a receive immediately precedes the if-statement containing the matching send.

The depth of `exchange_3` itself is 7. However, individually, we observe that without communication edges, the depth is 5 and that when using duplicate unique send/receive/wait routines, the depth is 4. Collectively, we observe that without communication edges and with duplicated routines, the depth is 2 (from the double-nested loops). Interestingly, in the presence of communication edges and multiple calls to the same routines, without any of the six if-statements per communication pattern the depth is only 3. The if-statements that guard each MPI call within `exchange_3` enable paths that skirt the tree-edge calls of the DFST in favor of the retreating-edge calls and calls incident upon communication edges.

The `rhs` subroutine is the root of LU_1's call graph, containing two calls to `exchange_3` and some quad-nested loops. The depth of LU_1 is 9. There are many different cycle-free paths in LU_1 with 9 retreating edges and we describe two types below. One type starts with a cycle-free path of depth 7 from `exchange_3`, branches out to the return node of the first call to `exchange_3` within `rhs`, and continues through to the second call. It re-enters `exchange_3` and reaches a communication edge not in the prefix path. The second call to `exchange_3` and the communication edge bring the number of retreating edges to 9.

The second path type starts with 4 retreating edges from a quad-loop within `rhs`, continues to the second call to `exchange_3`, where it enters and skirts/takes a combination of 3 retreating-edge calls and a communication edge, for a total of 9 retreating edges.

The root of LU_2's call graph, `ssor`, calls `rhs` twice: once inside and once outside of a major timing loop. It contains a looping call to `blts`, which in turn calls `exchange_1` twice. `exchange_1` has a similar structure to `exchange_3`, with four sends, four receives and single-nested loops. These calls combine with those in `exchange_3` resulting in 16 communication edges, in which 8 of them cross procedure boundaries: 4 in each direction.

Once we have the depth for each benchmark graph, we are able to calculate the upper-bound number of iterations for a data-flow analysis per benchmark. We collected the number of variables per benchmark per graph as an expression of the lattice height and calculated the iteration upper bound as the product of the depth and the number of variables. Table 4 displays these results for each benchmark. The left half of the table contains results for the MPI-ICFG graph and the right half for the ICFG. In the first three columns for each graph we record the number of variables, the depth, and the iteration upper bound. For these benchmarks, the number of variables is much larger than the depth and has a larger impact on the iteration upper bound. The iteration upper bounds range from one to tens of thousands. Should the data-analysis actually approach these upper bounds, the data-analysis would be prohibitively time consuming.

Fortunately, the actual number of data-flow analysis iterations is strongly correlated with the depth and the lattice height does not have a significant effect. We performed two data-flow analyses, reaching constants and vary, on each graph per benchmark. In Table 4, we recorded the number of iterations of each analysis. The number of iterations for the reaching constants analysis is listed in the fourth column for each type of graph, with the number of iterations for the vary analysis listed in the last column. The actual number of iterations per analysis is far below the upper bound. The largest number of iterations was in the SOR benchmark, 17 for each graph and analysis. There was little difference between the actual number of iterations per analysis between the ICFG and the MPI-ICFG. The single difference occurred in the reaching constants analysis over the MPI-ICFG of LU_2, taking two more iterations than that over the ICFG.

## 6. Conclusions

This paper explores the algorithmic complexity of performing data-flow analysis on MPI programs when the program is modeled with an MPI-ICFG graph. An MPI-ICFG is an interprocedural control-flow graph with communication edges between sends and receives. More precise data-flow analysis is possible for nonseparable analyses when it is possible to model the flow of data over communication edges. Our results show that the depth of the MPI-ICFGs is a good indicator for how many iterations over the graph will be needed for convergence. We compute the depth for some benchmark MPI programs despite the fact that computing the depth for irreducible graphs is NP-complete. We also present an algorithm that builds on the 1976 Fong-Ullman algorithm for computing the depth of a reducible graph. By reducing the reducible parts of the graphs first, the graphs on average became 80% smaller with 90% fewer cycle-free paths, resulting in a 10x faster algorithm.

## Acknowledgments

We thank Karen Degi for an initial version of the depth-analysis algorithm. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy (DOE) under award #DE-FG02-06ER25724 and contract DE-AC02-06CH11357.

**Table 4.** Depth, upper bound and actual iterations for ICFG and MPI-ICFG.

| | MPI-ICFG | | | | | ICFG | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Up. Bound | Actual Iter. | | | | Up. Bound | Actual Iter. | |
| Benchmark | # Vars | Depth | Iter. | R-Consts | Vary | # Vars | Depth | Iter. | R-Consts | Vary |
| Biostat | 208 | 7 | 1456 | 5 | 6 | 202 | 5 | 1010 | 5 | 6 |
| CG | 299 | 12 | 3588 | 5 | 7 | 285 | 5 | 1425 | 5 | 7 |
| LU_1 | 431 | 9 | 3879 | 5 | 9 | 417 | 8 | 3336 | 5 | 9 |
| LU_2 | 1697 | 16 | 27152 | 10 | 16 | 1631 | 12 | 19572 | 8 | 16 |
| MG_2 | 517 | 6 | 3102 | 7 | 9 | 503 | 4 | 2012 | 7 | 9 |
| SOR | 863 | 10 | 8630 | 17 | 17 | 821 | 5 | 4105 | 17 | 17 |
| Sw_1 | 376 | 8 | 3008 | 8 | 11 | 374 | 8 | 2992 | 8 | 11 |

## References


[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley Longman, 1983.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools, second edition*. Pearson Addison Wesley, Boston, MA, USA, 2007.

[3] F. Allen and J. Cocke. Graph theoretic constructs for program control flow analysis. Technical report, IBM Research Report RC 3923, Thomas J. Watson Research Center, Yorktown Heights, N.Y., 1972.

[4] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137, 1976.

[5] 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.

[6] 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.

[7] G. Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[8] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):341–395, 1990.

[9] L. Carter, J. Ferrante, and C. Thomborson. Folklore confirmed: reducible flow graphs are exponentially larger. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 106–114, New York, NY, USA, 2003. ACM.

[10] J. Cocke and R. E. Miller. Some analysis techniques for optimizing computer programs. In *Proceedings of the 2nd Hawaii Conference on System Sciences*, pages 143–146, 1969.

[11] 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.

[12] A. C. Fong and J. D. Ullman. Finding the depth of a flow graph. In *STOC '76: Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, pages 121–125. ACM Press, 1976.

[13] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.

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

[15] M. W. Hall, J. M. Mellor-Crummey, A. Carle, and R. G. Rodriguez. FIAT: A framework for interprocedural analysis and transformation. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 522–545, 1994. ISBN 3-540-57659-2.

[16] L. Hascoët, U. Naumann, and V. Pascual. "To Be Recorded" analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8), 2004.

[17] M. A. Heroux. Software challenges for extreme scale computing: Going from petascale to exascale systems. *International Journal of High Performance Computing Applications*, 23(4):437–439, 2009.

[18] D. Hovemeyer and B. Pugh. Finding bugs is easy. In *OOPSLA*, 2004.

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

[20] 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.

[21] 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.

[22] Lawrence Livermore, Los Alamos, and Sandia National Laboratories. The Accelerated Strategic Computing Initiative (ASCI) sweep3d benchmark code. `http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html`, 1995.

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

[24] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA, 1997.

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

[26] 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. URL `http://www.cis.udel.edu/ hiper/passages/pubs.htm`.

[27] 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.

[28] M. M. Strout, B. Kreaseck, and P. D. Hovland. Data-flow analysis for MPI programs. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, August 2006.

[29] 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.

[30] J. D. Ullman. Fast algorithms for the elimination of common subexpressions. *Acta Informatica*, 2(3), 1973.