# Differences in Algorithmic Parallelism in Control Flow and Call Multigraphs

Vincent Sgro and Barbara G. Ryder*
Rutgers University

May 17, 1994

Our parallel hybrid analysis methods facilitate the parallelization of the analysis phase of a software transformation system, by enabling deeper semantic analyses to be accomplished more efficiently than if performed sequentially. Our previous empirical studies profiled these hybrid techniques on the Reaching Definitions problem [LMR91, LR92a, LR92c]. Recently, we have applied our method to the Interprocedural May_Alias Problem for Fortran programs in a prototype implementation. The interpretation of our results suggested further performance studies, comparing our region partition algorithms (both Bottom_Up and Forward partitioning [LRF94]) on call multigraphs and control flow graphs. These comparisons yielded statistically significant differences in performance of our graph partitioning techniques applied to these two graph populations. This research is part of a larger effort to calculate the modification side effects problem (MOD) for Fortran programs using our parallel hybrid techniques.

## 1 Introduction

*Data flow analysis* is a compile-time technique that gathers semantic information about a program from its source code. Many software applications can use more precise data flow information than is practically available with sequential data flow algorithms today; often precision is sacrificed for analysis speed. Seldom is information calculated across procedure boundaries; however, many software transformation tools need such information to be most effective. These include debuggers, data-flow-based testers, optimizing sequential compilers, parallelizing compilers, program slicers, semantic program browsers and semantic change analyzers [Ryd89]. Our parallel hybrid data flow analysis algorithms are aimed at speeding up time-consuming analysis calculations by utilizing multiple processors that soon will be found in desktop workstations.

Our parallel hybrid algorithms are general-purpose data flow solution procedures, based on the family of sequential hybrid data flow algorithms [MR90a]. These algorithms decompose a flow graph perinto regions, solve local problems on each region and stitch together their solutions to find the global solution on the entire graph. These algorithms are an amalgam of fixed point iteration (i.e., within the regions) and elimination methods (i.e., propagation from region to region). Previously, we profiled the performance of the original parallel hybrid algorithm on the Reaching Definitions problem [ASU86] on a distributed memory machine, an iPSC/2 at the Cornell Theory Center. We published empirical results for this unoptimized algorithm for Reaching Definitions on Fortran procedures from *Linpack* and *BCF*, a curve-fitting program [LMR91]. Subsequently, we developed the *region partition* optimization and presented empirical results [LR92a, LR92b] comparing optimized and unoptimized parallel hybrid algorithm performance on a larger data set, using the same Reaching Definitions problem. These newer studies included flow graph characteristics for our data that proved to be significant to algorithm performance. In [LR92c], we offered an initial look at these flow graph properties for SPICE, one program in the Perfect Benchmarks, and also presented improvements in

*This work was supported, in part, by National Science Foundation grants CCR92-13518 and CCR90-23628.

algorithm performance for the SPICE procedures. These properties impacted the effectiveness of our region partition optimization and revealed *algorithmic parallelism* in our solution process.

Previously, we have discussed three sources of possible parallelism in data flow analysis: *separate-unit* parallelism gained from processing distinct program constructs separately (e.g., loops, procedures etc.); *independent-problem* parallelism gained from solving more than one data flow problem on the same flow graph; and *algorithmic* parallelism gained from finding possible parallelism in the solution procedure itself [LR92a]. Our parallel hybrid algorithms decompose the flow graph of a program into regions, perform local analyses on these regions, do a global propagation of information between regions and then propagate global information within regions. A good decomposition must strike a balance between the amount of parallelism and an appropriate grain size for computation. Thus, the ability of the region partitioning algorithms to determine a good grain size for computation is crucial to the success of our approach, especially on distributed memory machines. The performance of these partitioning techniques proved most significant in interpreting the results of algorithm performance on an interprocedural data flow problem versus an intraprocedural problem.

This paper describes our experiences with an implementation of the Interprocedural May_Alias problem for Fortran. The disappointing speedups obtained led us to subsequent study of the effectiveness of our region partitioning methods on call multigraphs as compared to control flow graphs. We developed two metrics, *Region_Size* and *Interregion_Communication*, to measure the relative goodness of a given decomposition and then used these metrics to compare the effectiveness of both of our *Bottom_Up* and *Forward* partitioning techniques [LRF94, LR92b] on call multigraphs and control flow graphs. After statistical analyses of these measures on two data sets of 34 call multigraphs and 696 control flow graphs, we found a significant difference in the structure of these two graph populations that influenced the decompositions induced. This work is an important step toward optimizing our parallel hybrid algorithms to work on interprocedural problems, and eventually, to solve the family of problems in Fortran MOD. By design, hybrid algorithms exemplify algorithmic parallelism. By using all three forms of parallelism mentioned above, we believe we can effectively parallelize the entire analysis phase of a software tool; this is the eventual goal of our research.

In the rest of the paper, we briefly define necessary data flow analysis concepts and describe our parallel hybrid algorithms in Section 2. Then, we detail our experiences with our implementation of the Interprocedural May_Alias problem on an iPSC/2 at the Cornell Theory Center in Section 3. The performance metrics we developed and the data collected on them from our call multigraph and control flow graph populations also are explained. The *effective call multigraph*, a subgraph developed to aid graph decomposition into regions, is defined and demonstrated in Section 4. We discuss related work in parallel data flow analysis in Section 5. Finally, we present our conclusions and plans for future work in Section 6.

# 2 Background

## 2.1 Data Flow Analysis

Data flow analysis involves the collection of information about how variables and expressions are defined and used in a program [ASU86]. It is performed under the common assumption that all execution paths in the program are actually feasible, that is, traversable on some program execution. Barth terms this assumption *precise up to symbolic execution* [Bar78]. In order to do this analysis, a program is transformed into an internal representation of its control and data flow, called a *flow graph*. A *flow graph* $G = (N, E, \rho)$ is a rooted, directed graph with the unique root $\rho$ such that for any node $v \in N$ there is a path from $\rho$ to $v$. In *intraprocedural* analysis, a procedure is abstracted as a *control flow graph*, in which a node represents a *basic block*, and an edge represents a possible control transfer from one basic block to another [ASU86]. In *interprocedural* analysis, a program is abstracted as a *call multigraph*, in which a node represents a procedure and an edge, a possible procedure call [Hec77].[1]

[1] In the following discussions, we will use the term *flow graph* to mean either control flow graph (intraprocedural flow graph) or call multigraph (interprocedural flow graph).

A data flow problem is either *intraprocedural*, or *interprocedural* depending on whether the data flow information is propagated within one procedure or, in addition, between procedures. The *Reaching Definitions* problem is intraprocedural; its solution at a flow graph node reveals which value-setting statements can affect the values of variables used in code at that node [ASU86]. The *May_Alias* problem for Fortran is interprocedural; its solution for a procedure lists pairs of variables which refer to the same storage location during some execution of that procedure. May_Alias is used in the solution of the Modification Side Effects problem, (MOD) [Ban79, Bur90, CK87, CK89].

Data flow problems can be uniformly represented by a lattice theoretic model termed *a data flow framework*, which formally defines a lattice of solution values for the problem, a function space of functions that describe the semantic effect of traversing a node or edge in the flow graph, and the flow graph of this problem instance [Hec77, MR90b]. General purpose solution procedures for data flow analysis are explained as solving a set of equations, defined on the flow graph nodes and/or edges, that describe the data flow problem [RP86, MR90b]. The structure of the equations is determined by the problem-specific data flow functions and the shape of the flow graph. Solutions are elements in the problem lattice.

There are three families of general-purpose solution procedures for data flow analysis: *iterative, elimination* and our *hybrid* technique [MR90a], the latter an amalgam of the first two approaches. An *iterative* algorithm, based on fixed-point iteration, starts with a safe, initial solution, and then proceeds to a maximum fixed point for the equations; this technique requires that the data flow equations be *monotone* [Hec77]. An *elimination* algorithm has two phases and is conceptually similar to Gaussian elimination [RP86]. In the *elimination* phase, the flow graph is partitioned into intervals (or regions), local data flow information within an interval is summarized, and the interval is condensed into a node. The process continues until there is only one node left, for which the data flow problem is easily solved. In the *propagation* phase, each node in a condensed graph is expanded into an interval, and global data flow information is propagated into that interval. This continues in reverse order on the sequence of condensed graphs, until the original flow graph is solved.

A *hybrid* data flow algorithm uses the strongly connected component (SCC) decomposition to divide a flow graph into single-entry node regions [MR90a]. Intuitively, the hybrid algorithm solves local data flow problems defined within regions iteratively and then propagates global information on the condensed region graph in an elimination-like manner. The hybrid algorithm is applicable to *factorable* data flow problems; most interesting intraprocedural and interprocedural problems have been shown to be factorable [MR90a, Mar89, MR91], although constant propagation is not.

In addition to the above general-purpose algorithms, there is a family of data flow solution procedures for specific intraprocedural problem called the *partitioned variable technique (PVT)*; these partition the solution of a data flow problem by variables and find the relevant information for a single variable, one at a time [Zad84].

## 2.2 Parallel Hybrid Data Flow Algorithm

Our parallel hybrid data flow analysis algorithms are fashioned from the sequential hybrid algorithm described briefly above by finding algorithmic parallelism in the solution process. The parallel hybrid algorithm is organized in a *master-worker* model of parallelism. A master task performs problem initialization and task creation. The worker tasks are associated with each region; some tasks are *independent,* while others, associated with propagation of global information through the condensed flow graph, are *interdependent.*

The phases of the parallel hybrid algorithm[2] are [LMR91]:

1. *Flow graph condensation.* Construct the flow graph and find its strongly connected components. Find a topological order for the strongly connected component condensation, whose nodes represent flow graph regions.

[2] applied to a forward data flow problem in which information is propagated along the direction of execution [Hec77].

2. *Problem setup.* Determine local information from flow graph nodes (e.g., variables, value-setting statements), setup the local data flow problem lattice on each region and set up the global problem lattice.

3. *Local solution.* In each region, iterate the local data flow problem to solution.

4. *Global propagation.* In topological order on the condensed graph, propagate the global data flow information. For each region, combine the local problem solution with incoming global information at the region entry node to obtain global information at region exit edges.

5. *Local propagation.* Use the local problem solutions to compute global data flow information for every node in every region.

The master tasks are flow graph condensation and problem setup; each region spawns one worker task each for local solution, global propagation and local propagation. The interdependences of tasks associated with different regions are captured by the edges of the condensed flow graph. Tasks associated with the same region must be performed in the following order: local solution, global propagation, local propagation.

Our algorithm design was targeted for distributed memory architectures. A simple mapping strategy was used; all tasks associated with a region were mapped to the same processor to insure data locality. In addition, a compile-time mapping strategy estimated the work per region and statically allocated tasks to attempt to load balance the processors and avoid run-time overhead. We built local dynamic schedulers on each processor, using the following task priorities: (highest to lowest): global propagation, local solution, local propagation.

Initially, we used the sequential hybrid algorithm directly as a model for our parallel data flow method, but there were too many trivial components in the strongly connected component decompositions. The computation in these regions was too fine-grained, and diminished the savings we obtained by parallelizing the solution process. Therefore, we designed two region partition techniques, *Forward* and *Bottom_Up*, to design more effective parallelism in our graph decompositions; both preserve the necessary single-entry node property (i.e, the *entry constraint*) of our regions [LMR91, LR92b, LRF94]. The Forward clustering technique is based on Allen/Cocke intervals [AC76], tempered with a maximal size (i.e., the *size constraint*).[3] The Forward method forms regions by proceeding along the direction of execution flow on flow graph edges. The entry constraint is satisfied, since we include a node in a region only if all of its immediate predecessors are already in the region. The size constraint is satisfied since we include a node in a region only if the resulting region has its size no larger than $S$. The advantages of the Forward algorithm are efficiency and ease of implementation; however, it uses no knowledge of the graph structure and proceeds in an oblivious manner. This can result in very poor partitioning for a graph with many leaf nodes (i.e., nodes with no successor), for example, a balanced tree with every leaf a region.

The Bottom_Up method uses the dominator tree of the flow graph to direct the clustering and a topological order on the flow graph to prevent multiple-entry regions from being formed. In this algorithm, children are merged with parents in the dominator tree as long as the single-entry condition is not violated and the maximal size is not exceeded. Because of nested control structures, there may be paths from one dominator tree sibling to another in the flow graph. If we use a topsort order on the flow graph to order sibling nodes, we can show that the entry constraint will be maintained.

We have shown that finding a minimal region partition for a fixed maximal region size is *NP hard* [Lee92], so our techniques are actually approximation algorithms. Both of these techniques are *greedy* in that they try to make the region as large as possible without violating the size constraint.

## 2.3 May_Alias Problem Definition

The May_Alias problem is important for calculation of side effects in a program because aliases mean that variables which don't explicitly appear in a value-setting statement may still suffer side effects from the

[3] This will be represented as $S$.

statement. There are solutions for this problem for languages like Fortran where call by reference parameter passing methods are the only sources for dynamic aliasing [CK89]. These are formulated on the call multi-graph representing possible procedure calls in a program. Algorithms for Fortran May_Alias depend on the following observations:

**Alias Introduction:** There are specific constructs a program that introduce aliases:

1. A single variable occurs in more than one position in a call. The corresponding formals in the called procedure will be aliased during that call.
2. A global variable occurs as an actual in a call. The corresponding formal in the called procedure will be aliased to that global during that call.

**Alias Propagation:** Once an alias is created, there are specific constructs that propagate these aliases to other procedures. Assume $< x, y >$ are aliased in procedure $M$ at a call of procedure $N$ :

1. If $x$ and $y$ are formal parameters of $M$ used as actuals in the call to procedure $N$, then the corresponding formals of procedure $N$ will be aliased on entry to $N$.
2. If $x$ is a formal parameter of procedure $M$ is used as an actual in the call to procedure $N$, and $y$ is global to $N$, then the formal corresponding to $x$ in $N$ and the global $y$ will be aliased on entry to $N$.
3. If $x$ is a formal parameter of procedure $M$, $y$ is global to procedure $N$, and both are used as actuals in the call to procedure $N$, then the corresponding formals of procedure $N$ will be aliased on entry to $N$.

## 2.4 May_Alias Problem Formulation for Hybrid Algorithm

The hybrid algorithm solution for this problem requires the definition of four local problems on each region [MR91]:

**Restricted Problem - R:** This is the May_Alias problem restricted to the region; that is, we solve for May_Alias as though the region were the entire program. For this we use the same formulation as was described above, with alias introduction and propagation.

**The Formal Pair Problems - F:** This is a set of problems that solve for possible aliases between 2 formal parameters of a procedure within the region, induced by an assumed alias between two region entry node formals. Each F problem corresponds to a different entry node formal alias pair. Alias propagation rule 1 is used to propagated these aliases through the region.

**The Head Global Problems - H:** This is a set of problems that solve for aliases between a formal within the region and a global variable. Each problem assumes that a representative global variable is aliased to a region entry node formal. These aliases are propagated by alias propagation rule 2 throughout the region.

**The Internal Global Problem - I:** This is a set of problems that correspond to the situation in alias propagation rule 3, namely when a formal is aliased to a global and both that formal and that global are used as arguments in a procedure call. Given an instance of this situation involving formal $f$ and global $g$, we use the H problem for $f$ to initialize the corresponding I problem, by substituting $g$ for the representative global and using alias propagation rule 3 to propagate this alias into alias pairs involving the corresponding two formals of the called procedure. Then, alias propagation rule 1 is used to further propagate these aliases.

As can be seen, these local problems are rather complex, although not difficult to solve. The example problem and its solution in Figure 5 shows May_Alias formulated with our sequential hybrid algorithm.

# 3 Investigations

## 3.1 May_Alias Implementation

A team of undergraduate students coded a solution to the May_Alias problem as previously described in [CK89] on call multigraphs using Bottom_Up partitioning on an iPSC/2. Discouragingly, the reasonable speedups realized on control flow graphs were not obtained from our experiments with call multigraphs. There are many possible reasons why this speed increase was not realized, but it is difficult to calculate which was the most significant. The implementors of the algorithm used good software engineering methods for ordinary platforms; however, they did not translate very well to a distributed, parallel machine environment. For the most part, the problems were the result of using object oriented programming techniques. These techniques, though they lend themselves to well organized and maintainable code, put a level (or more) of abstraction between the programmer and the machine architecture. More appropriate data structures and operations could have been chosen, but were not, since they existed in a different abstraction level than the programmers were coding. Examples of these include:

**Solution Representation.** The hybrid algorithms for Reaching Definitions were formulated as *bit vector* problems; the May_Alias problem, on the other hand, was not. The data flow information, sets of pairs of variable names, was kept as unencoded strings, a form that was far less efficiently manipulated. The equality test used was a simple string comparison which introduced context switching overhead. Performing set unions and membership tests required quite a lot of work.

**Message Encoding.** Since the processors existed on a distributed machine, the method of information transfer was through message passing. The data communicated was a representation of flow graph structure. The machine was designed to pass arrays of information as messages. If the low-level representation had been some sort of array structure (e.g., an adjacency matrix), then the message encoding would have been straightforward; however, the graphs were represented as structures and pointers, which had to be translated twice every time a message needed to be passed.

**Heap Manipulation.** There were several places where blocks of memory were allocated and immediately freed within a tight loop. It was not evident that this was happening since it was buried deep in the object structure.

In addition to the problems already cited, there are differences between call multigraphs and flow graphs that may have had an influence on the performance of the analysis algorithms. Since the May_Alias problem involved call multigraphs, and the programs being analyzed in the experiments were Fortran programs which had no recursion, the graphs had only trivial strongly connected components. Thus, in the absence of loops in the graph, there was no need to do iteration. One topologically ordered pass through a region was enough to obtain a local solution. These local tasks contain little computation and thus, the communication between processors probably dominated all computation. In comparison, the control flow graphs did contain many loops, so there were many non-trivial components on which iteration was necessary to solve the Reaching Definitions problem.

Also, the *shapes* of control flow graphs and call multigraphs are quite different. Control flow graphs have nodes with no more than two (usually) edges leaving them. Call multigraphs, on the other hand, may have many edges leaving and entering them. This difference in shape may affect the performance of the graph partitioning algorithms.

The work required by the data flow problems that we profiled was quite different. It is not clear that the quantity of information being manipulated in the control flow and call multigraph problems was the same.

## 3.2 Metrics

The performance differences cited above clearly pointed to graph structure as a possible major discriminant between the two experiments. Therefore, we focused our attention on measuring the heuristic performance of the two graph partitioning algorithms.

**Algorithm Constraints and Desirada.** A well distributed parallel execution would be characterized by regions of moderately balanced size (i.e. tasks) and minimal inter-region communication. Having regions of approximately equal size helps to simplify task assignment. Since the Fortran call multigraphs did not have any cycles, the size of the region was directly related to the amount of computation required for finding local solutions and doing local propagation. In distributed memory machines, the cost of communication is quite high. If the solution found by a partitioning algorithm has a great many edges between the regions, then there is a risk that cross-processor communication will dominate the computation; then parallelization of the analysis will only increase the total processing time.

Though the size constraint designed into the partitioning algorithms is closely related to the idea of balanced region size, it is not clear whether an algorithm that fulfills the size constraint will fulfill the balanced region criterion optimally. Also, the entry constraint does not guarantee to minimize the number of edges between regions. In fact, the balanced region criterion and communication constraint may interfere with each other, a possible source of trouble with the implementation of the parallel hybrid aliasing algorithm. To test the significance of this difference, some metrics were developed and used to compare the performance of the Forward and Bottom_Up algorithms on control flow and call multigraphs.

**Region Size Metric.** The primary reason for performing graph partitioning is to identify convenient sets of graph nodes that can be processed separately, thus creating parallel tasks. However, once the number of regions exceeds the number of processors being used on the parallel machine, mapping multiple regions to the same processor needs to be done. In such a situation, load balancing can become a problem particularly with regions that have cycles since, in the presence of cycles, the amount of work needed to process a region becomes unpredictable. However, in our experiment, aliasing analysis was performed on the call multigraphs of Fortran 77 programs for which recursion was not supported. Therefore, there were no cycles to complicate the estimation.

We want regions corresponding to tasks that are easily mapped to processors. To evaluate the effectiveness of the partitioning algorithms to perform this task, a *quality measure* was needed. The partitioning algorithms have a parameter that represents maximal region size. Presumably, this parameter is chosen based on the number of processors available to the algorithm. It was, therefore, assumed that the best an algorithm could do was to divide the call multigraph into regions each exactly the size specified. (Of course, one may have to be smaller.) Using this assumption, the difference between the number of regions actually formed and the best the algorithm could have done, produces a useful measure of algorithm quality with respect to region sizing. Therefore, the Region_Size metric is:

$$Region\_Size = 1 - \frac{(a - \frac{n}{S})}{n - \frac{n}{S}}$$

where $a$ is the number of regions actually formed by the partitioning algorithm, $n$ is the number of call multigraph nodes in the graph being processed, and $S$ is the maximum size parameter. The numerator is the distance between the optimal partitioning with respect to size and the partitioning obtained. The denominator is a normalization factor. Subtracting the normalized difference from one provides a more intuitive measure, in that the closer Region_Size is to one, the better the algorithm performed.

It may be argued that the standard deviation of the sizes of the regions may be better to use for this measure. However, there are situations where the standard deviation discerns too strictly. For example, assume there is a call multigraph with twelve nodes and that there are three processors available. The maximum region size is chosen to be $S = 4$ in the hopes that there will be three regions of four nodes each, giving each processor one region, as in Figure 1. However, the algorithms are very unlikely to actually produce such a partitioning. They may also produce five regions of varying sizes. Figure 1 shows two such possible suboptimal partitionings. It will take more work to decide on a processor mapping in the two cases with five regions than when there are three. However, the actual work done in the two five region cases is equivalent, ignoring that of context switching between regions. The standard deviation would differentiate between the two, incorrectly showing suboptimal 2 to be superior over suboptimal 1. Our Region_Size metric will not make that distinction.

**Minimizing Communication.** Since communication throughput is a known bottleneck in parallel processing, it is also an objective of the partitioning algorithms to form regions that will cause the tasks that they represent to have as little data interdependence as possible. Since aliases are propagated through function and procedure calls, the number of call multigraph edges into a region provide a good estimate of the degree that each region depends on information from sources outside that region.

To simplify, we will measure how well each algorithm does with respect to inter-region communication by counting the number of inter-region call multigraph edges. The worst either algorithm can do is to have every edge be an inter-region edge. Therefore, the quality measure is:

$$Interregion\_Communication = 1 - \frac{a}{e}$$

where, $a$ is the actual number of inter-region edges and $e$ is the total number of edges in the call multigraph. As with the Region_Size quality measure, the normalized value is subtracted from one to provide a more intuitive value. The closer Interregion_Communication is to one, the better the algorithm performance in reducing the inter-region communication.

## 3.3 Experiments

The two partitioning methods, Forward and Bottom_Up, were tested against each other on the same set of graphs to show if, in general, one performs better than the other. Then, each method was tried on both control flow graphs and call multigraphs to see if the graph type affects their performance.

**Data Set Quality.** First, all the analyses are being performed on a limited set of Fortran programs, mostly scientific applications and libraries.[4] The application area may have a strong impact on the structure of the graphs and, therefore, possibly the performance of the partitioning algorithms. It should be noted that the results given here may or may not be extendable across a range of applications. However, this data does explain the differences in performance we experienced between the Reaching Definitions and May_Alias implementations, since the graphs used here are those generated in these experiments.

Second, the number of control flow graphs available vastly outnumbered the number of call multigraphs.[5] The $t$-test used in this section is normally rather stable so this should not matter much. Nonetheless, although the results shown are for the entire sample, the tests were verified on random sub-samples of control flow graphs chosen so that the sample sizes would be equal.

**Methods.** The statistical methods involved summarizing the results of the tests and expressing the differences that these summaries show.[6] The first step in the tests was to apply each partitioning method to each call multigraph and control flow graph while varying the size parameter $S$ that each method uses, from two to half the number of nodes in each graph. Then, the two metrics were calculated for each partitioning. Thus, we obtained a set of performance results for each method on each graph. For example, the leftmost plot of Figure 2 shows the performance of the Bottom_Up partitioning algorithm, as $S$ is varied from two to half the graph size. Notice how the data points are not linear.

The next step was to find a way to summarize the results of the test so that simple statistical tests could be performed on the data. It was decided that the data would first be *transformed* so that the results be as close to a linear relationship to the size parameter $S$ as possible. The transformation used was: $q/(1-q)$, where $q$ was the value of the quality measure. A regression was applied to the resulting linearized data and the $x-$intercept and slope of the fitted lines were used as the summary information. The $x-$intercept is a rough estimate of how the algorithms perform in the lower values of $S$ and the slope expresses the improvement of the algorithms as the $S$ parameter increases. For example, the rightmost plot of Figure 2

[4] We have found it difficult to obtain a broad range of applications through user solicitation and investigation of repositories on the Internet.

[5] The data set size difference was due to the fact that any one program may contain many control flow graphs, but only one call multigraph.

[6] The methods used were suggested by Dr. Javier Cabrera of the Rutgers University Statistics Department.

shows the same data as the leftmost plot, after it has been transformed. Notice how the data is closer to the regression line.

Scatter plots of the $x-$intercept against slope showed clusters of points that seemed differentiated in some way. This was especially true for the Region_Size metric. For example, Figure 3 is a scatter plot of the results of the Region_Size metric calculated from partitionings obtained by the Bottom_Up algorithm on both the control flow graphs and the call multigraphs. The differences in the data points of the plot in Figure 3 are almost linearly separable. Other plots were not as clean, but the means of the clusters were significantly different.

To show the significance of the separation, a $t-$test was used on the means of the intercepts and the means of the slopes. For this test, the null hypothesis is:

$$H_0 : \mu_1 - \mu_2 = 0$$

where $\mu_1$ and $\mu_2$ are the population means we are testing. Since we will always assign the higher mean to $\mu_1$, the alternative hypothesis is:

$$H_a : \mu_1 - \mu_2 > 0$$

What is being done is a test for a significant difference of the means. If we *do not reject* the null hypothesis, then there is insufficient evidence to indicate that the means are different. If we *reject* the null hypothesis, then we can conclude that there is a significant difference and, further, that $\mu_1$ is the greater of the two.

In all of the analyses, the statistical confidence parameter $\alpha$ was chosen to be $\alpha = .025$, except when otherwise noted. This means that if we conclude that there is a significant difference in the population means, then there is a .025 probability that we do so erroneously.

### 3.4 Results

The most dramatic of the differences can be seen in the analysis of the Bottom_Up and Forward partitioning methods across the two types of graphs. In each of the cases, the null hypothesis is rejected and a significant difference is found. It may be concluded with some measure of confidence that both the algorithms perform better on control flow graphs than they do on call multigraphs. The results can be seen in Table 1.

The performance difference between the Bottom_Up and Forward methods on the same type of graph is less dramatic and can be seen in Table 2. In each case, the slope is significantly better for the Bottom_Up method. This seems to imply that, with some measure of confidence, the Bottom_Up algorithm improves faster than the Forward method as the size limit parameter $S$ increases. On the other hand, there was insufficient evidence to conclude that either method did better on the $x-$intercept. In fact, in the first case, the Forward algorithm seems to have done better.[7]

## 4 The Effective Call Multigraph

In an attempt to try to make the task of partitioning easier, modifications can be made to the flow graph being analyzed, such as the removal of unnecessary edges. In the case of May_Alias, this means removing call multigraph edges that can not introduce nor propagate aliases. (See Section 2.3.) The resulting graph is called the *effective call multigraph.* Removing edges can leave the resulting graph disconnected and/or without a unique root node. Since the partitioning algorithms can only deal with graphs that represent legitimate call multigraphs or control flow graphs, an additional transformation was needed. A supernode was created with all the roots as children before the partitioning algorithms were executed. It is important to remember that the edges are removed and the supernode added before the partitioning algorithms are applied.

[7] Here, the Forward algorithm had the higher mean and the null hypothesis was rejected, implying that it may have done significantly better.

The meaning of the metrics is also changed slightly and deserves some explanation. Recall that the Interregion_Communication metric was:

$$Interregion\_Communication = 1 - \frac{a}{e}$$

In this case, we want to see how removing the edges improves the degree of communication relative to the original graph; therefore, we left $e$ as the total number of edges in the original graph. Then, if removing the edges actually causes the partitioning algorithms to reduce the number of inter-region edges, the Interregion_Communication metric increases.

Also, recall the Region_Size metric:

$$Region\_Size = 1 - \frac{(a - \frac{n}{S})}{n - \frac{n}{S}}$$

In this case, we are concerned with how balanced the sizes of the regions are. The balance is relative to the effective graph itself, not the original graph. Therefore, the value of $n$ is the number of nodes in the effective graph and not the number of nodes in the original graph.

The results of the experiment are shown in Table 3. The results of this experiment are more ambiguous. We can no longer, with the same certainty, see a significant difference between the communication performance of the Bottom_Up algorithm applied to the two types of graphs, though the difference still seems to be present in the Region_Size performance. While the Forward algorithm only seems to improve communication faster as the $S$ parameter is increased and has better region sizes at lower values of $S$. These results changed very little when varying the certainty within reasonable limits.

An attempt was made to compare the metrics on an effective call multigraph directly against the corresponding original call multigraph. There seemed only to be a solid significant improvement with the effective call multigraphs at a 90% certainty. The results are shown in Table 4.

# 5 Related Work in Parallel Data Flow Analysis

Previous work by others on parallel data flow algorithms was reported in [Zob90, GPS90, KGS94] Kramer *et al.* presented an approach based on parallel prefix operation (i.e., scan). Although interesting in approach, this can only handle reducible flow graphs as formulated; a "by hand" comparison is offered on the effectiveness of graph decomposition by this algorithm and our unoptimized algorithm [LMR91]. Both Zobel and Gupta *et al.* designed parallel elimination algorithms. Zobel parallelized Allen-Cocke interval analysis [AC76] and reported some empirical experiences for the available expressions problem on five C functions [Zob90].[8] Since her approach had no control over the flow graph partitioning, very large intervals limited parallelism and caused load imbalance. If a flow graph is acyclic, this approach offers no parallelism at all. In their parallel elimination method, Gupta *et al.* attempted to partition a reducible flow graph into single-entry, single-exit regions instead of intervals using a syntax-directed method [GPS90]. Although their approach had some control over the size of regions in flow graph partitioning, the restriction that regions must be *Single-entry and single-exit* required the corresponding program to be well-structured. Thus, in practice, this technique is not sufficiently effective. Additionally, there were no experimental results reported.

# 6 Future Work and Conclusions

Because of the homogeneity of the data studied, it might make sense to repeat our tests using a more heterogeneous data set. If our results are corroborated, then some obvious directions emerge. The statistics showed that the partitioning algorithms do not perform as well on call multigraphs as they do on control flow graphs.

[8] The Available Expressions problem involves information necessary for common subexpression elimination [ASU86].

Methods of improving this performance should be investigated. Currently, we are working on improving the Bottom_Up algorithm by using heuristics to choose which of the children of a node in the dominator tree will be merged into its region. Hopefully, our heuristics will lower the amount of communication between regions and obtain better load balancing.

Addressing the larger problem of "What went wrong with our aliasing implementation?" it may be that the May_Alias problem has a small amount of work required at each graph node, unlike Reaching Definitions. If this is so, then there may not be enough work for the processors to do in each worker task that would allow them to overcome the communication bottleneck. We hypothesize that adding other problems to be solved in conjunction with the May_Alias analysis will improve the situation. As stated previously, we intend to investigate the concurrent solution of several interprocedural data flow problems in the context of an implementation of the Fortran MOD problem [LR92a]. Parallelizing the entire data flow analysis phase of a software translation tool is appealing, in that it holds promise of generating enough work for overcoming the communication to computation cost ratio.

Another possible avenue for future investigation is to implement our parallel hybrid algorithm on a shared address space architecture. This might uncover different performance problems which can be addressed.

To summarize our accomplishments delineated in this paper, we built an initial implementation of our parallel hybrid algorithm applied to an interprocedural data flow problem on a distributed memory machine, and discovered an essential difference in the structure of control flow graphs versus call multigraphs. This difference, corroborated by the statistical analyses detailed here, affected how well our region-finding algorithms could decompose these call multigraphs for effective parallel computation. We developed a new representation, effective call multigraphs, which shows promise of yielding better decompositions with our current partitioning algorithms. By better tuning the partitioning algorithms to program calling structures and using this new representation, we hope to attain useful speedups in our next round of experiments with our parallel hybrid algorithms.

**Acknowledgments:** Our colleagues Richard Martin and David Hanson, two Rutgers seniors at the time they were involved with our research, were responsible for the May_Alias implementation; without their diligent efforts, we would not have been able to make the observations in this paper. These students were supported by funds provided by the National Science Foundation under its Research Experiences for Undergraduates program and an Undergraduate Research Internship sponsored by the Rutgers University Provost's Office. We also thank Jyh-shiarn Yur, who provided the empirical data for evaluation of the effective call multigraphs and Dr. Javier Cabrera for suggesting the statistical methods used in this study.

# References

[AC76] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, 1976.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Ban79] J. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 29–41, January 1979.

[Bar78] Jeffrey M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, September 1978.

[Bur90] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[CK87] K. Cooper and K. Kennedy. Complexity of interprocedural side-effect analysis. Computer Science Department Technical Report TR87-61, Rice University, October 1987.

[CK89] K. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49–59, January 1989.

[GPS90] Rajiv Gupta, Lori Pollock, and Mary Lou Soffa. Parallelizing data flow analysis. In *Proceedings of the Workshop on Parallel Compilation*, Kingston, Ontario, Canada, May 1990.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, Amsterdam, Netherlands, 1977.

[KGS94] Robert Kramer, Rajiv Gupta, and Mary Lou Soffa. The combining DAG: A technique for parallel data flow analysis. *IEEE Transactions on Parallel and Distributed Systems*, August 1994. to appear.

[Lee92] Yong-fong Lee. *Performing Data Flow Analysis in Parallel.* PhD thesis, Department of Computer Science, Rutgers University, May 1992.

[LMR91] Yong-fong Lee, Thomas J. Marlowe, and Barbara G. Ryder. Experiences with a parallel algorithm for data flow analysis. *The Journal of Supercomputing*, 5(2):163–188, October 1991.

[LR92a] Yong-fong Lee and Barbara G. Ryder. A comprehensive approach to parallel data flow analysis. In *Proceedings of the ACM International Conference on Supercomputing*, pages 236–247, July 1992.

[LR92b] Yong-fong Lee and Barbara G. Ryder. A comprehensive approach to parallel data flow analysis. Technical Report LCSR-TR-192, Laboratory for Computer Science Research Technical Report, August 1992. This is an expanded version of the ICS92 paper and has been submitted for journal publication.

[LR92c] Yong-fong Lee and Barbara G. Ryder. Parallel hybrid data flow algorithms: A case study. In *Conference Record of 5th Workshop on Languages and Compilers for Parallel Computing, Yale University*, pages 296–310, August 1992. *Springer-Verlag Lecture Notes in Computer Science, Number 757.*

[LRF94] Yong-fong Lee, Barbara G. Ryder, and Marc E. Fiuczynski. Region analysis: A parallel elimination method for data flow analysis. May 1994. to appear in *Proceedings of the IEEE Conference on Computer Languages.*

[Mar89] T. J. Marlowe. *Data Flow Analysis and Incremental Iteration.* PhD thesis, Rutgers University, August 1989.

[MR90a] T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–196, January 1990.

[MR90b] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: a unified model. *Acta Informatica*, 28:121–163, 1990.

[MR91] T. J. Marlowe and B. G. Ryder. Hybrid incremental alias algorithms. In *Proceedings of the Twentyfourth Hawaii International Conference on System Sciences, Volume II, Software*, January 1991.

[RP86] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[Ryd89] B. G. Ryder. Ismm: Incremental software maintenance manager. In *Proceedings of the IEEE Computer Society Conference on Software Maintenance*, pages 142–164, October 1989.

[Zad84] F.K. Zadeck. Incremental data flow analysis in a structured program editor. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 132–143. ACM Press, June 1984. Montreal, Canada.

[Zob90] Angelika Zobel. Parallel interval analysis of data flow equations. In *Proceedings of the 1990 International Conference on Parallel Processing, Vol.II*, pages 9–16. The Penn State University Press, August 1990.

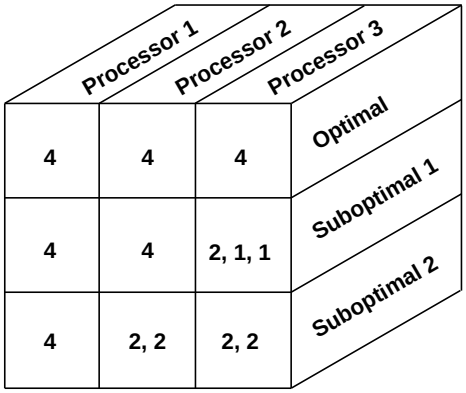


Figure 1: An optimal and two suboptimal partitionings with $S = 4$ and a call multigraph of twelve nodes.

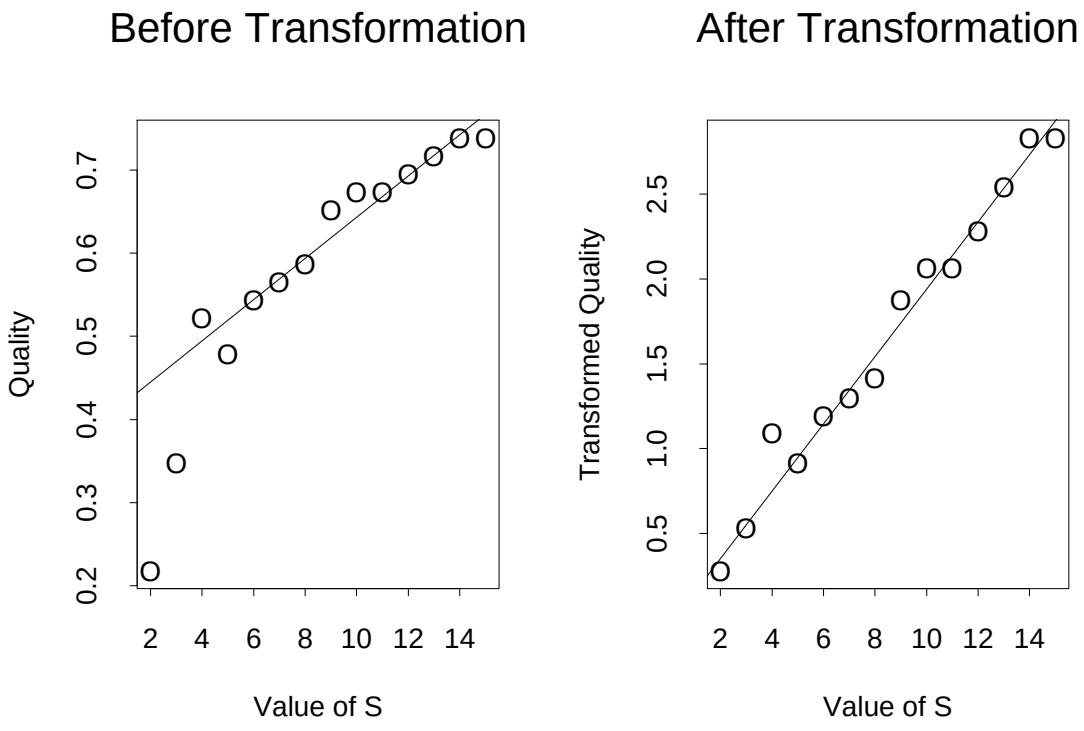


Figure 2: A plot of the performance of the Bottom_Up algorithm on a *single* call multigraph as $S$ is varied. The second plot is the same information after it is transformed.

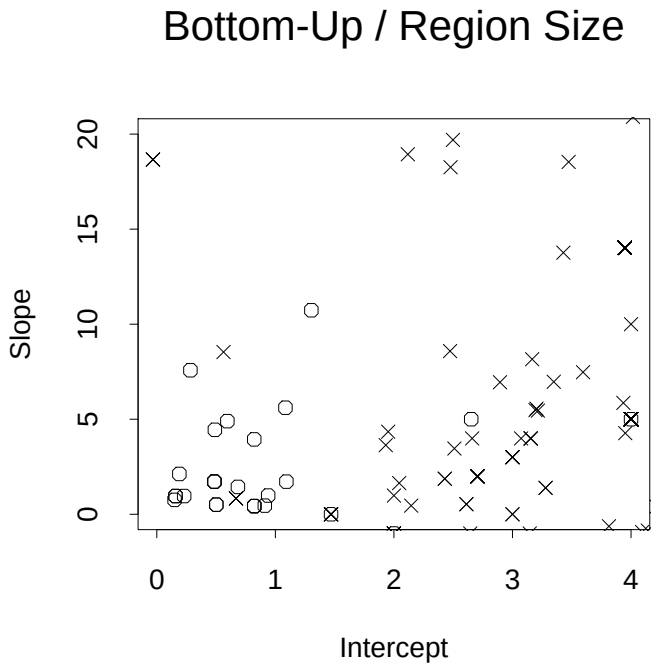


Figure 3: Scatter plot showing call multigraphs (O) and control flow graphs (X).

Table 1: Control flow graphs had the higher mean.

| *Between Control Flow and Call Multigraphs* | | Intercept | Slope |
|---|---|---|---|
| Bottom_Up / Communication | Threshold | 1.96420 | |
| | $t$ statistic | 2.71985 | 4.09056 |
| | Conclusion | **reject** | **reject** |
| Bottom_Up / Region | Threshold | 1.96464 | |
| | $t$ statistic | 3.76389 | 3.060471 |
| | Conclusion | **reject** | **reject** |
| Forward / Communication | Threshold | 1.96420 | |
| | $t$ statistic | 6.51384 | 4.79076 |
| | Conclusion | **reject** | **reject** |
| Forward / Region | Threshold | 1.96464 | |
| | $t$ statistic | 3.75678 | 2.86510 |
| | Conclusion | **reject** | **reject** |

Table 2: Bottom_Up had the highest mean except where the conclusion is italicized.

| *Between Bottom_Up and Forward* | | Intercept | Slope |
|---|---|---|---|
| Control Flow / Communication | Threshold | 1.96223 | |
| | $t$ statistic | 2.09734 | 3.20279 |
| | Conclusion | *reject* | **reject** |
| Control Flow / Region | Threshold | 1.96248 | |
| | $t$ statistic | 1.57803 | 4.71373 |
| | Conclusion | **not reject** | **reject** |
| Call / Communication | Threshold | 2.00488 | |
| | $t$ statistic | 0.21242 | 2.78743 |
| | Conclusion | *not reject* | **reject** |
| Call / Region | Threshold | 2.00488 | |
| | $t$ statistic | 0.73695 | 3.50183 |
| | Conclusion | **not reject** | **reject** |

Table 3: Control flow graphs had the higher mean except where the conclusion is italicized.

| *Between Control Flow and Effective Call Multigraphs* | | Intercept | Slope |
|---|---|---|---|
| Bottom_Up / Communication | Threshold | 1.96423 | |
| | $t$ statistic | 0.71726 | 0.18322 |
| | Conclusion | *not reject* | **not reject** |
| Bottom_Up / Region | Threshold | 1.96468 | |
| | $t$ statistic | 3.47234 | 2.96874 |
| | Conclusion | **reject** | **reject** |
| Forward / Communication | Threshold | 1.96423 | |
| | $t$ statistic | 0.71364 | 3.26978 |
| | Conclusion | *not reject* | **reject** |
| Forward / Region | Threshold | 1.96468 | |
| | $t$ statistic | 3.33632 | 1.10478 |
| | Conclusion | **reject** | **not reject** |

Table 4: Effective call multigraphs had the higher mean except where the conclusion is italicized.

| *Between Call Multigraphs and Effective Call Multigraphs* | | Intercept | Slope |
|---|---|---|---|
| Bottom_Up / Communication | Threshold | 1.29413 | |
| | $t$ statistic | 1.66628 | 1.76561 |
| | Conclusion | **reject** | **reject** |
| Bottom_Up / Region | Threshold | 1.29413 | |
| | $t$ statistic | 1.20558 | 0.50546 |
| | Conclusion | *not reject* | *not reject* |
| Forward / Communication | Threshold | 1.29413 | |
| | $t$ statistic | 3.96464 | 1.56187 |
| | Conclusion | **reject** | **reject** |
| Forward / Region | Threshold | 1.29413 | |
| | $t$ statistic | 1.66628 | 1.76561 |
| | Conclusion | **reject** | **reject** |

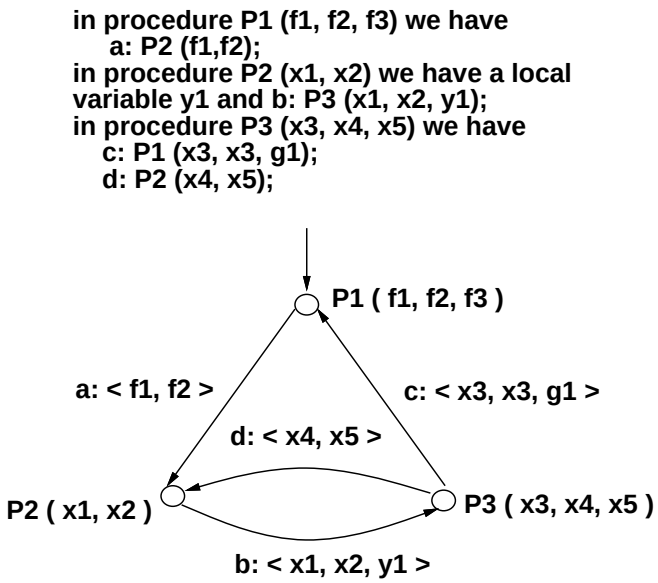


Figure 4: Call multigraph with relevant code

| | | | |
|---|---|---|---|
| $\mathbf{R}$ | $(P1:)\{(g1,f3),(f1,f2)\}$ | $\mathbf{H}\langle G1,f1\rangle$ | $\mathbf{R} \cup$ |
| | $(P2:)\{(x1,x2)\}$ | | $(P1:)\{(G1,f1),(G1,f2)\}$ |
| | $(P3:)\{(x3,x4)\}$ | | $(P2:)\{(G1,x1),(G1,x2)\}$ |
| $\mathbf{F}\langle f1,f2\rangle$ | $\mathbf{R}$ since $(f1,f2) \in \mathbf{R}(P1)$ | | $(P3:)\{(G1,x3),(G1,x4)\}$ |
| $\mathbf{F}\langle f1,f3\rangle$ | $\mathbf{R} \cup (P1:)\{(f1,f3)\}$ | $\mathbf{H}\langle G2,f2\rangle$ | $\mathbf{R} \cup$ |
| | and same for $\mathbf{F}\langle f2,f3\rangle$, $\mathbf{H}\langle G3,f3\rangle$ | | $(P1:)\{(G2,f1),(G2,f2)\}$ |
| $\mathbf{I}\langle g1,f3\rangle$ | $\mathbf{H}\langle G3,f3\rangle[G3 \leftarrow g1] = \mathbf{R}$ | | $(P2:)\{(G2,x1),(G2,x2)\}$ |
| $\mathbf{I}\langle g1,f1\rangle$ | $\mathbf{H}\langle G1,f1\rangle[G1 \leftarrow g1] \cup (P1:)\{(f1,f3),(f2,f3)\}$ | | $(P3:)\{(G2,x3),(G2,x4)\}$ |
| $\mathbf{I}\langle g1,f2\rangle$ | $\mathbf{H}\langle G2,f2\rangle[G2 \leftarrow g1] \cup (P1:)\{(f1,f3),(f2,f3)\}$ | | |

If set of external bindings is $\{(f1,f3),(g1,f2),(g2,f3)\}$, then the alias solution at $Pi$ will be the union of the problems $\mathbf{R}$, $\mathbf{F}\langle f1,f3\rangle$, and the $\mathbf{I}$ problem $\mathbf{I}\langle g1,f2\rangle$, and the two instantiated $\mathbf{H}$ problems, $\mathbf{H}\ \langle G2,f2\rangle[G2 \leftarrow g1]$ and $\mathbf{H}\ \langle G3,f3\rangle[G3 \leftarrow g2]$. Thus,

$$
\begin{aligned}
Alias(P1) &= \{(g1,f2)(f1,f2)\} \cup \{(f1,f3)\} \cup \{(g1,f1),(g1,f2)\} \cup \{(g2,f3)\} \cup \{(f1,f3),(f2,f3)\} \\
&= \{(f1,f2),(g1,f3),(f1,f3),(g1,f1),(g1,f2),(f2,f3),(g2,f1),(g2,f2)\} \\
Alias(P2) &= \{(x1,x2),(g1,x1),(g1,x2),(g2,x1),(g2,x2)\} \\
Alias(P3) &= \{(x3,x4),(g1,x3),(g1,x4),(g2,x3),(g2,x4)\}
\end{aligned}
$$

Figure 5: Alias Solution for above call multigraph