A Comprehensive Approach to Parallel Data Flow Analysis*

Yong-fong Lee† **Barbara G. Ryder**†

Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

Abstract

We present a comprehensive approach to performing data flow analysis in parallel. We identify three types of parallelism inherent in the data flow solution process: independent-problem parallelism, separate-unit parallelism and algorithmic parallelism; and describe a unified framework to exploit them. Our investigations of typical Fortran programs reveal an abundance of the last two types of parallelism. In particular, we illustrate the exploitation of algorithmic parallelism in the design of our parallel hybrid data flow analysis algorithms. We report on the empirical performance of the parallel hybrid algorithm for the Reaching Definitions problem and the structural characteristics of the program flow graphs that affect algorithm performance.

Keywords. Data flow analysis, parallel algorithms, parallel data flow analysis.

1 Introduction

1.1 Motivation

Data flow analysis is a compile-time analysis technique that gathers information about the flow of data in the program. Data flow information is essential in program development environments [CKT86a], testing [RW85, OW91], debugging [Wei84], software maintenance [Ryd89], program integration [HPR89, YHR90], program parallelization [BC86, PW86], and compiler optimization

*This research was supported, in part, by National Science Foundation grants CCR-8920078 and CCR-9023628-1/5. An earlier version of this paper appears in Proceedings of the 6th ACM International Conference on Supercomputing, pp. 236-247, Washington, D.C., July 1992.

†Email addresses: {lee,ryder}@cs.rutgers.edu

[ASU86]. This information can be approximated at varied levels of precision; its utility varies directly with its precision. Computation of precise data flow information needs to account for data flow effects across procedure boundaries. Since this kind of interprocedural analysis can be very time-consuming, it is often either not performed or performed approximately.

An example of approximation is provided by Loeliger *et al.* in [LMSS91]. To facilitate vectorization and parallelization of C programs, they perform an analysis called pointer target tracking that determines the range of memory locations potentially pointed to by a pointer during execution. This analysis is performed not only within individual procedures, but also across procedure boundaries. In order to make their interprocedural pointer tracking analysis practical, they assume that the range for a global pointer variable on entry to a procedure is the union of all ranges associated with that variable during the execution of *all* procedures; otherwise, they claim that their analysis is far too slow to process real applications.

Lack of precise data flow information limits the tools that use it to ensure the correctness of program transformations. For example, an optimizing compiler cannot perform aggressive optimizations such as interprocedural constant propagation [CCKT86] and interprocedural register allocation [SO90]. A parallel programming tool cannot aid users as effectively in parallelizing their programs because of too many spurious data dependences [HHLS90]. Furthermore, without such information a debugger cannot focus one's attention on a reasonably small portion of code where errors may exist, especially when multiple procedures are involved [Wei84].

This problem, in principle, could be alleviated by use of parallel data flow algorithms. Although parallel machines are increasingly available, current algorithms for data flow analysis are sequential. Recently, some researchers have designed parallel data flow algorithms [Zob90, LMR90, GPS90, KGS91]; preliminary empirical results were reported in [Zob90], [LMR91] and [LR92]. Although these results are encouraging, no one has proposed a comprehensive approach to performing data flow analysis in parallel. The formulation of the following problem statement is a starting point of such an investigation:

> *Given a set of data flow problems*[1] *to be solved for program Q on parallel machine P, what is the best parallel execution time achievable?*

To answer this question, we need to consider at least the following issues: (1) interdependence among these data flow problems, (2) algorithms for solving these problems, (3) characteristics of the program (or the flow graphs representing the program), and (4) attributes of the parallel machine chosen. The first three issues are related to the maximal parallelism inherent in a particular instance of the above problem. The last will decide the suitable task granularity at which we can effectively exploit useful parallelism.

[1] For example, an optimizing compiler normally solves several data flow problems in order to perform different optimizations.

In this paper, we report our approach to solving the above problem. We identify and classify three types of parallelism existing in the data flow solution process, and present a comprehensive approach to exploit them based on a unified framework. We also review three families of general-purpose algorithms for data flow analysis and discuss their parallelization. In particular, we use the design and implementation of our parallel hybrid algorithms to demonstrate the exploitation of algorithmic parallelism. To study the flow graph characteristics of real programs, we use Fortran procedures from the *Perfect Benchmarks* and the *netlib* libraries for this investigation.

We expect our approach to allow additional and more precise static analyses to be obtained in shorter analysis time. Because compilers and debuggers are among the most frequently used software tools, shorter response times can be an important factor for increasing user productivity. Furthermore, with more varied and precise data flow information these systems can more effectively perform their functions.

1.2 Applications

Performing data flow analysis in parallel can benefit any system needing data flow information. Our work is aimed at facilitating efficient static analysis on parallel machines; thus we are adding to the body of work addressing issues of compilation performed in parallel [SWJ+88, GZZ89, Gaf90, BS90]. Interprocedural data flow analysis is rarely performed currently in sequential compilers, in part because of its expected cost. By speeding up the data flow solution process, we can compute more precise interprocedural data flow information, previously too time-consuming to calculate, and thus enable an optimizing compiler to perform more aggressive optimizations.

Parallelizing compilers [ABC+88, Pol88, ZBG88, Wol89], like conventional optimizing compilers, collect data flow information for a source program and use this information to detect potential parallelism, determine an appropriate grain size, and then transform the program into a functionally equivalent program with a higher degree of parallelism. They can take hours to compile programs of reasonable size [GZZ89]. Although the parallel code generated may be quite efficient, these long compilation times must be shortened to make the use of these compilers practical. Since a large amount of compilation time is spent in computing (especially interprocedural) data flow information, our work in parallel data flow analysis can be seen as enhancing the development of *parallel* parallelizing compilers. Although the effectiveness of parallelizing compilers is arguable, it is beyond doubt that their success relies on precise data flow information.

The detection of parallelism is not the main issue in compiling explicitly parallel programs; nevertheless, compilers for such languages need even more precise and larger amount of data flow information to perform optimizations and further concurrentization [CK88, MP90]. These optimizations are normally aimed at reducing nonlocal data accesses in computer systems with distributed memories [HKT91] or hierarchical shared memories [GV91].

Furthermore, there are interactive parallel program development tools, such as PTOOL [HHLS90]

and ParaScope [CCH+88], designed to help the user parallelize sequential programs and debug parallel programs. It is required that these tools have as precise data flow information as possible, in order to assist the user in parallelizing, debugging, and testing programs. Performing data flow analysis in parallel has two advantages in this setting: (1) computing more precise information by allowing more analyses, and (2) shortening these tools' response time to a user's query.

1.3 Outline

The rest of the paper is organized as follows. Section 2 introduces data flow analysis and describes our machine and computation models for parallel data flow analysis. Section 3 categorizes three types of parallelism in the data flow solution process and discusses our exploitation of them. Section 4 discusses the design of parallel hybrid algorithms. Section 5 gives typical Fortran program characteristics that reveal embedded parallelism with respect to data flow analysis. Our implementation of the parallel hybrid algorithm for Reaching Definitions and empirical results are described in Section 6. Finally, Section 7 presents our conclusions.

2 Preliminaries

2.1 Graph-Theoretic Terminology

A directed graph $G = (N, E)$ consists of a finite set N of nodes and a set E of edges. An edge in E is an ordered pair (u, v), where $u, v \in N$; u is an *immediate predecessor* of v and v is an *immediate successor* of u. A path of length k from u to w is a sequence of nodes $p = (u = v_0, v_1, \ldots, v_k = w)$ such that $(v_i, v_{i+1}) \in E$ for $0 \leq i < k$. The path is a *cycle* if $v_0 = v_k$. A graph is *acyclic* if it contains no cycles.

A *flow graph* $G = (N, E, \rho)$ is a rooted directed graph with the unique root ρ such that for any node $v \in N$ there is a path from ρ to v.

A graph $G_1 = (N_1, E_1)$ is a subgraph of $G = (N, E)$ if $N_1 \subseteq N$ and $E_1 \subseteq E$ ($E_1 \subseteq N_1 \times N_1$). Let $\pi = \{N_1, N_2, \ldots, N_k\}$ be a partition of N; that is, $N_1, N_2, \ldots, N_k$ are mutually disjoint and are a covering of N. Then the *graph condensation* of G with respect to π generates a *condensed graph* $G' = (N', E')$ such that

- $N' = \{w_1, w_2, \ldots, w_k\}$, where w_i represents the subgraph induced by N_i for $1 \leq i \leq k$, and
- $(w_i, w_j) \in E'$ if and only if there are $u \in N_i$ and $v \in N_j$ such that $(u, v) \in E$.

Let $G = (N, E, \rho)$ be a flow graph, let $N_1 \subseteq N$, let $E_1 \subseteq E$, and let $h \in N_1$. $R_h = (N_1, E_1, h)$ is called a *region* of G with *head node* h if and only if in every path $(v_1, \ldots, v_k)$, where $v_1 = \rho$ and $v_k \in V_1$, there is some $i \leq k$ such that

- $v_i = h$,

- $v_{i+1}, \ldots, v_k$ are in N_1, and
- $(v_i, v_{i+1}), (v_{i+1}, v_{i+2}), \ldots, (v_{k-1}, v_k)$ are in E_1.

That is, access to every *region internal node* in $N_1 - \{h\}$ must go through the region head node; every region is single-entry [Hec77].

2.2 Data Flow Analysis

Data flow analysis is a process of collecting information about the way variables 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 (i.e., traversable on some program execution). Barth terms this assumption *precise up to symbolic execution* [Bar78].

More formally, an instance of a *monotone data flow framework* is specified by a tuple, $D =< G, L, F, M, \eta >$, where G is a flow graph (N, E, ρ), L is usually a meet semilattice, F is a space of monotone functions mapping L into L, M is a mapping of edges E of G into F, and η is an element of L [Hec77, MR91]. Intuitively, L is a lattice of data flow solutions, M the assignment of transition functions to nodes or edges, and η the entry solution at ρ; F is usually not represented explicitly. Often M can be thought of as specifying a system of $|N|$ or $|E|$ equations [RP86].

2.2.1 Program Representation

A program consists of one or more procedures; it is specified by a pair, $Q = (S, A_1)$, where S is a set of procedures $\{A_1, A_2, \ldots, A_b\}$, and A_1 is the main procedure. Data flow analysis algorithms use a directed graph representation of the program. In *intraprocedural* analysis, a procedure is abstracted as a *control flow graph*, in which a node represents a *basic block*,[2] 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 graph*, in which a node represents a procedure and an edge, a possible procedure call [Hec77]. Note that call graphs are multigraphs, for a procedure can call itself or another procedure more than once. Consequently, the number of edges in a call graph is the number of call sites in its corresponding program.

In the following discussions, we will use the term *flow graph* to mean either control flow graph (intraprocedural flow graph) or call graph (interprocedural flow graph). We assume the flow graphs for a program have been generated before we begin to solve data flow problems.

A flow graph is *irreducible* if and only if it contains a subgraph as shown in Figure 1, and is *reducible* otherwise [Hec77]. The root ρ and nodes a, b and c are distinct except that ρ and a may be the same; there are node-disjoint paths among them as the figure indicates. Intuitively, an irreducible flow graph contains a cycle with multiple entry nodes. Nodes b and c in the figure are entry nodes of a cycle. By contrast, a reducible flow graph consists of a nested set of single-entry

[2] In some analyses, such as program dependence analysis, a node represents a statement in the program.

regions. It can be reduced by some graph transformations into the trivial graph comprising one single node and no edge.

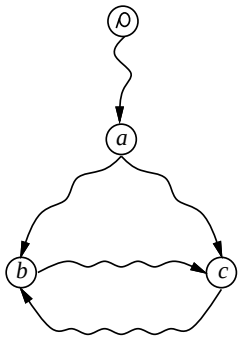


Figure 1: The paradigm irreducible flow graph.

2.2.2 Intraprocedural and Interprocedural Analysis

A data flow problem is either *intraprocedural* or *interprocedural*. *Forward* intraprocedural problems include Reaching Definitions (REACH), Available Expressions (AVAIL), and Constant Propagation (CONST). (These problems are described below.) They are called forward problems because the data flow information is propagated forward along edges in the control flow graph. *Backward* intraprocedural problems include Live Uses of Variables (LIVE) and Very Busy Expressions (VBUSY). For these problems, data flow information is propagated in an opposite direction to that specified by the control flow graph edges. Solutions to intraprocedural problems are used by optimizing compilers for safe program transformations such as copy propagation, common subexpression elimination and dead-code elimination [ASU86].

A downward exposed definition d of variable x at control flow graph node v *reaches* the top of node w, if there is a path from v to w such that x is not redefined along the path. Such a path is called a *definition-clear* path for x. REACH is the data flow problem asking what definitions reach the top of a node. An expression $x + y$ is *available* at the top of control flow graph node v if every path from the control flow graph root to v evaluates $x + y$, and after the last such evaluation prior to reaching v, there are no subsequent definitions of x or y. Finding the expressions available at the top of a node is the solution to AVAIL. A variable x is a *compile-time constant* at the top of control flow graph node v, if every path from the root to v will evaluate x (at compile-time) to be the same constant value. CONST asks what variables have what constant values incoming to a node.

An upward exposed use u of variable x at control flow graph node w is *live* at the bottom of node v, if there is a definition-clear path for x from v to w. Finding the live uses at the bottom

of a node is the solution to LIVE. An expression $x + y$ is *very busy* at the bottom of control flow graph node v, if no matter what path is taken from v, the expression will be evaluated before x or y is redefined. VBUSY asks what expressions are very busy at the bottom of a node.

To obtain more precise solutions for intraprocedural problems, we must account for the *side effects* of procedure calls. The interprocedural side effect analysis problems include MOD, REF, KILL and USE. They are solved on the call graph.[3] MOD asks what variables *may be modified* (or defined) by the execution of a statement [Ban79, CK84, Bur90]. REF asks what variables *may* be used (or *referenced*) by the execution of a statement [Bar78, CKT86b, CK87, Bur90].[4] KILL asks what variables *must* be defined (or *killed*) by the execution of a statement [Cal88]. USE asks what variables *may be used* (or referenced) before being redefined by the execution of a statement [Cal88]. All these problems are complicated by chains of procedure calls.

MOD and REF are flow-insensitive; KILL and USE are flow-sensitive. Solving *flow-insensitive* problems does not require the knowledge of control flow within procedures, whereas solving *flow-sensitive* problems does. Thus, we need more complex algorithms in both time and space to solve flow-sensitive problems.

2.2.3 Solution Procedures

Intuitively, a data flow problem can be represented by a system of equations; its solution is the desired data flow information. There are three families of general-purpose solution procedures for data flow analysis: iterative, elimination and hybrid.

An *iterative* algorithm is based on fixed-point iteration. It starts with a safe, initial solution, and then proceeds to get the maximum fixed point for the equations [Hec77]. Although iterative algorithms do not rely on the flow graph being reducible, they are normally not as efficient as elimination or hybrid algorithms.

An *elimination* algorithm has two phases and is conceptually similar to Gaussian elimination [RP86]. Most elimination algorithms require the flow graph to be reducible.[5] In the *elimination* phase, the flow graph is partitioned into intervals (or regions), data flow information local to an interval is summarized, and an interval is condensed into a node. The process continues until there is only one node left. The data flow problem is then easily solved on this single node. In the *propagation* phase, the algorithm proceeds in the reverse direction of the elimination phase. A node is expanded into an interval, and global data flow information is propagated into nodes within the interval.

A *hybrid* algorithm uses the strongly connected component decomposition of the flow graph, and combines aspects of both iterative and elimination algorithms. Intuitively, it is iterative within

[3] Similarly, backward interprocedural problems are solved on the reverse call graph.

[4] However, Cooper *et al.* use USE to denote our REF [CKT86b].

[5] The method of Graham and Wegman can handle irreducible flow graphs [GW76]. In practice, node splitting can be used to transform an irreducible flow graph into an equivalent, reducible one [Hec77].

components and elimination-like in its propagation on the strong component condensation. It can handle irreducible flow graphs by application of fixed-point iteration within components [MR90].

In addition to the above general-purpose algorithms, there is a family of data flow solution procedures called the *partitioned variable technique (PVT)* [Zad84]. A PVT algorithm partitions a data flow problem by variables and finds the relevant information for a single variable, one at one time. The technique is only applicable to a small set of intraprocedural problems, and is not general-purpose.

2.3 Parallel Machines

A parallel machine usually comprises tens to thousands of processors. It can be classified as a SIMD (single-instruction multiple-data) or MIMD (multiple-instruction multiple-data) machine. Examples of SIMD machines include the Connection Machine CM-2 and the ATM DAP 500. All the processors of a SIMD machine synchronously execute a single stream of instructions while each processor manipulates different data. SIMD machines naturally support the data parallel programming model, which users often find easier in developing parallel programs. However, the synchrony property can result in low utilization of processors. This may explain why the most recent model of the Connection Machine, CM-5, is a MIMD machine.

There are basically two families of MIMD machines: shared-memory machines and distributed-memory machines. In shared-memory machines, such as the Encore Multimax and the Sequent Symmetry, processors exchange information by access to the same memory locations. Synchronization and memory latency are main issues for these machines. In distributed-memory machines, such as the Intel iPSC/2 and the NCUBE/10, processors with their own local memories exchange information by explicitly sending and receiving messages. The overhead of message passing is a main issue for these machines.

It has been observed [van89] that the distinction between these two families of MIMD machines is disappearing. For shared-memory machines, mechanisms such as a well-organized hierarchical memory system are introduced to hide the latency of memory access. In the future, synchronization may be handled more efficiently and combined with the memory access. For distributed-memory machines, better message routing schemes are being applied to reduce the message-passing overhead. With this trend, we can expect distributed-memory machines to support finer-grained computation in the future.

The single global, shared memory space on a shared-memory machine provides a clearer programming model for MIMD machines. However, employment of a hierarchical memory system still will not allow shared-memory machines to scale up satisfactorily. By contrast, distributed-memory machines can more easily scale up, but are much harder for users to program. At the *ACM/IEEE Supercomputing'91 Conference*, Alliant Computer Systems introduced the Campus/800, which has 800 processors. They claimed that the Campus/800 is the first system to support both distributed

and shared memory. They expected the system made it easier to program and more efficient for real-world applications [Mye92]; however, recent economic developments have dimmed the hopes for this system.

2.4 Computation Model

We choose MIMD machines to be our machine model, because they are more appropriate for us to discover massive parallelism in data flow analysis. A parallel machine $P = \{P_1, P_2, \ldots, P_p\}$ consists of p processors. Our computation model is based on Hoare's Communicating Sequential Processes (CSP) [Hoa78]. We will formulate the parallel execution of data flow analysis as a *task system*. Intuitively, a task system is a parallel composition of asynchronous communicating tasks. A task is executed sequentially and will run to completion without interruption. This asynchrony property more closely reflects execution of processes on MIMD machines. Tasks communicate with one another in order to exchange information and enforce synchronization. For the sake of efficiency in our application domain, we require that communication and synchronization be performed only at the entry and/or the exit of a task; we do not want any task to be blocked during its execution because of communication or synchronization. In the literature, the computation model is also called the macro-dataflow model [Sar89].

We specify a task system with a directed acyclic *task dependence graph* $G = (T, E)$, where nodes $T = \{t_1, t_2, \ldots, t_n\}$ represent all the tasks in the system, and edges E represent precedence constraints among tasks: the directed edge $e_{ij} = (t_i, t_j)$ is in E if and only if t_j cannot start its execution until t_i completes because of synchronization or communication. Each task t_i has computation cost $c(t_i)$. Each edge e_{ij} has synchronization or potential communication cost $c(e_{ij})$; the communication cost exists when t_i and t_j are executed on different processors.

The cost of a path $p = (t_i, t_j, \ldots, t_k)$ is defined by $c(p) = c(t_i) + c(e_{ij}) + c(t_j) + \ldots + c(t_k)$; that is, all the computation and communication/synchronization costs on the path. A path p is a *critical path* if $c(p)$ is the maximum for all the paths; the critical path length of the task system is therefore $c(p)$. If there were an unbounded number of processors, the critical path length would be the parallel execution time of the task system.

3 Parallelism in Data Flow Analysis

Our problem statement, as stated previously, is:

> *Given a set of data flow problems $D_1, D_2, \ldots, D_d$ to be solved for program Q on parallel machine P, what is the best parallel execution time achievable?*

To answer this question, our first effort is to identify the three sources of parallelism in the data flow solution process: independent-problem parallelism, separate-unit parallelism and algorithmic parallelism. To effectively exploit these sources of parallelism, we propose a unified framework,

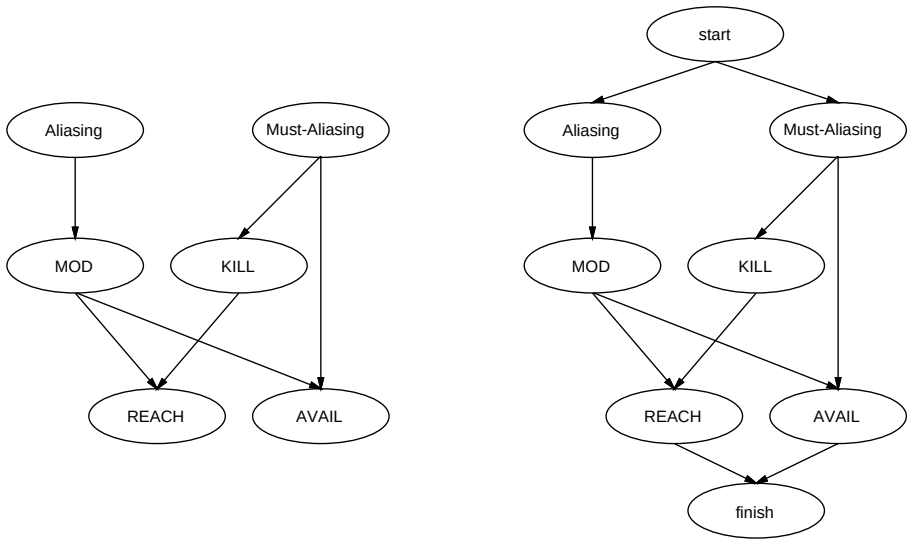
which is based on task refinement for a system of tasks which together solve the data flow problems. We present our design of a parallel data flow analyzer to illustrate this idea in Section 3.4.

3.1 Independent-Problem Parallelism

Independent-problem parallelism occurs when multiple independent data flow problems are to be solved. Two data flow problems are *independent* if and only if determination of the solution of one problem does not depend on the solution of the other. For example, REACH and AVAIL are independent, so we can solve them in parallel.

We use the precision of computed data flow information to formalize the notion of problem dependence. A data flow problem D_1 is dependent on another problem D_2 if and only if our solving D_1 without using the solution of D_2 results in less precise data flow information for D_1. For example, if we were to solve REACH without MOD information, we would need to assume that any procedure call can modify all global variables and all actual parameters at a call site. The REACH information computed will be much less precise, and although correct, it may not be useful in practice. Consequently, REACH is dependent on MOD.

To effectively capture the dependence relationships among data flow problems, we use *problem dependence graphs*. Assume we want to compute precise REACH and AVAIL information for Fortran programs. Then we can have the problem dependence graph as shown in Figure 2.



(a). A problem dependence graph

(b). A task dependence graph

Figure 2: A problem dependence graph and its corresponding task system (TDG_1).

Two variables are *aliases* when they both refer to a common memory location at the same time during execution. For a procedure, *Aliasing* asks for the set of variable pairs (x, y) such that x and

y *may* be aliases in *some* invocation of that procedure[6] [Ban79, CK89]. *Must-Aliasing* asks for the set of variable pairs (x, y) such that x and y *must* be aliases in *all* invocations of that procedure.

MOD is dependent on Aliasing, and KILL on Must-Aliasing. REACH is dependent on MOD, KILL and Aliasing. MOD information determines the creation of definitions in a statement. KILL information determines what variables must be redefined in a statement. The dependence relationship is transitive: REACH is dependent on Aliasing because REACH depends on MOD and MOD depends on Aliasing.

Similarly, AVAIL is dependent on MOD and Must-Aliasing. MOD information determines what expressions are no longer available after the execution of a statement since variables in expressions may have been redefined. Must-Aliasing information at an expression-generating statement allows us to conclude that other expressions are also generated because of aliases in a statement where an expression is generated explicitly.

Now to compute REACH and AVAIL information, we can create one task for solving each problem. The edges in the task system are defined by the edges in the problem dependence graph; therefore, the precedence constraints among tasks exactly reflect the dependence relationships among problems. Consequently, we have the task system TDG_1 in Figure 2, where we use the names of data flow problems for their corresponding tasks. Two dummy tasks `start` and `finish` of no cost are added to make a critical path always begin at `start` and end at `finish`.

The well-studied decomposition of MOD is another example to illustrate problem dependence. Burke's decomposition is shown in Figure 3 [Bur90]. Banning first isolated the effect of aliases on the problem from other effects [Ban79]. To improve the efficiency of Banning's technique, Cooper and Kennedy further separated the other effects into two parts: one is due to global variables and the other is due to formal parameters in the program [CK84]. (Interested readers should consult the three papers cited here for details.)

To better exploit independent-problem parallelism, we can compile a comprehensive set of data flow problems and construct the *complete* problem dependence graph for them. When solving data flow problems $D_1, D_2, \ldots, D_d$, we can extract the *tailored* problem dependence graph for these problems from the complete problem dependence graph. Once the tailored problem dependence graph is extracted, its corresponding task system can be automatically generated. With enough processors available, the task system will fully exploit independent-problem parallelism. The parallel execution time of the task system is therefore the critical path length of the task dependence graph.

[6] In Fortran programs, a variable is either local or global, and there are no pointer variables. This simplifies the problem of aliasing, for two variables are either aliased or not throughout execution of a procedure.

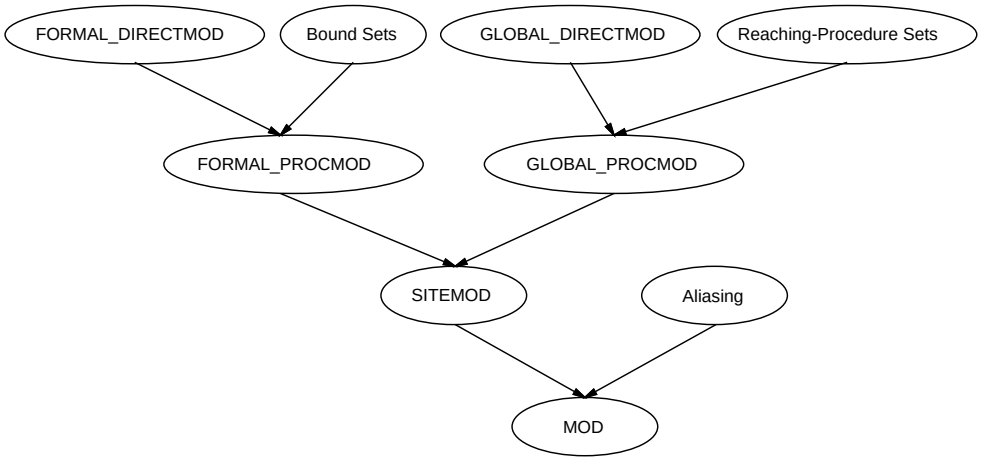


Figure 3: The MOD decomposition.

3.2 Separate-Unit Parallelism

Separate-unit parallelism exists in intraprocedural data flow analysis for any multiple-procedure program. Data flow information for each procedure can be computed in parallel and independently.[7] Since there is only one call graph for a program, such parallelism does not exist when we solve an interprocedural problem.[8]

We can exploit separate-unit parallelism to refine an intraprocedural problem task in Figure 2 (i.e., REACH or AVAIL). Assume the program has b procedures. Then we create b tasks for solving an intraprocedural problem: one task for each procedure. The refined task system of TDG_1 is shown in Figure 4 and is denoted by TDG_2. We use tasks $t_{r,1}, t_{r,2}, \ldots, t_{r,b}$ to refine the REACH task in TDG_1. Similarly, tasks $t_{a,1}, t_{a,2}, \ldots, t_{a,b}$ refine the AVAIL task. Since a critical path of TDG_1 contains the REACH or the AVAIL task, this refinement results in shorter critical path length in TDG_2. Thus, the parallel execution time of TDG_2 is shorter than that of TDG_1 if there are enough processors.

Naive parallel execution of data flow analysis exploits only separate-unit parallelism. It has the following disadvantages: (1) it does not exploit independent-problem parallelism; (2) the parallel execution time for solving an intraprocedural problem is dictated by the "most complicated" (usually the largest) procedure; (3) no speedup can be gained for solving interprocedural problems. In order to do better, we need to exploit the other two types of parallelism as well.

[7] Other levels of separate-unit parallelism (e.g., modules or basic blocks) may be appropriate for other data flow problems.

[8] We have assumed that all the nodes in a flow graph can be reached from the root. In practice, a call graph may consist of unconnected subgraphs, as in a subroutine library. This allows us to exploit separate-unit parallelism in interprocedural analysis too.

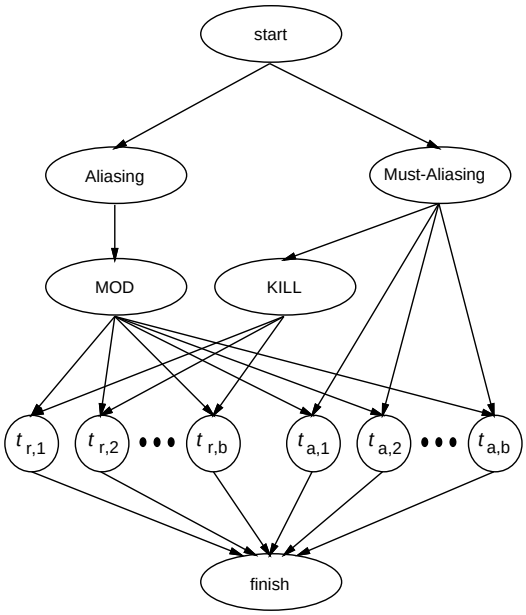


Figure 4: A refined task system (TDG_2) that exploits separate-unit parallelism.

3.3 Algorithmic Parallelism

Algorithmic (or solution-procedure) parallelism refers to the parallelism inherent in the solution procedures for data flow analysis. We can use it to refine tasks for solving intraprocedural problems on large procedures and solving interprocedural problems.

Recall a data flow problem can be represented by a system of equations, usually one for each flow graph node. An iterative algorithm visits flow graph nodes applying the corresponding equations until there is no change in solution. A straightforward parallelization of the algorithm can assign each node to one processor and visit *all* nodes simultaneously. Then a `gather` operation can be used, to check if there is any change in the solution for any node. Although iterative algorithms are simple in implementation, they do not allow as effective parallelization as elimination and hybrid algorithms.

Straightforward parallelization of an elimination algorithm can simultaneously summarize (and propagate) data flow information in independent intervals; however, this approach offers no control over the flow graph partitioning [Zob90]. For improving performance, we can partition a flow graph into regions instead of intervals [Lee92]. We can refine a large interval into smaller regions to improve parallelism and load balancing, and cluster small intervals into a larger region to improve load balancing. Too small intervals usually offer parallelism that is too fine-grained to be effectively exploited in practice.

Hybrid algorithms are amenable to parallelization too. We will discuss our design of parallel

hybrid algorithms in more detail later (see Section 4.1).

3.4 Putting All Together

In the previous sections, we have progressively refined tasks in order to reveal a higher degree of parallelism inherent in the data flow solution process. Here we present a top-down, comprehensive approach to effectively exploit the available parallelism. Our approach is based on a *unified framework* $PD = (G, R)$, where G is a (hierarchical) task dependence graph and R is a collection of task refinement rules. A task $t \in G$ can be refined according to some task refinement rule $r \in R$. In the following, we will use the design of a parallel data flow analyzer to illustrate the approach.

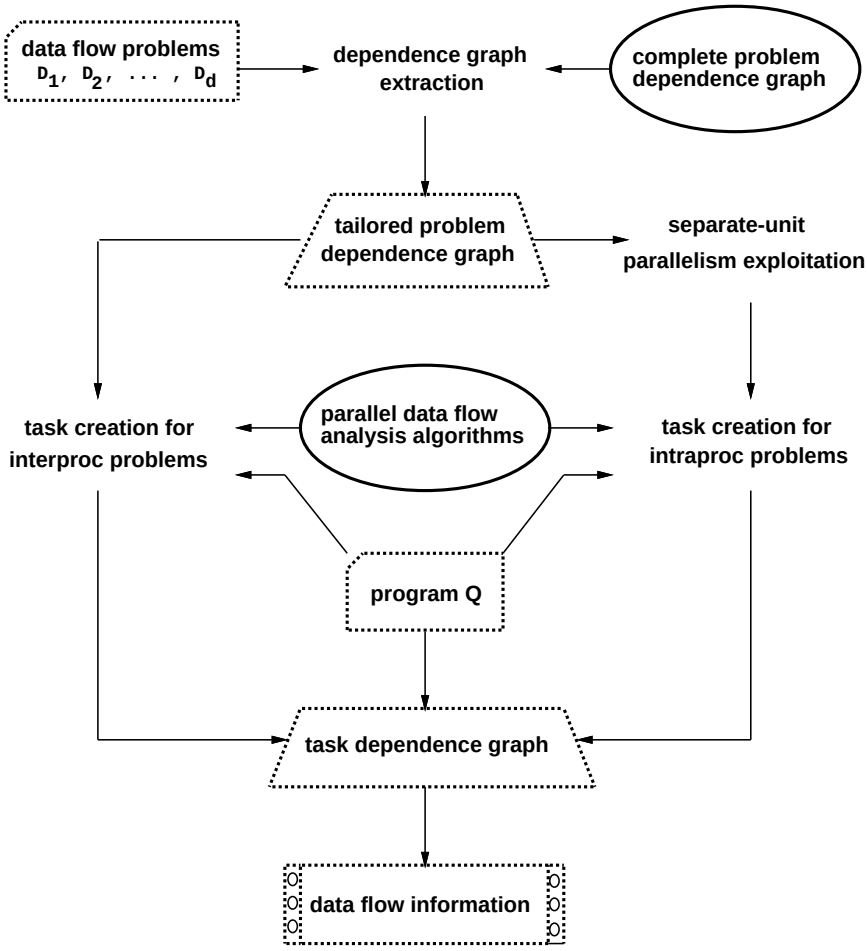


Figure 5: A parallel data flow analyzer.

The structure of our parallel data flow analyzer is shown in Figure 5. There are two repositories:

one for the complete problem dependence graph, and the other for various parallel algorithms for data flow analysis. An input instance includes a program Q and some data flow problems $D_1, D_2, \ldots, D_d$ to be solved for the program. To exploit independent-problem parallelism, the parallel analyzer first extracts a tailored problem dependence graph from the complete problem dependence graph. Note that data flow (sub)problems can be generated in the process although they are not specified in the input.

With the tailored problem dependence graph, our parallel analyzer exploits separate-unit parallelism to refine any task for an intraprocedural problem into b tasks, one for each procedure. At the same time, it exploits algorithmic parallelism in solving an interprocedural problem; it selects a suitable parallel algorithm for the interprocedural problem based on call graph characteristics.[9] Once the parallel analyzer has created a task for solving an intraprocedural problem on one procedure, it can further exploit algorithmic parallelism to refine this task. Based on the control flow graph characteristics, it chooses an appropriate parallel algorithm to perform the intraprocedural analysis on the procedure.

The advantage of our comprehensive approach to parallel data flow analysis is that we can flexibly refine tasks according to the parallel machine used. We can exploit the finest-grained parallelism, but we may choose not to do so in practice. The suitable grain size can be determined by such factors as the number of currently available processors and the overhead of communication and synchronization. For example, if we are to perform data flow analysis for a program with 20 procedures on a parallel machine with 10 processors currently available, we may exploit independent-problem parallelism, algorithmic parallelism in solving any interprocedural problems, and separate-unit parallelism, but may choose not to exploit algorithmic parallelism in solving any intraprocedural problems, because the exploitation of coarse-grained parallelism is sufficient to keep these processors busy.

Since we probably will not exploit the finest-grained parallelism most of the time, it is important to know how to make the above choices in practice. Only after we build a prototype of the parallel data flow analyzer and experiment with different combinations of input programs and data flow problems, can we gain experience and understanding of the issues.

4 Parallel Hybrid Algorithms

4.1 Design

We have designed a family of parallel hybrid algorithms for data flow analysis [LMR91] based on the general-purpose, hybrid algorithms [MR90]. The phases of a sequential hybrid algorithm are:

[9] A flow-sensitive problem normally contains a subproblem of considering control flow within procedures. We can also exploit separate-unit parallelism in solving the subproblem.

1. *Flow graph condensation.* Construct the flow graph and find its strongly connected components (in short, strong components or components). Find a topological order for the condensed flow graph, which is the strong component condensation of the flow graph and whose nodes represent regions of the flow graph. (Form improper regions to encapsulate irreducibilities for any strong components that have two or more entry nodes.)

2. *Problem setup.* Determine local information (such as variables, definitions, and functions) and set up the global problem lattice. In each region, determine the appropriate instances of the *restricted* and *representative* problems, and set up local lattices.

3. *Local Solution.* In each region, iterate the representative and restricted problems to solutions.

4. *Global Propagation.* In a topological order, propagate the global data flow information on the condensed flow graph. For each region, combine the local solutions with incoming global information to the region head node to obtain global information on region exit edges.

5. *Local Propagation.* Construct global data flow information for every node in each region.

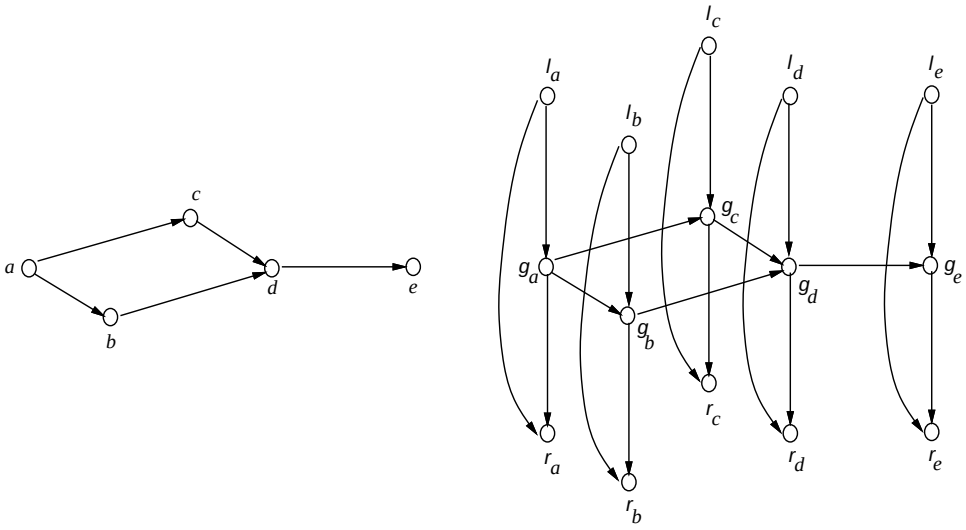
For the parallel execution of hybrid algorithms, we designate a master task to do initialization: flow graph condensation and mapping of tasks onto processors.[10] In addition, each region of the flow graph induces three worker tasks, corresponding to the last three phases of the hybrid algorithm. These tasks include a local-solution, a global-propagation and a local-propagation task. All the worker tasks, subject to precedence constraints, are to be executed in parallel.

Figure 6 illustrates a condensed flow graph and its induced worker tasks, where l_i are local-solution tasks, g_i are global-propagation tasks, and r_i are local-propagation tasks. The edges in the induced task dependence graph specify the precedence constraints among tasks. In general, the condensed flow graph specifies the dependences among global-propagation tasks. Local-solution tasks for different regions are independent of each other, so are local-propagation tasks. Although any edge from l_i to its corresponding r_i is not needed to enforce precedence constraints, we include it to indicate the required communication when l_i and r_i are assigned to different processors. Furthermore, any critical path for an induced task dependence graph consists of one local-solution task, followed by one or more global-propagation tasks, followed by one local-propagation task.

Our design and implementation of parallel hybrid algorithms are targeted for distributed-memory machines, on which the communication costs for passing messages between processors have significant effect on performance. We believe that our work can migrate to shared-memory machines with little difficulty once we have fully understood the design and implementation issues on distributed-memory machines (e.g., mapping and scheduling).

The goal of mapping and scheduling is to achieve the shortest completion time for a parallel algorithm. Mapping assigns tasks to processors. Scheduling defines a total ordering on tasks.

[10] Problem setup is partially done by the master task and partially done by local-solution tasks.



(a). A condensed flow graph. (b). The induced task dependence graph.

Figure 6: A condensed flow graph and its induced task dependence graph.

Since mapping has strong interaction with scheduling when we want to achieve the goal, it is usually viewed as a subproblem of scheduling. Several scheduling problems subject to precedence constraints have been shown to be $\mathcal{NP}$-complete [LK78]. The usual approach to scheduling is to devise a set of heuristics to achieve a suboptimal but good schedule with a reasonable amount of effort. We propose two heuristics for scheduling our task systems.

Mapping heuristic M1. The three tasks induced by a flow graph region n are mapped onto the same processor. This heuristic is aimed at reducing the communication costs from l_n to g_n, from l_n to r_n, and from g_n to r_n. More importantly, the data structures shared by the three tasks (e.g., the internal flow structure of a region) do not need to be stored in duplicate on different processors as when the tasks are not assigned to the same processor.

Scheduling heuristic S1. We prioritize global-propagation tasks when they are ready for execution over local-solution tasks, which are prioritized over local-propagation tasks. The justifications are as follows: (1) global-propagation tasks are more likely to be on a critical path than other tasks; (2) global-propagation tasks usually have two or more immediate successors whereas local-solution tasks have two and local-propagation tasks have none. Our intention is, on the one hand, to propagate global data flow information to the last executed global-propagation task(s) as quickly as possible and, on the other hand, to make more tasks ready.

Our approach to scheduling parallel hybrid algorithms is a combination of static mapping and dynamic scheduling. *Static mapping* is aimed at reducing run-time overhead and is performed before execution of the task system. *Dynamic scheduling* is aimed at maintaining better load balancing and is performed during execution of the task system. However, a *global* dynamic scheduler

can incur high communication overhead and degrade system performance on distributed-memory machines. Our decision is to have a *local* dynamic scheduler on each processor in place of the global dynamic scheduler. These local dynamic schedulers can take advantage of local-solution tasks to keep processors busy and, at the same time, enforce scheduling heuristic *S1* to make ready more global-propagation and local-propagation tasks.

4.2 Region Partition

Partitioning a flow graph into strong components can generate many small and single-node regions which induce tasks too fine-grained to be effectively exploited in practice. By clustering small regions into larger ones, we intend to reduce the following costs entailed by a parallel hybrid algorithm:

- dynamic scheduling cost, which is proportional to the number of regions since one region induces three tasks;
- global propagation cost, which is proportional to the number of regions in the worst case when all the global-propagation tasks are totally ordered;
- communication cost, which is potentially higher when there are more inter-region edges.

This optimization is very important for call graphs, because they tend to have large proportions of nodes outside any cycles.

Recall that a *region* is a connected subgraph of a flow graph such that all the incoming edges from other parts of the flow graph to the region enter into its *head node*. This clustering problem can also be viewed as that of partitioning a condensed flow graph into regions, each of which contains one or more previously smaller regions. More formally, we define the region partition problem as the following:

> **Definition (Region Partition).** Given a size limit $\mathcal{S} \in Z^+$, and an *acyclic* flow graph $G = (N, E, \rho)$, where each $v \in N$ has size $s(v) \in Z^+$, partition G into r regions $R_{h_i} = (N_i, E_i, h_i)$ with region size $s(R_{h_i}) = \sum_{v \in N_i} s(v) \leq \mathcal{S}$, $1 \leq i \leq r$, such that r is minimized.

There are two constraints in forming a region: (1) the *entry constraint*, which requires a region to have only one entry node; and (2) the *size constraint*, which requires the size of a region to be no larger than $\mathcal{S}$. We assume that $\mathcal{S}$ is always large enough to allow feasible solutions to the problem. For the purpose of optimizing parallel hybrid algorithms, the input graph of the region partition problem is the condensed flow graph. The size of a condensed flow graph node v is defined to be the size of the strong component (or region) that v represents.

In our formulation of the region partition problem, we want to minimize the number of regions whose sizes are constrained by a certain size limit. Since clustering small regions can decrease the

degree of parallelism in the induced task dependence graph, we should choose a size limit in order not to lose useful parallelism. This selection normally depends on the finest task granularity that can be effectively supported by the parallel machine.

The minimization objective in our region partition problem is intended to directly reduce dynamic scheduling cost, and is expected to reduce global propagation cost. For a specific flow graph, fewer regions are more likely to create fewer inter-region edges; therefore, communication and synchronization costs can be reduced too.

The region partition problem is in general provably $\mathcal{NP}$-hard [Lee92]. Because of the computational complexity, we attempted to find approximation algorithms that can do well at relatively small cost on flow graphs from real programs. Here we present our forward algorithm for region partition, which is based on the Allen-Cocke interval-finding algorithm [AC76].[11]

The algorithm is shown in Figure 7, where a region is simply represented by a set of nodes. Since it forms regions by proceeding along the direction of execution flow on flow graph edges, it is called the *forward algorithm*. The complexity of this algorithm is $O(n+e)$, where n is the number of nodes (i.e., $|N|$) and e is the number of edges (i.e., $|E|$). This algorithm is greedy: it makes a region as large as possible while not violating the size constraint. In so doing, we expect to have a smaller number of regions formed.

An example of region partition performed by the forward algorithm is given in Figure 8, where we assume the region size limit $\mathcal{S}$=4 and every node has the unit size. There are four regions of which the head nodes are 1, 5, 9 and 11, respectively. It is tempting for one to cluster node 11 with nodes 9 and 10, but the resulting subgraph induced by the node subset {9, 10, 11} has multiple entry nodes and is thus not a region.

In our formulation of the region partition problem, we assume that the size limit $\mathcal{S}$ is always large enough to admit feasible solutions. In practice, we can relax the size constraint as follows: A region can have its size larger than $\mathcal{S}$ only if it contains one single node. That is, we allow single-node regions of very large size while clustering regions of smaller size. We define this problem variant to be the *relaxed* region partition problem.

5 Characteristics of Typical Fortran Programs

The input to our parallel data flow analyzer is a set of data flow problems that dictate independent-problem parallelism and a program that determines separate-unit and algorithmic parallelism. In this section, we discuss our investigations into some relevant characteristics of Fortran programs that affect separate-unit and algorithmic parallelism: the number and size of procedures in the program. We also look at structural properties of flow graphs that influence their suitability for parallel hybrid algorithms.

[11] The other and more complicated bottom-up algorithm, can be found in [Lee92].

```
Input:  flow graph G and size limit S.
Output:  a list of regions RP that partitions G.
Data Structures:    1.  H a set of head nodes.
                    2.  R a single region.
                    3.  RP the current set of regions.
                    4.  pred(v) the set of immediate predecessors of v.

for all nodes v do included(v) = false;
/* no node is included in any region yet. */
RP = ∅;
H = {ρ}; /* root must be the head node of a region. */
while H ≠ ∅ do
     delete h from H;
     R = {h}; /* h is the head node of region R. */
     included(h) = true;
     size(R) = size(h);
     while ∃ v ((included(v) = false) and (pred(v) ⊆ R)
               and (size(R) + size(v) ≤ S)) do
          R = R ∪ {v};
          included(v) = true;
          size(R) = size(R) + size(v);
     endwhile
     RP = RP ∪ {R};
     while ∃ v (pred(v) ∩ R ≠ ∅ and included(v) = false) do
          H = H ∪ {v};
     endwhile
endwhile
```

Figure 7: The forward algorithm.

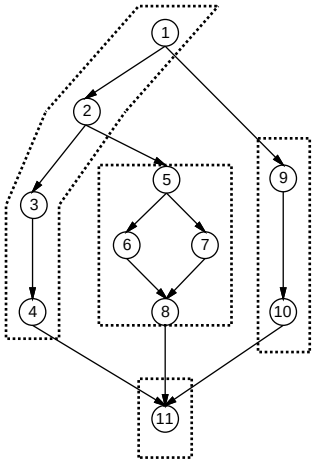


Figure 8: A partitioning due to the forward algorithm.

Flow Graph Size. We used the 13 Fortran programs from the *Perfect Benchmarks* as representatives [CKPK90]. Table 1 gives the size statistics of these programs. The size of a procedure is defined to be the number of nodes in its control flow graph and is denoted by n.

All the programs except `ti` and `lw` have a moderate to high degree of separate-unit parallelism. Since the number of procedures in a program is the number of nodes in its call graph, programs having a higher degree of separate-unit parallelism also potentially have a higher degree of algorithmic parallelism in the solution of an interprocedural problem. Assume we are to exploit algorithmic parallelism in solving an intraprocedural problem for procedures with $n \geq 20$. Then most programs have high percentages of such procedures for exploitation. Although these observations are neither general nor conclusive, they are indications that these two types of parallelism abound in many programs.

Incidentally, different program design and coding styles can affect the size of control flow graphs. For example, none of the 39 control flow graphs in program `sr` has more than 50 nodes. The designer of this program might have intended to make every procedure small so that it could be easily understood and maintained. By contrast, program `cs` has 22 control flow graphs of size more than 100 (the largest control flow graph has 380 nodes and 587 edges).

To evaluate parallel hybrid algorithms for *intraprocedural* data flow analysis, we additionally used five *netlib* libraries[12] [DG87]. These libraries are listed in Table 2. We chose procedures from both application programs and libraries because their procedures might have different characteristics.

A procedure is *Small* if and only if the size of its control flow graph is less than 20. Among all

[12] Among these netlib libraries, `eispack`, `fftpack`, `fishpack` and `linpack` are collections of library routines, but `paranoia` is a program.

Perfect Benchmarks					
Program	Number of Procedures of size n				
	Total	$n < 20$	$20 \leq n < 50$	$50 \leq n < 100$	$n \geq 100$
ap	97	72	20	4	1
cs	127	56	34	15	22
lg	35	25	6	3	1
lw	16	9	6	1	0
mt	32	21	7	4	0
na	42	18	16	6	2
oc	35	22	10	2	1
sd	77	66	9	2	0
sm	28	16	7	5	0
sr	39	27	12	0	0
tf	28	11	12	5	0
ti	7	4	2	1	0
ws	64	31	21	11	1

Table 1: Size of Fortran programs.

Netlib Libraries				
Library	Descriptions	Number of Procedures of size n		
		Total	$n < 20$	$n \geq 20$
eispack	eigenvalues and vectors	62	29	33
fftpack	Swarztrauber's Fourier transforms	14	10	4
fishpack	separable elliptic PDEs	97	38	59
linpack	Gaussian elimination, QR, SVD	50	16	34
paranoia	Kahan's floating point test	26	15	11

Table 2: Descriptions and size of Fortran libraries.

the 876 procedures studied, there were 486 Small procedures (i.e., 55%). No matter what algorithm is used, we do not expect to gain any significant speedup for solving intraprocedural problems on Small procedures. Consequently, the exploitation of separate-unit parallelism is very important to a program consisting of only smaller procedures. Table 4 lists the number of Small procedures found.

Size of Largest Component. Since parallel hybrid algorithms perform tasks for each component sequentially, their effectiveness depends on the size of the largest component in the flow graph, rather than only on the size of a flow graph.[13] The largest component size itself imposes a limit on the possible speedup obtainable. Thus, for a flow graph, we are interested in the percentage of nodes in its largest component.

To evaluate parallel hybrid algorithms for *interprocedural* data flow analysis, we used the 13 programs from the Perfect Benchmarks. All the call graphs for these programs are acyclic; thus, no large component exists to hinder the effectiveness of parallel hybrid algorithms.

For *intraprocedural* data flow analysis, the PLC (the percentage of nodes in its largest component) statistics for non-Small procedures in our sample are summarized in Table 3. The PLC can be as high as 98%, the value obtained for a procedure in program `cs` which mainly consists of one loop nest. In this case, parallel hybrid algorithms cannot gain any significant speedup at all. By contrast, procedures in program `paranoia` are quite suitable for parallel hybrid algorithms.

Suppose that each parallel hybrid algorithm performs the same amount of work on every flow graph node, and suppose that *the* critical path of any task system consists solely of the three tasks induced by the largest component. Then $\frac{100}{PLC}$ would be the greatest speedup of a parallel hybrid algorithm on a flow graph. Using this value as a rough estimate of speedup, then for non-Small procedures, parallel hybrid algorithms can gain:

- speedups more than 10 for 74 procedures (19%);
- speedups between 5 and 10 for 49 procedures (13%);
- speedups between 2 and 5 for 127 procedures (33%);
- speedups no more than 2 for 140 procedures (36%).

In general, the speedups for intraprocedural problems will not always increase with the size of control flow graph, because these algorithms are more sensitive to the *structure* of these graphs.

For a non-Small procedure, we say it is *Big-Comp* if and only if the largest strong component of its control flow graph contains at least 50% of nodes in the entire graph. There were 140 Big-Comp

[13] More precisely, the size of the largest *region* imposes a limit on the speedup of parallel hybrid algorithms. In a reducible flow graph, the largest region is exactly the largest component. If the largest component is not single-entry, we need to form an improper region to accommodate the irreducibility. In this case, the size of the improper region is larger than that of the component.

Library or Program	Subtotal	Number of CFG's with PLC=c				max c for all CFG's
		$c < 10$	$10 \leq c < 20$	$20 \leq c < 50$	$c \geq 50$	
eispack	33	0	0	5	28	97
fftpack	4	1	1	1	1	82
fishpack	59	22	4	17	16	90
linpack	34	2	4	21	7	89
paranoia	11	9	2	0	0	18
ap	25	8	1	4	12	88
cs	71	17	13	20	21	98
lg	10	0	0	3	7	97
lw	7	0	2	1	4	90
mt	11	0	0	6	5	77
na	24	4	3	10	7	89
oc	13	3	0	9	1	70
sd	11	1	2	5	3	91
sm	12	2	6	3	1	94
sr	12	4	3	1	4	92
tf	17	1	3	11	2	70
ti	3	0	0	1	2	56
ws	33	0	5	9	19	95
Total	390	74	49	127	140	xx
Percentage	100	19	13	33	36	xx

PLC (in %) = percentage of nodes in the largest strong component.

Table 3: Percentage of nodes in the largest strong components in non-Small procedures.

procedures (i.e., 36% of non-Small procedures). The speedup of any parallel hybrid algorithm for an *individual* Big-Comp procedure is normally at most two.

We classified all the 876 procedures into three categories: Small, Big-Comp and OK. A procedure is an *OK* procedure if and only if it is neither Small nor Big-Comp. We expected parallel hybrid algorithms to perform well for the 250 OK procedures (i.e., 29%).

In addition, we were also interested in an orthogonal classification of procedures, based on control flow graph reducibility, the lack of which might result in a more difficult initial flow graph decomposition. The results of such classifications are reported in Table 4. There are 22 Irreducible procedures (i.e., 3%), whose control flow graph sizes are as follows:

- 4 graphs have between 20 and 50 nodes.
- 7 graphs have between 50 and 100 nodes.
- 11 graphs have more than 100 nodes.

Only two out of 13 programs in the Perfect Benchmarks have Irreducible procedures. In contrast, three out of five libraries contain Irreducible procedures. It is, therefore, an important consideration whether a data flow analysis algorithm can handle irreducible flow graphs. Hybrid algorithms do not need to form improper regions to accommodate irreducibilities in 12 procedures, since these irreducibilities are embedded in single-entry strong components. For the remaining 10 Irreducible procedures, improper regions are needed.

Since the Fortran procedures studied are mainly for scientific computing, they usually consist of loops in which values of array elements are modified or calculated. Procedures for other applications or written in other languages such as C may have quite different characteristics.

6 Implementation and Findings

We have reported our preliminary experiments on the parallel hybrid algorithm for Reaching Definitions in [LMR91]. In this section, we give the empirical results of additional experiments profiling the performance of the *optimized* parallel hybrid algorithm (using relaxed region partition) on a larger test data set.

6.1 Experiments and Data Used

We implemented the parallel hybrid algorithm on the Intel iPSC/2 machine at the Cornell Theory Center. The iPSC/2 is a distributed-memory machine whose multiple processors are interconnected as a hypercube. We executed the master task on its host machine, and worker tasks on the hypercube processors. Our program is based on the SPMD (single-program multiple-data) programming model: We have the same code for all processors, but each of them manipulates a different set of data. In our case, the set of data for each processor is a subset of flow graph regions.

Library or Program	Number of Procedures					
	Subtotal	Small	Big-Comp	OK	Reducible	Irreducible
eispack	62	29	28	5	53	9
fftpack	14	10	1	3	14	0
fishpack	97	38	16	43	92	5
linpack	50	16	7	27	50	0
paranoia	26	15	0	11	24	2
ap	97	72	12	13	97	0
cs	127	56	21	50	122	5
lg	35	25	7	3	35	0
lw	16	9	4	3	15	1
mt	32	21	5	6	32	0
na	42	18	7	17	42	0
oc	35	22	1	12	35	0
sd	77	66	3	8	77	0
sm	28	16	1	11	28	0
sr	39	27	4	8	39	0
tf	28	11	2	15	28	0
ti	7	4	2	1	7	0
ws	64	31	19	14	64	0
Total	876	486	140	250	854	22
Percentage	100	55	16	29	97	3

Table 4: Classifications of Fortran procedures.

Our static mapping scheme uses a threshold value TH equal to the average number of flow graph nodes per available processor. A greedy algorithm assigns regions (and thus their induced tasks) to processors in a topological order so that each processor expects to take TH flow graph nodes. If this cannot be realized due to large regions, it assigns between $\frac{1}{2}TH$ and $\frac{3}{2}TH$ flow graph nodes whenever possible.[14] Thus, the cost for mapping is insignificant. We implemented a local scheduler for each processor to enforce our scheduling heuristic *S1*. Ready tasks on each processor are maintained in a priority queue; global-propagation tasks are prioritized over local-solution and local-propagation tasks.

In our experiments, we performed the parallel hybrid algorithm for each procedure on one, two, four and eight processors of the iPSC/2. We used a maximum of eight processors because most of the Fortran procedures tested rarely offered enough parallelism to effectively utilize more than 8 processors. In each case, the execution time reported was the execution time of the longest running hypercube processor.[15] We ran each experiment five times for each procedure; little variance in the millisecond timings was observed and mean times are reported. To validate the correctness of the results, we also implemented a sequential iterative algorithm for Reaching Definitions and compared solutions obtained.

We used 85 Fortran procedures from the Perfect Benchmarks and *netlib* libraries. Characteristics of their internal structure are reported in Table 5. We divided them into five groups in order to evaluate our region partition technique (see Section 6.2). For example, a graph in the third group has less than 75% and no less than 50% of nodes outside any cycles. Since the procedures in the first two groups were relatively small, fewer of these procedures were selected for our experiments.

Group	Characteristics	Number of procedures				
		Total	m $<$ 50	50 $\leq$ m $<$ 75	75 $\leq$ m $<$ 100	m $\geq$ 100
1	a = 100	10	7	3	0	0
2	75 $\leq$ a $<$ 100	15	6	4	2	3
3	50 $\leq$ a $<$ 75	20	4	4	6	6
4	25 $\leq$ a $<$ 50	20	6	5	3	6
5	a $<$ 25	20	7	5	2	6

m = number of nodes in control flow graph.
a = percentage of control flow graph nodes outside any cycles.

Table 5: Characteristics of data used (Fortran procedures).

[14] This general rule is violated when there is one very large nontrivial region and the number of available processors is more than one. For instance, consider the case where there are 4 processors and a control flow graph has 80 nodes and has a nontrivial region with 50 nodes. In this case, the TH value is 20, and 50 is greater than $\frac{3}{2}TH$.

[15] We did not account for execution time of the master task since it was executed sequentially.

6.2 Findings

We give the relative speedup results of our original, unoptimized parallel hybrid algorithm for Reaching Definitions in Table 6, under the heading *Without optimization*. The execution time of the parallel hybrid algorithm on multiple processors is compared with that on one processor. Let $T_{A,p}$ be the execution time of the parallel hybrid algorithm for procedure A on p processors, and let $S_{A,p} = \frac{T_{A,1}}{T_{A,p}}$ be the relative speedup. Also let n_g be the number of procedures in group g. Three measures of speedup are used for each group g on p processors (p is 2, 4 or 8):

- mean speedup: the mean of speedups for all the procedures in the group, or $\frac{\sum_A S_{A,p}}{n_g}$.
- gross speedup: ratio of the total execution time for all the procedures in the group on one processor to that on p processors, or $\frac{\sum_A T_{A,1}}{\sum_A T_{A,p}}$.
- best-case speedup: the largest speedup found for a procedure in the group, or $\max_A(S_{A,p})$.

The gross speedup measure can filter noises caused by our simple static mapping of tasks on a distributed memory machine; however, it is biased by results for large graphs. The mean speedup measure reports more typical results. We make the following observations:

- All the speedups increase as the number of processors increases. However, the marginal effect of using more processors declines, so parallel hybrid algorithms can obtain only moderate speedup for intraprocedural analysis of Fortran procedures.
- The speedups for group one are not as good as those for other groups. With a fixed number of processors, since the procedures in group one have smaller control flow graphs, each processor works on fewer nodes. As a result, the ratio of communication overhead to computation cost is relatively high.
- On two and four processors, we have several instances where the best-case speedups are more than two and four, respectively. We feel this is due to anomalies in dynamic allocation of memory (i.e., malloc) in our implementation. When multiple processors are used, we do not need to allocate as much memory on one processor as when only one processor is used, so we do not trigger possible garbage collection by the system.
- On eight processors, we have the best-case speedup of 7.6 in one instance. This indicates that we can achieve larger speedups on larger procedures or for more complicated data flow problems.
- Most gross speedups are better than their corresponding mean speedups. This also indicates that we usually can achieve larger speedups on larger procedures.

Measure	Group	Relative speedup					
		Without optimization			*With optimization*		
		Two	Four	Eight	Two	Four	Eight
Mean speedup	1	1.7	2.4	2.9	1.9	3.0	3.8
	2	1.8	2.6	3.5	2.0	3.3	4.5
	3	1.9	3.0	4.0	2.0	3.3	4.7
	4	1.8	3.0	4.1	1.9	3.0	4.3
	5	1.9	3.2	4.5	1.9	3.0	4.4
Gross speedup	1	1.7	2.4	2.8	1.9	2.9	3.8
	2	2.0	3.3	4.7	2.3	4.0	6.1
	3	1.8	3.0	4.2	1.9	3.4	5.1
	4	2.0	3.4	4.7	2.0	3.4	5.2
	5	2.0	3.0	5.2	1.9	3.3	4.9
Best-case speedup	1	1.8	2.8	4.1	2.2	3.8	4.6
	2	2.2	4.2	6.4	2.6	4.9	8.2
	3	2.1	3.7	4.9	2.3	4.1	6.5
	4	2.2	4.3	6.0	2.3	4.3	7.4
	5	2.2	4.4	7.6	2.1	4.0	7.5

Table 6: Relative speedup of a parallel hybrid algorithm.

In our experiments, we found that Fortran control flow graphs contain many small or single-node strong components. We have implemented the forward algorithm for relaxed region partition to optimize the parallel hybrid algorithm. To better illustrate the improvement by this optimization, we compare the execution time of our optimized parallel hybrid algorithm on multiple processors with that of the unoptimized parallel hybrid algorithm on one processor. These results are given in Table 6, under the heading *With optimization*. We observe improvements in most cases, and there are significant improvements for the first three groups.

For more fair comparison, we also give the relative speedup results for the optimized parallel hybrid algorithm in Table 7. This time we compare the execution time of the optimized parallel hybrid algorithm on multiple processors with that on one processor. Comparing these results with those under the heading labeled *Without optimization* in Table 6, we observe that our optimization has positive effect on the speedup results; that is, improvements by the optimization usually increase as the number of processors increases.

Although the Fortran procedures tested rarely offer enough moderate-size regions for us to utilize more processors, we did observe more significant speedups for some test data. For instance, the hybrid algorithm gained 10-fold and 20-fold relative speedup for one procedure on 16 and 32 processors, respectively.[16] Since call graphs usually do not contain large strong components, we expect parallel hybrid algorithms to have bigger payoff in interprocedural analysis. Furthermore,

[16]The procedure is in the SPICE program from the Perfect Benchmarks.

Measure	Group	Relative speedup		
		Two	Four	Eight
Mean speedup	1	1.7	2.6	3.4
	2	1.9	3.0	4.1
	3	1.9	3.1	4.4
	4	1.8	2.9	4.3
	5	1.9	3.0	4.3
Gross speedup	1	1.7	2.6	3.4
	2	2.1	3.6	5.5
	3	1.8	3.3	4.9
	4	2.0	3.4	5.2
	5	1.9	3.3	4.8
Best-case speedup	1	2.0	3.4	4.1
	2	2.3	4.3	7.1
	3	2.1	3.9	4.2
	4	2.2	4.5	7.8
	5	2.1	4.0	7.8

Table 7: Relative speedup of the optimized algorithm.

we can exploit separate-unit parallelism, which abounds in most programs.

In order to understand the effect of our optimization on different groups of procedures, we compare the optimized algorithm with the unoptimized one on a fixed number of processors. Table 8 reports the execution time improvement ratio in percent, under the heading labeled *Real Implementation*. Let $T^o_{A,p}$ be the execution time of the optimized algorithm for procedure A on p processors. We use three measures of improvement ratio for each group g of procedures:

- *Mean improvement*: the mean of improvements for all the procedures in the group, or $\frac{\sum_A \frac{(T_{A,p} - T^o_{A,p})}{T_{A,p}}}{n_g}$.
- *Gross improvement*: ratio of Δt to t, where Δt is the total parallel execution time shortened for all the procedures when the optimization is applied, and t is the total parallel execution time for all the procedures when the optimization is not applied, or $\frac{\sum_A (T_{A,p} - T^o_{A,p})}{\sum_A T_{A,p}}$.
- *Best-case improvement*: the best improvement found for a procedure, or $\max_A \frac{(T_{A,p} - T^o_{A,p})}{T_{A,p}}$.

A few observations follow.

- In general, our optimization is successful in improving the execution time of the parallel hybrid algorithm.
- The improvements in the case of one processor are entirely due to the decrease in global propagation and dynamic scheduling costs damped by the increase in local-solution cost.

- The improvements for the first two groups are more remarkable than those for the others, since procedures in the first two groups have the majority of their control flow graph nodes outside any cycles.
- In general, all improvements increase as the number of processors increases.
- The gross improvements are quite close to their corresponding mean improvements.
- Two negative mean improvements may be caused by our simple static mapping. They can also indicate that for some procedures, the increase in total local-solution cost outweighs the decrease in other costs. (These effects can be seen in Table 6 as well.)

Measure	Group	Improvement ratio (in %)							
		Real implementation				*Simulation*			
		One	Two	Four	Eight	One	Two	Four	Eight
Mean improvement	1	11	13	18	20	13	17	23	27
	2	10	13	21	20	12	16	24	24
	3	6	7	9	16	8	9	12	20
	4	3	4	2	5	6	5	4	7
	5	2	2	-2	-1	2	2	-2	0
Gross improvement	1	10	13	18	23	13	17	23	30
	2	10	15	21	24	11	17	24	27
	3	4	5	15	18	5	6	16	20
	4	4	5	4	9	6	6	6	10
	5	2	1	2	2	2	1	2	3
Best-case improvement	1	16	26	32	44	19	29	36	51
	2	15	22	39	40	17	23	41	43
	3	15	21	23	32	15	25	23	37
	4	10	13	25	16	38	13	27	18
	5	8	9	13	14	9	9	14	17

Table 8: Improvement by the optimization of a parallel hybrid algorithm.

Global propagation in the solution of Reaching Definitions is very fast. We believe that more payoff could be gained from our optimization when the data flow problems being solved need relatively more costly global propagation. As evidence for our conjecture, we conducted simulations with the cost for global-propagation tasks slightly increased.[17] Our simulation results are also reported in Table 8, under the heading labeled *Simulation*. Comparing them with the results of our real implementation, we observe better improvements.

[17] We inserted an extra loop [`for(i=0; i<50; i++) empty-statement;`] into the code for each global-propagation task.

7 Conclusions and Future Work

Performing data flow analysis in parallel is a significant step in the development of parallel compilers and interactive parallel program development environments. We have identified three types of parallelism inherent in the solution process, and presented a unified framework to exploit them. We have investigated typical Fortran programs and found significant separate-unit parallelism and algorithmic parallelism in these programs. We have used our parallel hybrid algorithm for Reaching Definitions to demonstrate the exploitation of algorithmic parallelism. The empirical results are encouraging, for we believe better payoff can be gained for interprocedural analysis. Our work in this area is the first attempt at a comprehensive approach to performing static program analysis in parallel. Our future work includes a prototype of our parallel data flow analyzer, and the design and implementation of other parallel hybrid and elimination algorithms for data flow analysis.

Acknowledgements. We thank Tom Marlowe for fruitful discussions; Rich Martin for collecting Fortran call graphs; Marc Fiunczynski for testing flow graph irreducibility; Bill Appelbe and Kevin Smith, on whose PAT system we based our front end; and the Cornell Theory Center, which provided access to their Intel iPSC/2 machine.

References

[ABC+88] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5:617–640, 1988.

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

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

[Ban79] John Banning. An efficient way to find the side effects of procedural calls and the aliases of variables. In *Conference Record of the 6th 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.

[BC86] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 162–175. ACM Press, June 1986. Palo Alto, California.

[BS90] David Barnard and David Skillicorn, editors. *Proceedings of the Workshop on Parallel Compilation*, Kingston, Ontario, Canada, May 1990. Queen's University.

[Bur90] Michael 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.

[Cal88] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Languages Design and Implementation*, pages 47–56, June 1988.

[CCH+88] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4):84–99, 1988.

[CCKT86] David Callahan, Keith Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 152–161, June 1986.

[CK84] Keith D. Cooper and Ken Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*. ACM Press, June 1984. Montreal, Canada.

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

[CK88] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, pages 151–170, October 1988.

[CK89] Keith Cooper and Ken Kennedy. Fast interprocedural alias analysis. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 49–59. ACM Press, January 1989. Austin, Texas.

[CKPK90] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer performance evaluation and the perfect benchmarks. In *Proceedings of 1990 International Conference on Supercomputing*, pages 254–266, June 1990.

[CKT86a] Keith Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization in the $\mathbf{R}^n$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, 1986.

[CKT86b] Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 58–67. ACM Press, June 1986.

[DG87] Jack J. Dongarra and Eric Grosse. Distribution of mathematical software via electronic mail. *Communications of the ACM*, 30:403–407, 1987.

[Gaf90] Neal M. Gafter. *Parallel Incremental Compilation*. PhD thesis, Department of Computer Science, University of Rochester, June 1990.

[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.

[GV91] Elana Granston and Alexander Veidenbaum. Detecting redundant accesses to array data. In *Proceedings of Supercomputing '91*, pages 854–865, November 1991.

[GW76] Susan Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 23(1):172–202, January 1976.

[GZZ89] Thomas Gross, Angelika Zobel, and Markus Zolg. Parallel compilation for a parallel machine. In *Proceedings of the SIGPLAN '89 Conference on Programming Languages Design and Implementation*, pages 91–100. ACM Press, June 1989. Portland, Oregan.

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

[HHLS90] L.A. Henderson, R.E. Hiromoto, O.M. Lubeck, and M.L. Simmons. On the use of diagnostic dependence-analysis tools in parallel programming: Experiences using PTOOL. *The Journal of Supercomputing*, 4(1):83–96, March 1990.

[HKT91] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, pages 86–100, November 1991.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[HPR89] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[KGS91] Robert Kramer, Rajiv Gupta, and Mary Lou Soffa. The combining DAG: A technique for parallel data flow analysis. Technical Report 91-8, University of Pittsburgh, Pittsburgh, PA., March 1991.

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

[LK78] J.K. Lenstra and A.H.G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, 1978.

[LMR90] Yong-fong Lee, Thomas J. Marlowe, and Barbara G. Ryder. Performing data flow analysis in parallel. In *Proceedings of Supercomputing '90*, pages 942–951, November 1990.

[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.

[LMSS91] Jon Loeliger, Robert Metzger, Mark Seligman, and Sean Stroud. Pointer target tracking - an empirical study. In *Proceedings of Supercomputing '91*, pages 14–22, November 1991.

[LR92] 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 183–190, August 1992.

[MP90] S.P. Midkiff and D.A. Padua. Issues in the optimization of parallel programs. In *Proceedings of the 1990 International Conference on Parallel Processing, Vol.II*, pages 105–113. The Penn State University Press, August 1990.

[MR90] Thomas J. Marlowe and Barbara G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 184–196. ACM Press, January 1990. San Francisco, California.

[MR91] Thomas J. Marlowe and Barbara G. Ryder. Properties of data flow frameworks: A unified model. *Acta Informatica*, 28(2):121–164, 1991.

[Mye92] Ware Myers. Special report on Supercomputing '91. *IEEE Computer*, pages 87–90, January 1992.

[OW91] T.J. Ostrand and E. Weyuker. Data flow based test adequecy analysis for languages with pointers. In *Proceedings of the 1991 Symposium on Software Testing, Analysis and Verification (TAV4)*, October 1991.

[Pol88] Constantine D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, Norwell, MA, 1988.

[PW86] David Padua and Michael Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

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

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, 1985.

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

[Sar89] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Research Monographs in Parallel and Distributed Computing. The MIT Press, 1989.

[SO90] Vatsa Santhanam and Daryl Odnert. Register allocation across procedure and module boundaries. In *Proceedings of the SIGPLAN '90 Conference on Programming Languages Design and Implementation*, pages 28–39, June 1990.

[SWJ+88] V. Seshadri, D.B. Wortman, M.D. Junkin, S. Weber, C.P. Yu, and I. Small. Semantic analysis in a concurrent compiler. In *Proceedings of the SIGPLAN '88 Conference on Programming Languages Design and Implementation*, pages 233–240. ACM Press, June 1988. Atlanta, Georgia.

[van89] André M. van Tilborg. Panel on future directions in parallel computer architecture. *Computer Architecture News*, 17(4):3–22, June 1989.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[YHR90] Wuu Yang, Susan Horwitz, and Thomas Reps. A program integration algorithm that accomodates semantics preserving transformations. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 133–143. ACM Press, December 1990. Irvine, California.

[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.

[ZBG88] Zima, Bast, and Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6(1):1–18, 1988.

[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.