IMPLEMENTATION OF A CLAN-BASED GRAPH
AND ITS USE ON PROGRAM DEPENDENCY
GRAPHS FROM FORTRAN SOURCE PROGRAMS

Technical Report 93-13

Alan D. Reed

Department of Computer Science and Engineering
Auburn University
Auburn University, AL 36849

November 24, 1993
IMPLEMENTATION OF A CLAN-BASED GRAPH 
AND ITS USE ON PROGRAM DEPENDENCY 
GRAPHS FROM FORTRAN SOURCE PROGRAMS

Technical Report 93-13

Alan D. Reed

Department of Computer Science and Engineering
Auburn University
Auburn University, AL 36849

November 24, 1993
Implementation of a Clan-based Graph Parser

and its Use on

Program Dependency Graphs from Fortran Source Programs

Project Report

for the

Master of Computer Science and Engineering Degree

by

Alan D. Reed

Submitted to

Dr. Carolyn McCreary

Dr. James Cross

Dr. Yu Gong
Table of Contents

1 Introduction .................................................. 1

2 Literature Review ............................................. 3
   2.1 Program Dependency Graph .................................. 4
   2.2 Computing the SSA form and Dependences ................. 5
   2.3 Parallelizing Assistant Tool ............................... 7
   2.4 Clans and Graph Decomposition ............................ 8
   2.5 Graph Decomposition Algorithm ......................... 9

3 Description of the Project .................................... 10
   3.1 Approach and Basic Types ............................... 10
   3.2 Parallelizing Assistant Tool ............................ 12
   3.3 SSA form of the CFG ..................................... 14
   3.4 Program Dependency Graph ................................ 16
   3.5 Clan and Clan Tree Classes .............................. 18
   3.6 Clan Parsing Algorithm ................................ 19
   3.7 Primitive Clan Breakdown ............................... 25
   3.8 Application Level Programs ............................. 28

4 Conclusions and Future Work ................................. 29

5 Examples ........................................................ 31
   5.1 Parse example ............................................ 31
   5.2 PDG Parse example ...................................... 33

6 References ..................................................... 37
1 Introduction

Parallel processor computer systems are much more difficult to program than are conventional computers. The main problem is determining which parts of the program to execute on each processor, and when those parts may execute. A trivial solution would be to just use one of the processors, but total execution time may be reduced by using several processors. Using too many processors can actually increase total execution time due to the communication costs between the processors. Determining the shortest possible execution time has been shown to be intractable, that is, an NP-complete problem. Hence, practical solutions strive to minimize total execution time by using heuristic methods.

A program is a sequential collection of statements, but the statements do not necessarily have to execute in the order given. For example, the statements

\[
\begin{align*}
A &= 2 \times D - 4 \\
B &= C / 3 + 1
\end{align*}
\]

could be executed in either order with no effect on the rest of the program. If the second statement referred to the variable \(A\) instead of \(C\), then the statements could not be executed in the reverse order. In that case, the second statement is said to depend on the first. A program can be split into independently executing parts only if these dependences are not violated.

One method of representing a program and its dependences is the program dependency graph (PDG). A PDG is a directed graph with nodes that correspond to statements and edges that correspond to dependences between the nodes. Creating a PDG is one stage of a typical optimizing compiler. PDG creation follows lexical and grammatical parsing, separation of statements into basic blocks or nodes, and addition of edges representing control flow. Traditional optimizations, such as constant propagation, are applied to the PDG. In an
automatically parallelizing compiler, special vector and parallel optimizations, such as loop unrolling, are also applied to the PDG.

Splitting a program into pieces for parallel execution can be viewed as partitioning the nodes of a graph, the PDG in this case. One approach to partitioning the nodes of a graph is clan-based decomposition. A clan is a subset of a directed graph's nodes such that each non-member node is related to all members in the same way (either ancestor or descendent or neither). Trivial clans include each node individually and the set of all nodes.

It can be shown that the nodes of an acyclic graph may be uniquely decomposed into a hierarchy of three types of clans: independent, linear, and primitive. The type of a clan in the hierarchy depends on its contained sub-clans. If the clan’s sub-clans form a sequence, with edges from one sub-clan going into the next sub-clan, then the clan is linear. If the sub-clans have no edges between them, then the clan is independent. Otherwise, the clan is primitive.

In the context of a PDG, independent clans identify sets of statements, or nodes, that may be executed in parallel, and linear clans identify nodes that must execute serially. Primitive clans do not provide any clues to potential parallelizable statements. Note that the decomposition into a hierarchy of clans shows that some sets of nodes must be executed before other sets, and that some sets may be executed in parallel with other sets. Due to the communication costs between processors, every possible parallelizable pair of nodes should not necessarily be executed in parallel. For each case it must be decided whether to parallelize or not. Based upon the target architecture, execution and communication costs can be estimated for the clans and edges in the hierarchy. For a simple example of two independent clans, they
are parallelized if the communication costs plus the maximum of execution times is less than the sum of the two execution times.

Clans and clan decomposition are general concepts, applicable to acyclic graphs, which may have other practical uses. One such use is in automatic graph layout. After parsing a graph, nodes from independent and linear clans are placed orthogonally. For example, nodes of independent clans could be placed horizontally and nodes of linear clans could be placed vertically. This places the nodes, the edges must then be routed. To reduce edge crossings, sub-clans of independent clans may exchange places.

The result of this project is software to create PDGs for programs specified in Fortran source code and to perform clan based decomposition of acyclic directed graphs. The final output, a PDG decomposed into clans, should be useful for evaluating different clan-based parallelizing schemes. The decomposition output is also usable by an existing clan-based graph layout program.

2 Literature Review

The motivation for this project comes from the Parallelizing Scalable Software Transformations (PSST) research group at Auburn University. The ultimate goal of the PSST research is automatic conversion of sequential programs into efficient parallel programs. The conversion will depend upon the target architecture and should be efficient for a range of architectures, from tightly coupled shared memory systems to distributed systems.

The overall problem may be broken into to four sub-problems: dependence analysis, dependence optimization, code partitioning, and scheduling of partitioned code on processors.
The PSST group concentrates on the last two areas, partitioning and scheduling, assuming that an optimized program dependency graph (PDG) is available. This project will produce PDGs, reducing the need to hand-craft them and increasing the supply for evaluating partitioning and scheduling methods.

The remainder of this section will discuss the research work that provides the algorithms and techniques used in the implementation of this project. This work includes: introduction of the PDG as a representation of a program, methods for computing dependences (PDG edges), introduction of PAT (a tool whose Fortran parser is used), introduction of clans and their basic theory, application of clan decomposition to PDGs, and an algorithm for the clan decomposition of graphs.

2.1 Program Dependency Graph

Ferrante [5] introduces the program dependency graph (PDG) as a directed graph whose nodes represent a program's statements or operators, and whose edges represent data values and control conditions. The motivation for the PDG is that it is a representation that allows powerful yet efficient optimizations, especially for vectorization and parallelization. Earlier approaches to program representations used various forms of data flow and data dependence graphs. Control dependences were converted to pseudo data dependences, or were represented in a form restricted to structured programs. These earlier forms were not convenient nor efficient for the application of traditional optimizations.

The paper unifies previous works on the representations of data dependences and control dependences. Basic definitions are reiterated for control flow graphs (CFG), dominators, and control dependence. A CFG is a directed graph with unique entry and exit nodes, such that
every node is on a path from the entry node to the exit node, and each node has at most two successors. Usually, the nodes represent blocks of statements from a program, and the edges represent the control flow between statements (both the normal sequential flow and branches from \texttt{gotos}, \texttt{ifs}, or loop statements). A node \( X \) dominates node \( Y \) if every path from the entry node to \( Y \) includes \( X \). Similarly, a node \( X \) post-dominates node \( Y \) (\( X \not\approx Y \)) if every path from \( Y \) to the exit node includes \( X \).

A node \( Y \) is control dependent on \( X \) if \( X \) is not post-dominated by \( Y \), and there is a path from \( X \) to \( Y \) with every (if any) node between them (i.e. not \( X \) nor \( Y \)) post-dominated by \( Y \). Essentially, this means that \( X \) must have two out edges. Following one edge always ends up at \( Y \) before reaching the exit node (the edge in the 'there is a' path). The other edge is the first on a path, which does not include \( Y \), to the exit node (\( X \) is not post-dominated by \( Y \)). Note that the path to \( Y \) may be a single edge with no nodes between \( X \) and \( Y \). Also, a node can be control dependent on itself; this is true for loop exit nodes.

A method is presented for computing the control dependences (or control condition edges) based upon the post-dominator tree of the CFG. Data dependences, including output (definition followed by definition) and anti (use followed by definition) dependences, are assumed to be computed by traditional methods. Several optimizing applications of the PDG are described, including code motion and loop fusion.

2.2 Computing the SSA form and Dependences

Cytron [2] presents an efficient algorithm for constructing a special form of CFG, the static single assignment (SSA) form. A program is in SSA form if each variable is the target
of exactly one assignment. This form includes only true data dependences; there will be no output or anti dependences. Program optimizations, such as constant propagation and dead code elimination, are more efficient and powerful when applied to the SSA form of the CFG. Our reason for including this work is that it provides a method of computing data dependences. Each variable will have exactly one definition, so data dependence edges for each variable connect the single definition node to the nodes containing uses of the variable. The paper also presents an algorithm to compute control dependences.

The SSA construction algorithm essentially converts an ordinary CFG into SSA form by renaming variables that are reused. The SSA form is trivial for straight-line code. But join nodes, where control comes from more than one predecessor node (in the CFG), present a problem. Suppose a variable \( V \) is the target of assignments in two predecessors. SSA form requires the assignment targets to be renamed, to say \( V_1 \) and \( V_2 \). A use of the original variable \( V \) in the join node cannot use just \( V_1 \) or \( V_2 \). The solution is a special function, the phi function, and the assignment of its value to yet another renamed version of the variable. The number of arguments taken by a phi function is the number of control flow predecessors to the node containing the function. The value of a phi function is its \( i^{th} \) argument, when control reaches the containing node from the \( i^{th} \) predecessor. In the simple case above, a new assignment, \( V_3 = \phi(V_1, V_2) \), would be inserted at the beginning of the join node, and uses of \( V \) in the join node would be changed to uses of \( V_3 \).

The conversion to SSA form of a CFG is performed in two steps: insert phi functions, then rename variables. Phi function insertion is based upon the concept of dominance frontiers. The dominance frontier of a node \( N \) is the set of nodes not dominated by \( N \) but which have a
predecessor dominated by N. A method is given to compute the dominance frontier for every node in the CFG, given the dominator tree. Phi functions for a variable V must be inserted in each node of the dominance frontier of a node containing an assignment of V. Variables are renamed, by suffixing a unique number, as CFG nodes are visited in dominator tree depth first order.

It is shown that control dependences may be computed using dominance frontiers. Specifically, Y is control dependent on X if and only if X is in the dominance frontier of Y in the reverse CFG.

2.3 Parallelizing Assistant Tool

Appelbe [1] describes the Parallelizing Assistant Tool (PAT), developed jointly at the Georgia Institute of Technology and the University of California at Santa Cruz. PAT is an interactive tool for helping programmers convert sequential code to parallel code. It has several aims, including increasing the efficiency of existing programs on parallel systems and being a vehicle for teaching programmers how to utilize the potential of those systems.

PAT is a source to source transformation tool for Fortran. It performs transformations, under user control, that optimize the program for a parallel or vector target machine. Several target environments are supported, and vectorized statements may be generated. This project uses the parts of the PAT software that parse Fortran source code, that build CFGs, and that compute the dominance relation and the dominator tree.
2.4 Clans and Graph Decomposition

Clan based decomposition is the PDG partitioning method being implemented in this project. The theory behind it comes from the definition and study of 2-structures by Ehrenfeucht and Rozenberg [3, 4]. The concept of a 2-structure lies between a graph and a relational structure (set of relations over a common domain). They are a generalization of graphs in that a 2-structure may be represented by a graph with labeled, directed, edges between every pair of nodes. Clans are defined for 2-structures and useful closure properties are presented.

The main result given by Ehrenfeucht is that each 2-structure has a unique decomposition into linear, complete (hereafter called independent), and primitive 2-structures. The decomposition is a hierarchy of 2-structures, where each 2-structure is represented by a single node of the hierarchically higher 2-structure. The nodes of the original 2-structure correspond to the nodes of the 2-structures at the bottom of the hierarchy, and the entire original 2-structure corresponds to a single node at the top of the hierarchy. These representations are analogous to productions in a string grammar, hence the process of finding the hierarchy is sometimes called parsing or deriving.

McCreary and Gill [7] show that the above results may be applied to graphs. There is a mapping from ordinary graphs to a class of 2-structures. The above results may be applied to mapped 2-structures, hence graphs may be uniquely decomposed into linear, independent, and primitive graphs. The nodes in these sub-graphs are clans in the original graph. The decomposition may be represented in a tree form with each tree node corresponding to a clan of the original graph. The root node corresponds to the entire graph, internal nodes correspond
to non-trivial clans, and leaf nodes correspond to the trivial singleton clans. These tree nodes may be labeled as linear, independent, or primitive according to their associated clan.

The paper shows how these results may be used to partition PDGs for parallel processing. The clans correspond to sections of the original program that may execute on separate processors. Clans may be chosen from different levels in the hierarchy so that their sizes are appropriate for any particular target parallel system. Hence the clan decomposition approach is very flexible.

2.5 Graph Decomposition Algorithm

McCReary gives an algorithm for building the clan parse tree for a Hasse graph, which is the transitive reduction of a directed acyclic graph (DAG). It first finds all clans of a graph, and then places the clans in a tree. A necessary and sufficient characterization for clans is given which provides a basis for generating node subsets that are potential clans. The clans are then the subsets that pass the characterization tests. The tests revolve around the source and sink nodes of a clan, which are nodes at the tails and heads, respectively, of edges between clan nodes and non-clan nodes. The discovered clans are placed in a tree, based upon subset relationships. The largest clan (which will be all nodes from the original graph) is placed at the root. Each clan, in decreasing size order, is added to the tree as a child of the smallest superset already in the tree. The clan types, independent, linear, or primitive, are determined throughout the process by several properties. For example, if two clans overlap, that is they intersect but neither is contained in the other, then the union of the two clans must be a linear clan.
McCreary presents a method to convert a primitive clan into a linear clan containing an independent clan and a clan of unknown type. This is done by adding edges from each of the primitive’s source sub-clans to all of the children of those source sub-clans. Adding edges to the PDG is a valid process, from the parallelization standpoint, since it adds ordering restrictions but does not remove existing ordering restrictions. Then all the source sub-clans form the new independent clan and the remaining nodes form the clan of unknown type. The remaining nodes define a sub-graph that can itself be parsed. So, via a recursive procedure, the final result will be a parse tree containing only independent and linear clans.

3 Description of the Project

3.1 Approach and Basic Types

The purpose of this project is to develop two programs: one to produce the PDG from Fortran source code, and one to produce the clan-based parse tree of a graph. The implementation approach is an object oriented style using the C++ language.

In order to avoid re-inventing the wheel, a search was made for existing classes that would be useful in the project. Generic lists, ordered lists, and stack classes from Tim Budd, Oregon State University are used. Since heavy use will be made of sets and graphs, an efficient implementation was sought. The project uses bit vector implementations by Dominic Giampaolo and Brian Walker, from the NSF sponsored Research Experiences for Undergraduates program at American University 1989.

The Set class is fairly complete, with union, intersection, difference, and complement operators, equality, subset, and superset relations, and membership and iterator functions. The
(directed) Graph class includes procedures to add and remove edges, and functions for parents, ancestors, children, and descendants of a node or set of nodes. The constructors require a size, which defines the maximum number of entities (Set members or graph nodes) in that object. The entities are integers from zero through that maximum minus one. The classes were enhanced by the addition of stream input/output operators.

Major additions were made to the graph class. Scheduling requires execution and communication times, so weights were added for nodes and edges. Edge weights are maintained in an adjacency list, but the bit matrix is retained for fast access to child nodes. The clan parsing algorithm frequently refers to ancestors and descendants of a node, so bit vector 'caches' were added for these sets.

Derived classes were created for DAGs (directed acyclic graphs) and Hasse graphs (transitively reduced graphs). The DAG class constructor from a directed graph verifies the absence of loops and in the process assigns a topological ordering to the nodes. The HasseGraph class constructor from a DAG performs a transitive reduction. The computation of the reduction turns out to be a straightforward, order $n^3$, triple loop, given bit matrix representations of adjacencies and the transitive closure:

```plaintext
for each node k
  for each node i
    if there is an edge from k to i in the graph then
      adjacency row i = adjacency row i
      AND NOT transitive closure row k
```
The last statement has the effect of removing edges from node i to any node j, where node j is reachable from node k. Edges in the transitive closure are from nodes to their descendants. As mentioned above, the descendants are 'cached' in the base Graph class.

3.2 Parallelizing Assistant Tool

The purpose of the Parallelizing Assistant Tool from Georgia Tech is to perform optimizations and transformations on Fortran source code to enhance the code's performance on parallel computers. It includes an old Fortran parser, a graphical user interface (GUI) to allow a programmer to select and/or approve transformations, and code for the analysis and evaluation of the transformations. PAT has a long history, with many authors and maintainers. It was originally written in C, and recently many parts were converted to C++. It is typical of many large old projects in that little documentation (internal comments or external documents) is available. The domain of the system is itself complex, which adds to the difficulty of comprehension.

A goal of this project is the parse tree representation of Fortran code, and it was thought that portions of PAT, such as the parser and dependence analysis, could be used. It turns out that PAT computes special purpose data dependences and performs loop control analysis to aid in its loop transformations. It does not compute all data dependences nor does it compute any control dependences. PAT does perform several important steps toward creating a program dependency graph (PDG):

- Parses Fortran source code into tokens.
• Constructs expression trees.

• Separates statements into basic blocks.

• Constructs the control flow graph (CFG).

PAT also contains much machinery that is useful for the process of creating the PDG from the CFG, including a variety of data types and computation of the dominance relation for a graph.

PAT’s FLOWGRAPH class represents a CFG. It is a subclass of PAT’s GRAPH class, and the contained nodes, FLOWNODEs, represent basic blocks. FLOWNODEs include an EXPRLIST, which is a list of EXPRESSION’s. The EXPRESSION class is a recursive structure implementing binary expression trees. The leaves of the expression tree may contain symbol names (program variables) or constants. Symbols are represented by the SYMTAB_ENTRY class, and a collection of symbols by the Symbol_Table class. There are Symbol_Table’s for the local variables for each procedure, and there is a Symbol_Table for the COMMON (or global) variables. Each statement from the original program is represented by an expression tree, which is in the expression list of the containing basic block from the CFG.

To create the CFG for a Fortran procedure, the source code must first be processed by PAT’s 'pass1' program, which performs lexical analysis and creates intermediate code in two output files. The intermediate code may then be digested by two functions (read_intermediate_code and scan_init_file), which create the CFG(s). A CFG is created for the main program and for each subroutine, and each CFG is inserted in a static (or global) list defined for the FLOWGRAPH class. This project is focused on the processes of creating and parsing a PDG, not interprocedural analysis, and so only considers the main program’s CFG, accessed by FLOWGRAPH::FlowRoot().
As discussed in the literature section, the PDG will be constructed by first converting the CFG into Static Single Assignment (SSA) form. The definition-use (DEF-USE) chains are then edges from the single definition of a variable to the use(s) of that variable, for each SSA variable. These edges are precisely the data dependence edges for the PDG. The control dependence edges will be constructed by finding the dominance frontiers of the CFG with edges reversed.

PAT’s Dominator_Tree class calculates the dominance relation for an input graph. It has methods to return the direct dominator of a given node, and to tell whether one node dominates another (i.e., is an ancestor in the dominator tree). The SSA conversion and control dependence construction require reverse dominator trees, post order traversals of dominator and reverse dominator trees, and dominance frontiers. Computing the reverse dominators (dominators in the graph with edges reversed), is defined only if the original graph has a single exit. FLOWGRAPH’s have a single exit, so the Dominator_Tree constructor was changed to accept a FLOWGRAPH (instead of a general GRAPH) and a dominator vs. reverse-dominator indicator. Calculation and access methods for dominance frontiers were also added to the Dominator_Tree class. The Dominator_Tree_Iterator class was created for the post-order traversal functions.

3.3 SSA form of the CFG

The SSA form of the CFG is computed by straightforward implementations of Cytron’s algorithms in a new class, SSA_FLOWGRAPH. The constructor takes a FLOWGRAPH, and converts that CFG (in place) into SSA form. Creating a copy of the input CFG would be better,
but deep copy methods do not exist for most of PAT's data types. The conversion is composed of three main steps:

- Convert array definitions and uses to SSA style operators.
- Insert Phi functions for each variable where required.
- Rename variables by suffixing with integers.

When an array component is assigned, it may be viewed as assigning a new value to the entire array, based on the array's previous value, the selected component, and the new component value. This view allows arrays to be treated in the same manner as scalar variables during SSA conversion. So the assignment $A(i) = x$ is transformed into the form $A = \text{Update}(A, i, x)$. Similarly, the use of an array, $x = A(i)$, is transformed into the form $x = \text{Access}(A, i)$. Actually, subroutine calls require conversion to a form that explicitly shows the variables (local and global) that are modified by the call. But as noted before, interprocedural concerns are ignored in this project. The ConvertCFGexprsToSSAstyle function performs the transformations of array definitions and uses over all expression lists in all FLOWNODEs. The UPDATE and ACCESS operators were added to the set of operators defined in the EXPRESSION class.

The PlacePhiFunctions subroutine inserts Phi functions at the beginning of join nodes for each variable. These nodes are those in the dominance frontiers of any CFG nodes that contain an assignment of the variable, including phi function assignments as they are added. The helper class VariableIterator returns all variable names, local and then global, one at a time. The inserted expressions are of the form $V = \phi(V, V, \ldots, V)$. The number of operands is the
number of predecessor nodes to the join node. The PHI operator was also added to the EXPRESSION class.

The final step to SSA form is renaming, or numbering, every occurrence of the variables. Each assignment must be to a unique instance of the target variable. Each use of a variable must refer to the instance that will be assigned during program execution, on any path to the node containing the use. In SSA form, there is exactly one such instance; it will be the target of a phi function assignment if there are multiple definitions reaching the use in the original CFG. The RenameVariables function initiates a depth first traversal of the CFG dominator tree by the RenameSearch function, which calls RenameSearchExpr to recursively examine each expression tree (i.e., program statement). Variable references are changed to specific instances by replacing the variable name in the EXPRESSION leaf node with the name concatenated with an underscore and the instance number. The new instance names are not added to PAT's symbol tables.

3.4 Program Dependency Graph

A Program Dependency Graph is represented by the PDG class, which is a subclass of the bit matrix Graph class (not PAT's GRAPH class). The PDG constructor requires one argument, an SSA_FLOWGRAPH, the results of the SSA conversion. The resulting Graph is of the type required for the ClanTree (clan parsing class) constructor. There are three steps in building the PDG:

- Convert from a basic block to a statement level node correspondence.
- Add data dependence edges.

16
• Add control dependence edges.

The parsed PDG will be used as input to a scheduling algorithm, and the granularity of scheduling can only be as fine as the PDG's smallest nodes (in terms of the number of program statements in each node). More flexibility in the scheduling algorithm is desired, so the PDG nodes will represent single statements, as opposed to basic blocks. This requires a translation during the PDG construction process, from SSA_FLOWGRAPH basic blocks to statements, which are EXPRESSION trees. The constructor counts the number of statements and allocates an array to map from PDG node numbers to EXPRESSIONs. The 'expression' method gives access to the map to users of the PDG class.

The data dependence edges are simply from the single assignment node of a variable to each of the variable's uses. An internal symbol table is created with each variable's name and the PDG node number (i.e. statement) where it is defined. Then the function Search_For_Uses is called for each node number, or statement. It traverses the statement's expression tree, and for each variable used an edge is added from the used variable's definition node to the node being searched.

The control dependence edges are computed based upon Cytron's observation that a node is control dependent on all nodes in its dominance frontier in the reverse CFG. However, with the translation from CFG nodes (basic blocks) to PDG nodes (statements), the edges added are not simply between a node and the nodes in its dominance frontier. If a basic block is control dependent on another, then all statements in the first block are dependent on the last statement of the second block. The Add_Control_Dependence_Edges function adds these edges using the Dominator_Tree class with the reverse option on the SSA_FLOWGRAPH.
3.5 Clan and Clan Tree Classes

A Clan class was created to encapsulate the description of a clan and is really just a structure with a constructor and destructor. The data includes the clan type (independent, linear, or primitive), the set of graph nodes in the clan, the source and sink node subsets, and the smallest topological number (within the graph) of the clan's nodes. ClanOrderedList and ClanStack classes are derived from Budd's generic classes. The clan list is ordered by the entries' topological numbers.

The clan parsing algorithm results in a tree, and during primitive decomposition sub-trees are manipulated and re-parsed. A tree class for which nodes are equivalent to the tree rooted at the node was desired. A search was made for an existing base tree class. Several binary tree classes were found, but a suitable n-ary tree was not.

An abstract generic tree class was developed in the same style as the Budd classes. Since it is an abstract class, it must be sub-classed to be used. The GenericTree class provides methods to add, remove, and access child nodes (trees), to insert a tree into another, and to access the parent and sibling trees (nodes). A mechanism (a virtual function) is present to allow ordering of child nodes. A separate class provides pre-order and post-order iteration functions, thus avoiding position data in every node.

The ClanTree class is derived from the generic tree class. It adds data to the tree node (the clan), adds the compare function that orders children, and adds a constructor from a DAG. The compare function in the clan tree sub-class just compares the topological numbers of the clans. In a tree node containing a linear clan, the first child sub-clan would have edges to the second, the second sub-clan would have edges to the third, etc.
The clan tree constructor from a DAG is the heart of the project. It is the implementation of McCreary's graph decomposition and primitive clan breakdown algorithms. This is done in the following steps:

- Convert the DAG to a Hasse graph.
- Apply the clan parsing (decomposition) algorithm to the Hasse graph, yielding a clan parse tree.
- Break down, or convert, all primitive clans in the tree to linear and independent clans.
- Reduce the tree, removing redundant clans from the tree.

The parsing algorithm requires a Hasse graph, so the first trivial step is to declare a Hasse graph object with the DAG as the initial value. This invokes the constructor described above. The parsing algorithm, which is a subroutine, is performed next yielding the clan tree that will ultimately be the result of the constructor. The clan tree at this time, however, may contain primitive clans. The next step is to breakdown each primitive clan, which involves re-parsing a subset of the clan. The sub-trees rooted at each primitive are replaced with new trees that may themselves contain primitive clans. By processing the clan tree in pre-order, the breakdowns can be performed in one traversal.

3.6 Clan Parsing Algorithm

The clan parsing, or decomposition, algorithm is a private subroutine that returns a clan tree given a Hasse graph and a subset of the graph's nodes. As seen from the above description of the clan tree constructor, the decomposition algorithm is performed at the beginning of the
process, and then again for each primitive clan that is found. McCreary’s algorithm starts by finding sets of graph nodes that are possible clans, and applying tests to those candidates to select the sets that are actually clans. The discovered clans are then placed in the tree hierarchy that will be returned. The types of some clans (independent, linear, or primitive) are determined during the discovery process, but some clans are determined to be linear only during a traversal of the completed clan tree.

The search for possible clans is based upon finding the sources and sinks of all clans. Note that all source nodes of a clan will have the same parents, and all sink nodes will have the same children. Two partitions of the graph’s nodes are created, one where nodes have the same parents (the sibling partition) and one where nodes have the same children (the mate partition). A clan is defined by its sibling and mate entries and all nodes in-between (all nodes on all paths from source nodes to sink nodes). Possible clans are then generated by iterating over all pairs from the sibling and mate partitions.

The partitions are implemented with an abstract base Partition class and two derived specialized classes, SiblingPartition and MatePartition. The base class contains the partitioning algorithm, which is to compare the equivalence set (parents or children) of each member to the equivalence sets already found. If there is a match, the member is added to the corresponding equivalence set, otherwise a new equivalence set is created. The virtual method equivTestSet is specialized to return parents in the SiblingPartition class and children in the MatePartition class. The derived classes’ constructors take a sub-graph, that is, a graph and a set of the graph’s nodes, and call the base partitioning algorithm. The results are accessible via the size
Figure 1. Example Graph.

(number of equivalence sets or partitions) and members methods. SiblingPartition's aStar method returns the union of the members of a partition entry and their ancestors within the subgraph; MatePartition's dStar method is similar but with descendants.

SiblingPartition (Potential source sets)

Nodes in partition
\{0\} \{1\} \{2-4\} \{5\} \{6\} \{7\}

Descendants (dStar)
\{0-7\} \{1-7\} \{2-7\} \{5 7\} \{6 7\} \{7\}

MatePartition (Potential sink sets)

Nodes in partition
\{0\} \{1\} \{2 3\} \{4\} \{5 6\} \{7\}

Ancestors (aStar)
\{0\} \{0 1\} \{0-3\} \{0 1 4\} \{0-6\} \{0-7\}

Figure 2. Partitions for the example graph
For each pair of entries from the sibling and mate partitions, a potential clan consists of all the sources (member nodes from the sibling partition entry), all the sinks (member nodes from the mate partition entry), and the nodes between them. This set of nodes is just the intersection of the aStar and dStar sets from the sibling and mate partition entries, respectively. If the set is empty, then obviously it is not a clan. If the set contains one element, then it is a trivial clan. Every graph node is a trivial clan, and these singleton clans will be handled later to avoid the complicated case that follows.

The remaining case is that the set contains more than one graph node. The entire set may be a clan, or a subset may be, or no clans may be contained in the set. If the set contains an independent clan, then it is a maximal independent clan and the sub-clans are separate connected components. So the first step in examining the potential clan set is to find the (weakly) connected components.

The ConnectedComponent class provides that function. Its constructor takes a graph and a subset of the graph’s nodes, and it finds the connected components. Components are found by starting with a node from the subset and then adding ancestors and descendants of nodes in the component until no new nodes are added. The nodes in the component are removed from the graph’s subset, and the process continues until the subset is emptied. The number of components and the set of nodes in each component are then accessible via the numberOfComponents and subscript operator methods.

A connected component will by construction contain at least one of the potential sources (nodes from the sibling partition entry) and at least one of the potential sinks (nodes from the
Figure 3. Potential clans of the example graph.

mate partition entry). If the component contains only one node, then it is a trivial clan. Otherwise, the component is a clan if there are no illegal entries or exits, that is, all parents and children of the internal (non-source and non-sink) nodes are contained in the component. These illegal edges can be detected by the expressions:

\[ A^*(M) \not\subset (A^*(S) \cup D^*(S)) \quad \text{implies illegal entry} \]

\[ D^*(S) \not\subset (D^*(M) \cup A^*(M)) \quad \text{implies illegal exit} \]

where

\[ S \] is the set of sibling partition entry nodes in the connected component

\[ M \] is the set of mate partition entry nodes in the connected component

\[ A^*(X) \] is \( X \) and all ancestors of all nodes in \( X \)
\( D^*(X) \) is \( X \) and all descendants of all nodes in \( X \)

If more than one connected component (including singletons) is a clan, then the union of all of those clans is a maximal independent clan. The type(s) of the individual connected component clan(s) are unknown.

Each non-singleton connected component clan, and the independent clan (if one), is then merged into the list of clans already found. If a new clan matches an existing clan, the new type (if known) is updated to the existing clan. Otherwise, if the new clan overlaps an existing clan, but neither is a subset of the other, then the two clans must be part of a linear clan. The existing clan is deleted and the new clan is replaced by the union of the two clans, which is marked as linear. The merge process continues with the replaced linear clan. If the new clan does not match or overlap any existing clan, then it is added to the list of existing clans. If its type is still unknown, it is labeled as primitive.

<table>
<thead>
<tr>
<th>Candidate</th>
<th>Clan list after merging the candidate</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0,1}</td>
<td>{0,1}</td>
</tr>
<tr>
<td>{0-6}</td>
<td>{0,1}, {0-6}</td>
</tr>
<tr>
<td>{0-7}</td>
<td>{0,1}, {0-6}, {0-7}</td>
</tr>
<tr>
<td>{1-6}</td>
<td>L:{0-6}, {0-7} overlaps {0,1}</td>
</tr>
<tr>
<td>{1-7}</td>
<td>L:{0-7}</td>
</tr>
<tr>
<td>I:{2,3}</td>
<td>I:{2,3}, L:{0-7}</td>
</tr>
<tr>
<td>{2-6}</td>
<td>I:{2,3}, {2-6}, L:{0-7}</td>
</tr>
<tr>
<td>{2-7}</td>
<td>I:{2,3}, P:{2-6}, P:{2-7}, L:{0-7}</td>
</tr>
</tbody>
</table>

where I: = independent, L: = linear, and P: = primitive

**Figure 4.** Non-singleton legal clans from the example graph.

After testing all connected components of all sets generated from all pairs of sibling and mate partition entries, all non-singleton clans will have been found. All independent clans will be labeled as such and are maximal. All maximal linear clans with at least three sub-clans will
be labeled as linear. Two element linears, including some subsets of the maximal linears, will be labeled as primitive. All primitive clans will be labeled as such.

The result is to be a tree, so the next stage of the parsing algorithm is placing each clan in its proper place in the parse tree. The root of the tree is the clan of the entire sub-graph, which will be the largest clan found. The remaining clans are the internal nodes of the tree, and each must be the child of a clan that is a superset. Hence the tree may be constructed by inserting each clan, from the largest to the smallest, in the tree as a child of the smallest superset already in the tree. The singleton clans, which will be leaf nodes of the tree, still need to be inserted. These could be inserted in the same fashion as the non-singleton clans (searching for the smallest superset), but it is simpler to make one post-order traversal over the tree, adding as leaf clans any nodes that are in the current clan but not already visited.

The final step is to determine which, if any, of the clans that are labeled as primitive are actually linear clans. Each clan labeled as primitive is examined; if the sinks of each sub-clan have edges to all the sources of the next (in topological order) sub-clan, then the clan is in fact a linear clan. This should only be the case for linear clans that contain two sub-clans; if a linear clan contains more than two sub-clans, adjacent pairs of the sub-clans (which overlap) will be found and result in proper labeling of the entire (maximal) linear clan.

3.7 Primitive Clan Breakdown

The primitive clans are broken down using the method mentioned previously (adding edges from each source to children of all sources). By construction, the method’s result is always a linear clan containing an independent clan (of the primitive’s source clans) and another
clan (of the primitive's non-source clan nodes). It is not necessary to actually insert graph edges and re-parse the entire graph to create the new clans; the linear and independent clans can just be constructed. The non-source nodes will be a clan, again by construction, but its clan decomposition has no relation to the original structure before edges were 'added'. The structure of a clan is the same whether it is a subset of a larger graph or it is the whole graph, so the structure of the 'other' non-source nodes clan may be determined by re-applying the clan parsing algorithm to the sub-graph composed of those nodes.

The implementation, using the inherited generic tree operations, is now described. A single preorder traversal is made over the initial clan tree. When a primitive clan is found, all of its child clans are examined for source nodes of the primitive. Child clans containing a source node (which must then be a source sub-clan) are moved to a newly created clan tree, which is labeled as an independent clan. Other, non-source clans are discarded. The primitive clan's tree is now empty. The primitive clan is re-labeled as linear, and the independent clan
tree just constructed is added as a child. Finally, the clan tree result from parsing the remaining non-source nodes is added as a child of the re-labeled linear clan.

The sources of P:\{2-6\} are \{2\}, \{3\}, and \{4\}, which are labeled as a pseudo independent (i:). The remaining nodes are \{5,6\}, which parse as an independent clan.

```
L:0-7
  S:0
  S:1
L:2-7
  L:2-6
  S:7
  i:2,3,4
  I:5,6
  S:2
  S:3
  S:4
  S:5
  S:6
```

**Figure 6.** Parse tree of the example graph after breakdown of primitives.

After the primitive breakdown step the initial clan tree has been transformed into a tree containing only linear and independent clans. The clan parsing algorithm finds maximal linear and independent clans, so linear clans will not contain linear clans (as immediate children) and independent clans will not contain independents. But if a primitive clan is a child of a linear clan, or if the re-parsed 'other' clan is linear, then the breakdown process will yield a linear in a linear. A linear clan in a linear clan does not add any structural information, in fact, any subset of contiguous children of a linear clan is also a linear clan. This is also true for independent clans. So the final step before returning the clan tree is a clean-up operation: if a clan's type is the same as its parent clan's type, then its children are moved up a level by making them children of that parent clan. This operation can be done in one post-order traversal
of the tree. The final result is a bipartite tree, with levels alternating between all linear clans and all independent clans.

![Final parse tree of the example graph](image)

**Figure 7.** Final parse tree of the example graph.

### 3.8 Application Level Programs

Two driver programs were written to make the objectives of the project accessible. The first, `parse`, reads a graph description and outputs the clan parse tree of the graph. It complains if the input graph contains a cycle. As mentioned in the introduction section, there may be several applications for clan based parsing. One in particular, automatic graph layout, has already been implemented by Clint Combs. That program expects a description of a graph including its clan structure. The parse program has an option to change the output format to that expected by the graph layout program, yielding a convenient front end. The command line to execute the graph parser is:

```
parse {-c} {-d} {graphfile}
```

All arguments are optional. The `-c` option requests that the output be formatted for Clint Combs's graph layout program. The `-d` option causes intermediate results to be printed, for debugging purposes or to gain insight into the mechanics of the parsing algorithm. The
graphfile (or standard input if no file is given) must be in the format expected by the Graph class constructor from a stream. This format is shown in the Examples section.

The second program, pgp_parse, creates the PDG for a program and parses the resulting graph. The input is the intermediate code version of Fortran source code that is created by PAT's pass1 program. The output includes the CFG, the PDG, the parse tree, and the statement expressions for each PDG node. The PDG parser requires the intermediate code version of the Fortran program to be analyzed. This is produced by PAT's pass1 program by executing the following command:

```
pass1 source
```

The source code file is assumed to have a '.f' extension, and the intermediate code will be placed in files with the same name but with extensions of '.i' and '.x'. Then the PDG parser may be run by a command formatted as shown below:

```
pdgparse {-d} source
```

Again, the file name given should not include extensions since these are assumed. The optional -d argument requests intermediate results to be printed, including the original and SSA form of the CFGs. The output always includes the PDG, in the Graph class output format, the statement expressions associated with each PDG node number, and the parse tree.

4 Conclusions and Future Work

The main benefit of this work is that it provides a base from which different clan based partitioning/scheduling schemes may be evaluated. At this time, before applying a scheduling method, graphs are parsed by hand. This is a tedious and time consuming process, and can be error-prone. It is practically impossible for large graphs, making it difficult to evaluate potential
scheduling methods. This project should alleviate these problems and make it possible to actually implement scheduling methods.

One original goal of the project was to determine the complexity, or order of magnitude, of the execution time of the clan parser. This is very difficult, if not impossible, to approach analytically. The sizes of the sibling and mate partitions and the number of primitive clans are important factors that are not directly related to the size of the input graph. This also makes an empirical approach difficult, since some measurable aspect(s) of the input graph are expected to be related to the parsing time. The set of graphs measured would also be suspect.

A small improvement to the clan parsing algorithm is suggested in this paper: the simpler method for labeling linear clans near the end of the decomposition portion of the algorithm. Testing the number of nodes, instead of comparing sinks to parents and children to sources, would be faster. If this suggestion is verified to be correct, the execution time for the algorithm would be slightly improved (certainly not by an order of magnitude).

The clan parsing algorithm requires a DAG, so graphs including loops cannot be parsed. This seems to severely restrict the uses of clan parsing. However, in the context of scheduling, cycles are a problem for all methods in the literature, and clan parsing may be compared with other methods for acyclic PDGs. For other applications of clans, such as graph layout, there may be techniques for eluding cycles. For example, cycles may be removed by reversing back edges before parsing and restoring the edges during layout.

Scheduling code for execution on a parallel system requires communication and execution costs. The Graph class as implemented includes node and edge weights for these costs. It
should be possible to estimate these weights based on the number and type of operators and the
number and sizes of operands.

Several simplifications were made in the construction of the PDG. PAT converts DO
statements into a single expression with the DO variable as the target of a range, i.e. DO 10
I=1,10 turns into a CBRANCH 10, I = (1, 10). For proper conversion to SSA form, there
should be an initial assignment definition, a use as it is compared, and a definition and use as
it is incremented. Call statements also require a conversion to SSA form, so that side effects
(assignments to argument variables or globals) may be accounted for. The control dependence
edges in the PDG correspond exactly to the definition of control dependence. Region nodes,
which combine dependences, were not implemented.

5 Examples

5.1 Parse example

Following are examples of graph parsing and PDG construction from Fortran. The
programs reside and examples were executed on the Auburn University College of Engineering
network. This work is part of the PSST project; the PSST home directory on the network is
/proj/mcclan. The graph parser and PDG constructor and parser are in the bin directory of
the project, /proj/mcclan/bin, and are named parse and pdgparse, respectively.

The main purpose of the first example is to show the graph description format used by
the Graph class. The graph used in demonstrating the clan parsing algorithm, its description,
and the output from the parse program for that graph are presented below:
8
0 1 (1 1);
1 1 (2 1) (3 1) (4 1);
2 1 (5 1);
3 1 (5 1);
4 1 (5 1) (6 1);
5 1 (7 1);
6 1 (7 1);
7 1;

The first line of the description contains the number of nodes. Remaining lines, one per node, contain the node number (from 0 to the number of nodes - 1), the node weight, the list of edges (if any) from the node, and a semicolon. Each edge, in parentheses, includes the target node and the edge weight.

The output of the parse program, given the above graph, is shown below. This is the bipartite parse tree with primitives decomposed:

```
Parse tree:
L: { 0-7 } {
  S: { 0 }{
    S: { 1 }{
      i: { 2-4 } {
        S: { 4 }{
          S: { 3 }{
            S: { 2 }{
          }{
        }{
      }{
    }{
  }{
}
```

Each line represents a node of the parse tree, with the indentation and parentheses showing the depth of the node. The root of the tree is the first, left-most line. Each node represents a clan and includes the type (L for linear, I for independent, i for pseudo-independent from a primitive decomposition, or S for singleton) and the set of original graph nodes in the clan.
5.2 PDG Parse example

A sample Fortran program and the results from pdgparse will now be presented. The program is from an example in the Cytron [2] paper, except that the two loop branches have been commented out. The loops were removed so that the PDG would be a DAG (acyclic graph), and thus be parse-able.

C Figure 5 from Cytron, et. al., Efficiently Computing Static Single C Assignment Form and the Control Dependence Graph

C234567890
PROGRAM CYTRON6
  INTEGER I, J, K, L
  INTEGER P, Q, R, S, T
  I = 1
  J = 1
  K = 1
  L = 1

10 CONTINUE
  IF (P .EQ. 1) THEN
    J = I
  IF (Q .EQ. 1) THEN
    L = 2
  ELSE
    L = 3
  ENDIF
  K = K + 1
  ELSE
  K = K + 2
  ENDIF
  CALL PRNT(I, J, K, L)

15 CONTINUE
  IF (R .EQ. 1) THEN
    L = L + 4
  ENDIF
  C  IF (S .EQ. 1) GO TO 15
  I = I + 6

C  IF (T .EQ. 1) GO TO 10

END
A CFG (control flow graph) is printed by PAT in the following form. Each basic block is given a name (i.e. MAIN, DN@00000). The first lines of a block include the name of the block and the successor block(s) in the CFG. The remaining lines, indented two levels, contain the prefix style listing of the statements in the block. This output is after the SSA conversion; the original CFG is the same except for the phi functions and numbered variables.

---------- Flowgraph in SSA form ----------
MAIN (line 5) branches to:
    L13

L13 (line 11) branches to:
    DN@000000
        = (i_1) (1)

DN@000000 (line 11) branches to:
    DN@000001
        = (j_1) (1)
        = (k_1) (1)
        = (l_1) (1)

DN@000001 (line 17) branches to:
    DN@000002
    DN@000006
        CBRANCH (.eq. (p_0) (1)) (0)

DN@000006 (line 26) branches to:
    _000prnt
        = (k_2) (+ (k_1) (2))

_000prnt (line 29) branches to:
    DN@000008
        Phi (1_2) (, (1_6) (1_1))
        Phi (k_3) (, (k_4) (k_2))
        Phi (j_2) (, (j_3) (j_1))

DN@000008 (line 32) branches to:
    DN@000009
    DN@000010
        CBRANCH (.eq. (r_0) (1)) (0)

DN@000010 (line 36) branches to:
    L14
        Phi (1_3) (, (1_2) (1_4))
        = (i_2) (+ (i_1) (6))

L14 (line 40) is terminal.

DN@000009 (line 32) branches to:
    DN@000010
        = (l_4) (+ (l_2) (4))

DN@000002 (line 17) branches to:
    DN@000003
DN@00004
    = (j_3) (i_1)
    CBRANCH (.eq. (q_0) (1)) (0)

DN@00004 (line 22) branches to:
DN@00005
    = (l_5) (3)

DN@00005 (line 24) branches to:
    _000prnt
    -000prnt
    Phi (l_6) , (l_7) (l_5))
    = (k_4) (+ (k_1) (1))

DN@00003 (line 19) branches to:
DN@00005
    = (l_7) (2)

The statement level PDG computed from that SSA CFG is shown below:

------------- PDG -------------

19
  0 1 (11 1) (13 1);
  1 1 (8 1);
  2 1 (5 1) (17 1);
  3 1 (6 1);
  4 1 (5 1) (13 1) (14 1) (16 1)
      (17 1);
  5 1 (7 1);
  6 1 (10 1) (12 1);
  7 1;
  8 1;
  9 1 (12 1);
  10 1;
  11 1;
  12 1 (10 1);
  13 1 (8 1);
  14 1 (15 1) (18 1);
  15 1 (16 1);
  16 1 (6 1);
  17 1 (7 1);
  18 1 (16 1);

Node number to Expression map:
  0  i_1=1
  1  j_1=1
  2  k_1=1
  3  l_1=1
  4  branch on ((p_0).eq.(1)) to L0
  5  k_2=k_1+2
  6  l_2=Phi(l_6, l_1)
  7  k_3=Phi(k_4, k_2)
  8  j_2=Phi(j_3, j_1)
  9  branch on ((r_0).eq.(1)) to L0
 10  l_3=Phi(l_2, l_4)
 11  i_2=i_1+6
 12  l_4=l_2+4
 13  j_3=i_1
 14  branch on ((q_0).eq.(1)) to L0
 15  l_5=3
 16  l_6=Phi(l_7, l_5)
 17  k_4=k_1+1
 18  l_7=2

The parsed clan hierarchy of that PDG is:

---------- Clan Parse Tree ----------

L: { 0-18 } 
  i: { 0-4 9 } 
  S: { 0 }
  S: { 1 }
  S: { 2 }
  S: { 3 }

35
\begin{verbatim}
S: { 4 }
S: { 9 }
I: { 5-8 10-18 }
S: { 11 }
L: { 5 7 17 }
I: { 5 17 }
S: { 17 }
S: { 5 }
S: { 7 }
L: { 6 10 12 14-16 18 }
S: { 14 }
I: { 15 18 }
S: { 18 }
S: { 15 }
S: { 16 }
S: { 6 }
S: { 12 }
S: { 10 }
L: { 8 13 }
S: { 13 }
S: { 8 }
\end{verbatim}

Figure 9. Final parse tree of the example program.
6 References


