Determining Transformation Sequences for Loop Parallelization

Bill Appelbe
Charles Hardnett

Srinivas Doddapaneni
Kurt Stirewlt

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Kevin Smith
Department of Mathematics and Computer Science
Emory University
Atlanta, GA

Abstract

Considerable research on loop parallelization for shared memory multiprocessors has focused upon developing transformations for removing loop-carried dependences. In many loops, more than one such transformation is required, and hence the choice of transformations and the order in which they are applied is critical.

In this paper we present an algorithm for selecting a sequence of transformations which, applied to a given loop, will yield an equivalent maximally parallel loop. The model is extensible to loop nests, and loops with control dependences. We also discuss incorporating performance models to determine the profitability of parallelism into the algorithm. The algorithms provided in the paper have been implemented and tested in PAT, a tool for interactive parallelization of Fortran.

Categories and Subject Descriptors: [**Programming Languages**]: Processors—*compilers, optimization*
General Terms: Compilers
Additional Key Words and Phrases: Dependence Analysis, Parallel Programming

1 Introduction

The majority of the potential parallelism in numerical/scientific programs occurs in loops. However, before loops can be parallelized they must often be transformed. Such transformations have as their principal goal the removal of all *loop-carried dependences* [18], in which an array variable is updated in one iteration and accessed in another. Research on loop parallelization can be classified as follows:

- Development of more explicit dependence tests, so that ranges of complex subscript expressions can be analyzed to determine if two subscript expressions can reference the same memory locations [15, 18, 6, 14].
- Development of new and improved transformations for *removing* or *protecting* dependences. Transformations which remove a dependence restructure the loop so that the dependent references lie entirely within a single iteration. Transformations which protect a dependence do not actually eliminate the dependence; instead, they introduce explicit synchronization (LOCKs or EVENTs) which prevents concurrent access/update.
- Development of program representations such as SSA/PDG [10, 9] and algorithms for systematizing the removal of all dependences [7, 5].

This paper addresses the third category. Callahan [7] developed a systematic approach to parallelization based upon minimizing the number of barrier synchronization points in a program. By contrast, our approach to parallelization is based upon the *fork-join* model of parallelism, and focuses upon achieving maximum parallelism for an individual loop.

The overall approach can be characterized as follows: Given a serial loop, convert it into a parallel loop via source code transformations; if the loop cannot be converted into a single parallel loop, convert it into a sequence of parallel and serial loops. The objective is to minimize the size of any remaining serial loops and to use a sequence of parallel loops with the lowest overhead, based upon a cost model for parallel loop overheads. If a loop has several loop carried dependences we are able to systematically remove these using the algorithms and representation below, provided that we can determine the the dependence distance or direction.

In order to discuss sequences of transformations we need some way to represent such sequences, and their affect on the program. We use a specialized program dependence graph, called the Inter-Variable Dependence Graph, or IVD Graph for this purpose, which includes information about the relationships between sequences of dependences. In addition we introduce a formal framework to justify the correctness of the transformations.

In the following section we outline the assumptions we adopt in our program model, argue for their validity and describe the ways in which they can be relaxed by more refined analysis and transformations. We then discuss the transformations applied, and introduce algorithms and heuristics (in situations where the selection of transformations for maximal parallelism is proved NP-complete) that generate a *maximally parallelized* loop.

2 Program Representation

Most discussions of program transformations focus on applying a single transformation, from a family of transformations, to a program. The correctness of the transformations are usually informally argued, based upon dependences and other program constraints. Some authors [7] have looked at sequences of transformations based upon global optimization criteria, but there has been no attempt to our knowledge to develop a formal framework for the entire process or to implement a systematic approach to selecting and applying sequences of transformations. In this paper we focus upon selecting sequences of transformations to obtain maximal parallelism for a single loop. Later we discuss the relationship between this model and other program representations and algorithms for parallelization.

The overall process of applying transformations can be summarized as:

$$\begin{array}{rc} SourceProgram & \overrightarrow{parse} \\ AbstractProgram_1 & \overrightarrow{transform} \\ AbstractProgram_2 & \overrightarrow{transform} \\ AbstractProgram_3 & \ldots \end{array}$$

An $AbstractProgram$ is a program representation such as a parse tree or abstract syntax description from which a $SourceProgram$ can readily be reconstructed. The abstract program representation we use is discussed below. The $SourceProgram$ is presumed to be in a programming language such as Fortran 77. Since the transformation between source and abstract programs is purely syntactic we will ignore it in further discussion, the term $Program$ will be used as an abbreviation for $AbstractProgram$. The correctness of the transformations requires a definition of equivalence:

$$Program_i \equiv Program_k \tag{1}$$

We define equivalence as being the values of all global variables are identical upon completion of the programs. Such equivalence is clearly transitive.

To discuss sequences of transformations we need to represent the program more abstractly, with information that describes program dependences. Examples of such representations are Program Dependence Graphs (PDGs) [10]. The representation we adopt below, the IVD graph, is tailored to representing loop-carried dependences. Reconstructing programs from such representations is more difficult than the syntactic transformation between source programs and abstract programs. For the IVD representation we define the mapping of programs to graphs:

$$ivdgraph : PROGRAM \mapsto IVDGRAPH \tag{2}$$

and the inverse mapping

$$program : IVDGRAPH \mapsto PROGRAM \tag{3}$$

so that

$$program(ivdgraph(Program_i)) \equiv Program_i \tag{4}$$

that is, the result of mapping a program to a graph then back to a program is equivalent to the original program.

Each transformation actually has two versions, a program transformation and the corresponding graph transformations:

$$\begin{array}{ccc} Program_i & \overrightarrow{program\_transform} & Program_k \\ \downarrow ivdgraph & & \uparrow program \\ Graph_i & \overrightarrow{graph\_transform} & Graph_k \end{array} \quad (5)$$

For each transformation tf, we need to show:

$$Program_i \equiv program\_tf(Program_i) \quad (6)$$

and

$$Program_i \equiv program(graph\_tf(ivdgraph(Program_i))) \quad (7)$$

that is, both the program transformation and the graph transformation lead to programs that are equivalent to the original program.

3 A Parallel Programming Model

The most widely used languages for programming shared memory multiprocessors are dialects of sequential languages such as Fortran and C with extensions for fork-join parallelism [12] in which a parallel program consists of sequences of alternating serial code and parallel loops and sections. All iterations of a parallel loop, or all cases of a parallel section, can be executed in parallel by an arbitrary number of tasks. We assume that the target of parallelization is a serial loop, since the majority of parallelism in practice occurs at the loop level. Hence, parallel loops (referred to as PARALLEL DO's) will be the target parallel construct.

In the analysis below we assume for simplicity that a loop bounds have been normalized to the range 1 to N, and the loop body consists of a sequence of assignment statements without reassignments to array variables. We also assume that there are no explicit jumps into or out of the loop body. Further we assumed that there are no assignments to scalar (including reduction) variables and no control flow statements (e.g., IF statement) inside the body of the loop. In fact the last assumption is not a limitation of the framework, later we show that both scalars and control flow can be handled in the IVD framework. Source code transformations such as *if-conversion* can be used to represent control-flow statements. A more general approach for representing control dependences is the Program Dependence Graph (PDG) [10], and later we discuss how this representation can be merged with the IVD representation.

Alignment of assignments within inner loops can also require that both the inner and outer loops are also aligned if the sign (direction) of the dependence is different in the inner loop.

Scalar assignments within a parallel loop can often be treated independently, as instances of reduction variables [16, 2] or they can be expanded to arrays [1]. Assignments to scalar variables can be handled without array expansion if the following restriction is met. After transformations on the loop body, the scalar value should be accessed either in the current iteration or in the next iteration. Therefore a scalar variable is sufficient to hold the current value. Reassignments to arrays can often be reduced to a single assignment by detecting the final assignment and replacing other references with temporaries [3].

Throughout the definitions of the transformations below we will adopt the following notation for programs:

I is the loop variable,

K, $K1$, $K2$, ... are integer constants, positive constants are denoted by + superscripts, such as K^+.

A, B, C, D, ... are one dimensional arrays read or written in the loop, with subscript expressions that are functions of I, usually linear. $A[I+K^+]$ is an array reference with a known subscript expression, whereas A_I denotes an array reference with a subscript expression which is an unknown function of I.

f, fa, fb, ... are arbitrary functions of array references (A_I, B_I, etc.)

Thus, a program consists of a loop with a sequence of statements of the form
$A_I = fa(A_I, B_I, C_I, ...)$,
Array reads may appear more than once on the right-hand side, such as
$A[I] = f(A[I+1], A[I-1])$
The fundamental assumption underlying our parallelization model is that all dependence directions are known (and preferably distances). If one or more dependence directions are unknown, then those statements involved in the unknown dependences must be

kept in the same serial loop. The model can readily be extended to handle unknown dependences, which are similar to dependence cycles or recurrences discussed below.

The correctness of the transformations that can be applied sometimes depend on whether a dependence direction is negative or positive. Thus, when needed we will label reads with their dependence direction. For example

$A[I] = f(A^{old}[I+1], A^{new}[I-1])$

The superscript *OLD* means a +ve dependence direction (distance +0 and above), and the superscript *NEW* means a -ve dependence direction (distance -0 and below). Using this notation for each read access effectively changes the loop into single-assignment form.

An arbitrary loop can be executed as a PARALLEL DO except where there is a forced ordering on the iterations of the loop; these forced orderings are identified by *loop carried dependences*. Loop carried dependences are derived from computations whose value will be affected by the order of execution of loop iterations. However, the program transformation below can be used to remove all loop carried dependences except where there are dependence *cycles*, or recurrences. Even these recurrences can be reduced to *single statement recurrences* (SSR's). An SSR consists of an assignment of the form

$A_I = f(A_I^{new}, B_I, C_I, ...)$,

Loops containing SSR's can still be parallelized by distributing the loop into three loops: an initial parallel loop, a sequential loop containing the SSR, and a trailing parallel loop.

```
DO I=1,N
  ...
C initial parallelizable code
  A[I] = f(A[I - K+], B_I, C_I, ...)
C more parallelizable code
  ...
ENDDO
```

```
PARALLEL DO I=1,N
  ...
C initial parallelizable code
ENDDO
DO I=1,N
  A[I] = f(A[I - K+], B_I, C_I, ...)
ENDDO
PARALLEL DO I=1,N
C more parallelizable code
  ...
ENDDO
```

This effect could be achieved by distributing all tightly coupled sections of code into separate loops but this increases unnecessarily the overhead of starting each parallel loop. We identify a *maximally parallelized* loop as one in which the only remaining dependences are in *single statement recurrence* (SSR) sequential loops, and in which the number of separate parallel loops is minimal. SSR's can sometimes be parallelized somewhat further [8] but we do not attempt to do so.

The analysis below assumes that the number of processors p is 'significantly less' than the number of loop iterations N for loops considered for parallelizing. This assumption is valid in practice considering the number of processors in typical shared memory systems and the loop structure of numerical computation programs, and implies that maximal parallelism can be achieved without using nested parallel loops (i.e., all processors can be kept 'busy' executing the parallel code of a single loop). The assumption of a large number of iterations implies that the overhead of peeling a few iterations for alignment can also be ignored.

4 Transformations for Parallelism

There are a large number of specialized transformations and techniques for parallelization. Our algorithm uses the following intra-loop transformations: alignment, value replication (or strip mining), assignment replication, and expression substitution. We define each of these transformations below. Each of the transformations maps sequential to sequential loops, and can be combined. The goal is to compose a sequence of transformations to generate one or more loops with no loop-carried dependences, which can then be executed as parallel loops.

4.1 Alignment

Alignment shifts subscripts in an assignment so that a read access of an array A in one assignment uses the same subscript value as a write access of A in another assignment. The general form of alignment is:

```
DO I=1,N
  B_I = fb(A[I - K+ + K1], C_I, ...)
  A[I + K1] = fa(C_I, ...)
  ...
ENDDO
```

After alignment of the assignment to B, by K^+:

```
C Calculate K iterations of B
DO I=1,K+
   B_I = fb(A[I - K+ + K1], C_I, ...)
ENDDO
C Peel K iterations
DO I=1,N-K+
   A[I + K1] = fa(C_I, ...)
   B_I |^I_{I+K+} = fb(A[I - K+ + K1], C_I, ...) |^I_{I+K+}
   ...
ENDDO
C Calculate K iterations excluding B
DO I=N-K+ +1,N
   A[I + K1] = fa(C_I, ...)
   ...
ENDDO
```

The notation $B_I \mid^I_{I+K^+}$ means "substitute $I + K^+$ for I in B_I". We will use a more general substitution notation later.

Alignment changes loop-carried dependences into intra-iteration dependences: after alignment is performed the aligned statement may need to be reordered as follows. If the direction of the alignment is positive, as above, the aligned statement must be after the assignment. This may require statement reordering, as in the example above.

Either of the two statements carrying the dependence could be aligned. However, if a sequence of alignments is to be performed, the choice of which statement to align matters, as explained in section 5.

When an alignment by K is performed, the loop bounds have to be altered and K iterations have to be *peeled* from the loop. Proving the correctness of alignment is simplified if alignment is regarded as an optimization of extending the loop bounds, and protecting each statement in the loop with the range of loop iterations for which it is to be performed:

```
DO I=1-K+,N
   IF(I ≥ 1) A[I + K1] = fa(C_I, ...)
   IF(I ≤ N - K+)
      B_I |^I_{I+K+} = fb(A[I - K+ + K1], C_I, ...) |^I_{I+K+}
   IF(I ≥ 1) ...
ENDDO
```

If K is negative the above transformations are minimally different: the alignment direction is reversed and hence iterations are peeled on opposite ends of the loop and the aligned statement must be before the assignment.

An important property of alignment is that it is symmetric, unlike other transformations. In the example above we could have aligned the assignment to A by a factor of $-K^+$ instead of aligning the assignment of B (containing the read of A) by K^+.

4.2 Value Replication

Value replication makes temporary copies of an array variable whose values are overwritten in different iterations from those in which they are used. Value replication can be defined as:

```
DO I=1,N
   A_I = fa(C_I, ...)
   B_I = fb(A_I^old, C_I, ...)
   ...
ENDDO
```

After replication of array A:

```
C Copy A
DO I=1,N
   AA[I] = A_I
ENDDO
DO I=1,N
   A[I] = fa(C_I, ...)
C Substitute AA for A in the assignment to B
   B_I = fb(AA[I], C_I, ...)
   ...
ENDDO
```

Value replication is defined as a substitution of variables, as compared with a substitution of subscripts for alignment.

Strip mining (or blocking or tiling for loop nests) is an alternative transformation to value replication with a significantly reduced cost (both memory required and execution time) in many cases. Unlike value replication, strip mining requires that the dependence distance is known:

```
C NSTRIPS == number of strips
C LSTRIP == iterations per strip
C N == NSTRIPS*LSTRIP

C Copy A at ends of strips
DIMENSION AA(NSTRIPS,K+)
DO J=1,NSTRIPS
   DO I=1,K+
      A[J][I] = A[LSTRIP * J + I]
   ENDDO
ENDDO
DO J =1,NSTRIPS, LSTRIPS
   DO I=J,J+LSTRIPS-1
      A[I] = fa(C_I, ...)
      IF(I.LT.J+LSTRIP-K+) THEN
         B_I = fb(A[I + K+], C_I, ...)
      ELSE
         B_I = fb(AA[J][I - (J + LSTRIP - K+ - 1)], C_I, ...)
      END IF
      ...
   ENDDO
ENDDO
```

4.3 Assignment Replication

Assignment replication calculates the values of an array variable when more than one such calculated value is required in a single iteration. For example:

```
DO I=1,N
```
$A_I = fa(C_I, ...)$
$B_I = fb(A_I, \bar{A}_I, ...)$
...
```
ENDDO
```

Since there are two reads of A, A_I and $\bar{A}_I$, in the assignment of B_I, transformations such as alignment cannot be used to remove the loop-carried dependences. However if the calculation of A is replicated, transformations can be applied to each statement individually.

```
DO I=1,N
```
$A_I = fa(C_I, ...)$
```
C replicate calculation of A(I)
```
$AA_I = fa(C_I, ...)$
$B_I = fb(A_I, AA_I, ...)$
...
```
ENDDO
```

4.4 Expression Substitution

Expression substitution eliminates a reference to a particular variable in an expression by substituting the calculation of the value of that variable. It is used to shrink recurrence cycles, and may also increase the distance in the recurrence. For example:

```
DO I=1,N
```
$A_I = fa(B_I^{new}, C_I, ...)$
$B_I = fb(A^{new}[fi(I)], C_I, ...)$
...
```
ENDDO
```

By substituting the expression for A into that for B we obtain:

```
DO I=1,N
```
$A_I = fa(B_I^{new}, C_I, ...)$
$B_I = fb($
$fa(B_I^{new}, C_I, ...) \mid^{I}_{fi(I)},$
$C_I, ...)$
...
```
ENDDO
```

4.5 Loop Distribution

Loop distribution splits a loop which can not otherwise be parallelized into a sequence of one or more loops, some of which can be parallelized.

The general form for a split into two loops is:

```
DO I=1,N
```
$A_I = fa(B_I^{old}, C_I, ...)$
$B_I = fb(A_I^{new}, C_I, ...)$
...
```
ENDDO
```

```
DO I=1,N
```
$A_I = fa(B_I^{old}, C_I, ...)$
```
ENDDO
DO I=1,N
```
$B_I = fb(A_I^{new}, C_I, ...)$
...
```
ENDDO
```

The criteria for correctness of the transformation is that:

- for all A assigned in the first loop, reads of A in the second loop must be *NEW*, and
- for all B assigned in the second loop, reads of A in the first loop must be *OLD*.

Other transformations such as unrolling, fusion, skewing and interchange [18], are all inter-loop transformations. Inter-loop transformations can be used to transform a program to generate the *largest parallelizable loop* for that program. These transformations are interrelated, and like intra-loop transformations can be applied in various orders to yield different loop structures (to which intra-loop transformations can subsequently be applied to yield maximally parallel loop structures). Some methods have been developed for combining sequences of inter-loop transformations [17], when dependence distances or directions are known to maximize the parallelism. Other authors [13] have developed algorithms for loop nests which focus upon optimizing data locality. The interaction between these approachs, and our approach for extracting maximum parallelism from a single loop, are an open research issue. We discuss the interaction between these approaches later.

Fortunately, in practice most programs have a relatively simple loop structure, so that only a few cases of loop-interchange, fusion, or skewing are feasible. The following analysis assumes that such inter-loop transformations have been performed, although the model could be embedded in a system which analyzed alternative loop structures.

4.6 Program Transformation Equivalences

To prove that two programs are equivalent under a transformation we need to prove that the transformation preserves dependences and that each assignment in the program is equivalent to an assignment in the other program. Preservation of dependences guarantees that the same values reach any assignment.

To show preservation of dependences, each read in the original program needs to be labeled as either *OLD* or *NEW*.

For the transformed version of the dependence direction labeled program to be equivalent to the original program we need to show:

- Every read access, A_I^{old} is not read after an assignment to the same element of A
- Every read access, A_I^{new} is not read before the assignment to the same element of A

Theorem: The transformations value replication, assignment replication, expression substitution loop distribution all yield equivalent programs under the restrictions imposed by the definitions of the transformations above.
Proof: Each of these transformations preserves the dependence directions of the original program, and by substitution each of the assignments in the transformed program calculates the same expression as the original program.

However, alignment does not necessarily preserve dependences. However, a sequence of alignments will preserve dependences under two conditions:
Theorem: If an alignment is applied to each assignment in a set, $Align\_Set = \{A, B, ...\}$, so that:
For each A in $Align\_Set$:

- every read access, A_I^{old} or A_I^{new}, are in the same iteration as the assignment to the same A_I.
- a statement with a read access A_I^{old} precedes the assignment to A_I, and a statment with a read access A_I^{new} follows the assignment to A_I in the same loop.

then the resulting program is equivalent to the program before the alignments were applied.
Proof: Like the other transformations, alignment does not change the expressions calculated by an assignment. The criteria in the theorem above merely state that there are no loop-carried dependences within the $Align\_Set$ after the alignment transformation, but the intra-iteration dependences preserve the original dependence directions.

5 A Modified Dependence Graph

We have found it necessary to define a specialized 'dependence graph' to support generation of sequences of the above transformations. In particular, replication requires that dependences be maintained between individual variable references rather than between statements. Also, since we consider loops individually we can dispense with the complexity needed to maintain dependences on multiple nested loops simultaneously. The IVD graph includes information about the relationships between dependences sequences, such as whether a statement is in a cycle. We refer to this modified graph as an *intervariable dependence graph* (IVD graph).

A node in the IVD graph correspond to an array write in the loop, hence there is a unique node for each statement in the loop. An edge from A to B ($A \rightarrow B$) implies A is referenced in an assignment of B, eg. $B[I] = f(A[I-1], C_I, ...)$.

The *distance* of an edge is the difference between the subscript of the source reference (a *READ*) and the subscript of the *WRITE* of that variable. If $A[I]$ was written in a loop containing the above assignment, the edge $A \rightarrow B$ would have distance "-1", and *direction* $<$ (negative, a new value of A computed in a prior iteration). Edges are labeled with distances (or directions if distances are not constant). Nodes are labeled with the array being assigned, or equivalently the assignment statement. the alignment of the statement).

Following is an example of an IVD graph. The following sections will show how it is transformed

DO I = 1,N
$A[I] = fa(C[I-1], B[I-1], E[I+1])$
$B[I] = fb(C[I-1], D[I-1])$
$C[I] = fc(I)$
$D[I] = fd(A[I+1], D[I-2])$
$E[I] = fe(G[I-1])$
$F[I] = ff(E[I-1])$
$G[I] = fg(I)$
ENDDO

Example 1: Serial Program

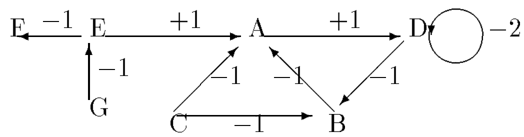


Diagram 1 : IVD Graph Before Transformation

Following sections describe an IVD graph formally, and the reduction of an IVD graph for program transformation.

5.1 Structure of the IVD Graph

An IVD graph contains directed edges and nodes identified as TREE, DAG or CYCLE nodes. Each TREE or DAG node is contained in a unique maximal TREE or DAG, and each CYCLE node is contained in one or more CYCLES.

The classification of nodes is used to determine the type of transformations to apply, and in what order.

The classification is based upon the graph structure (not the dependence distance or direction labels):

- Each node that appears in a directed cycle in an IVD graph is classified as a CYCLE node, and may be in more than one cycle.
- A DAG node is any node that is not a CYCLE node and that appears in a cycle ignoring the direction of the edges. A DAG node is in a unique maximal DAG.
- A TREE node is one which appears in a subtree of the graph. Any node that is not a CYCLE or DAG node is a TREE node. The root of a subtree may be a CYCLE or DAG node. A TREE node is in a unique tree.

An IVD graph, $\mathbf{G_{IVD}}$, consists of a tuple $\{\mathbf{NODES}, \mathbf{EDGES}, \mathbf{T}, \mathbf{D}, \mathbf{C}\}$, where:

NODES is the set of nodes, one for each assignment statement in the loop.

EDGES is the set of edges, one for each dependence between variables assigned in the loop. Some transformations, such as distribution, create new loops. Dependences between loops are represented as *loop edges* (loop independent dependences). The sequence of transformations applied to reduce an IVD graph generates such loop edges. However, loop edges generated do not affect the choice of transformations, only the final regeneration of a program from an IVD graph. The subset of **EDGES** that does not include loop **EDGES** is denoted by $\overline{\mathbf{EDGES}}$. Initially, $\overline{\mathbf{EDGES}} = \mathbf{EDGES}$.

T is the set of maximal trees in $\mathbf{G_{IVD}}$. A tree is maximal if it is not contained in any other tree.

C is a set of cycles in $\mathbf{G_{IVD}}$.

D is the set of maximal DAGs in $\mathbf{G_{IVD}}$. A DAG is maximal if it is connected and is not contained in any other DAG.

Trees are defined ignoring the direction of edges in the IVD graph. Observe that there can be no paths (undirected) between leaf nodes in the tree that are not within the tree (such paths can cause the tree to become a DAG when cycles are broken). Formally, a tree $\mathbf{t}$ consists of a triple $\{\mathbf{t}_{Leaves}, \mathbf{t}_{Nodes}, \mathbf{t}_{Edges}\}$ where:

- $\mathbf{t}_{Leaves}$is the set of leaf nodes of the tree, leaf nodes can either be CYCLE or DAG nodes, or leaf nodes in the IVD graph itself.
- $\mathbf{t}_{Nodes}$are the interior nodes in the tree
- $\mathbf{t}_{Edges}$consists of all edges in $\overline{\mathbf{EDGES}}$ to or from nodes in $\mathbf{t}_{Nodes}$

$\mathbf{t}$ must have the following properties:

- $\{\ \mathbf{t}_{Leaves} \cup \mathbf{t}_{Nodes}, \mathbf{t}_{Edges}\}$ is a tree
- $\mathbf{t}$ is not contained in any other DAG or tree $\bar{\mathbf{t}}$
- All paths in $\overline{\mathbf{EDGES}}$ between nodes in $\mathbf{t}_{Leaves}$are in $\mathbf{t}_{Edges}$

It follows that all trees $\mathbf{t}$ in $\mathbf{T}$ are disjoint.

A cycle $\mathbf{c}$ consists of a tuple $\{\mathbf{c}_{Nodes}, \mathbf{c}_{Edges}\}$ where:

- $\mathbf{c}_{Nodes}$is a set of $k > 1$ nodes: $\{\mathbf{n}_1, \mathbf{n}_2, \ldots \mathbf{n}_k\}$.
- $\mathbf{c}_{Edges}$is a set of k edges in $\overline{\mathbf{EDGES}}$ between the nodes in $\mathbf{c}_{Nodes}$ $\{\mathbf{n}_1 \rightarrow \mathbf{n}_2 \rightarrow \ldots \rightarrow \mathbf{n}_k \rightarrow \mathbf{n}_1\}$

There can be edges to nodes in $\mathbf{c}_{Nodes}$not in $\mathbf{c}_{Edges}$ from DAG, TREE or other CYCLE nodes or from nodes in $\mathbf{c}_{Nodes}$ to DAG, TREE, or other CYCLE nodes.
A CYCLE node has 1 or more successors which are CYCLE nodes and 0 or more successors which are DAG or TREE nodes.
A DAG $\mathbf{d}$ is an acyclic graph consisting of a quadruple $\{\mathbf{d}_{Roots}, \mathbf{d}_{Nodes}, \mathbf{d}_{LeafNodes}, \mathbf{d}_{Edges}\}$ where:

- The three sets of nodes: $\mathbf{d}_{Roots}$, $\mathbf{d}_{Nodes}$, $\mathbf{d}_{LeafNodes}$ are disjoint
- $\mathbf{d}_{Roots}$ is the set of nodes which are the roots of the DAG. These nodes can either be DAG or CYCLE nodes.
- $\mathbf{d}_{Nodes}$ is the set of DAG nodes, reachable along a path in $\mathbf{d}_{Edges}$.
- $\mathbf{d}_{LeafNodes}$ is the set of DAG or CYCLE nodes, reachable along a path in $\mathbf{d}_{Edges}$.
- $\mathbf{d}_{Edges}$ is the set of edges in $\overline{\mathbf{EDGES}}$ from nodes in $\mathbf{d}_{Roots}$ to nodes in $\mathbf{d}_{Nodes}$ or $\mathbf{d}_{LeafNodes}$, between the nodes in $\mathbf{d}_{Nodes}$, and from nodes in $\mathbf{d}_{Nodes}$ to nodes in $\mathbf{d}_{LeafNodes}$.

Since DAGs are connected, for all $\mathbf{n}_0$ in $\mathbf{d}_{Nodes}$ there is a $\mathbf{n}_1$ in $\mathbf{d}_{Roots}$ such that there is a path in $\mathbf{d}_{Edges}$from $\mathbf{n}_1$ to $\mathbf{n}_0$.

Also, for all $\mathbf{n}_0$ in $\mathbf{d}_{Roots} \bigcup \mathbf{d}_{Nodes}$ there is a $\mathbf{n}_1$ in $\mathbf{d}_{LeafNodes}$ such that there is a path in $\mathbf{d}_{Edges}$ from $\mathbf{n}_0$ to $\mathbf{n}_1$.

Complete algorithms for constructing the sets **T**, **C**, and **D** from the set of **NODES** and $\overline{\mathbf{EDGES}}$ for an IVD graph can be found in [3].

For the example program above, the sets are:

- CYCLE nodes: $\{A, B, D\}$, $\mathbf{C} = \{\{A, D, B\}, \{D\}\}$
- DAG nodes: $\{C\}$, $\mathbf{D} = \{\{A, C, B\}\}$
- TREE nodes: $\{E, F, G\}$, $\mathbf{T} = \{\{A, E, F, G\}\}$

The transformations which are applied to the graph yield modified IVD graphs. These transformations can result in the creation of new nodes (for replication), creation and deletion of edges and relabeling of edges (for alignment). Value replication is implemented by creating a new loop to replicate values. Dependences between loops are represented as loop edges in the graph which are shown as thick edges in the diagrams. The final result of IVD transformations for the example below is the following graph (nodes are annotated with their alignment):

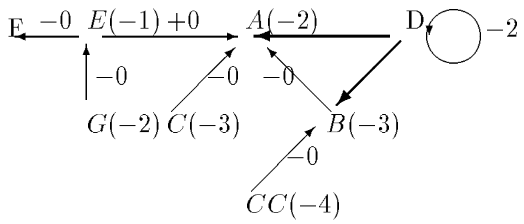


Diagram 2 : IVD Graph After Transformation

The derivation of this graph is explained in detail in the sections below.

The IVD Graph representation of dependences is tailored to the transformations we wish to apply to a loop body. It omits information not required by our analysis (such as direction vectors for nested or containing loops [18], or transitive dependences) and provides additional information required, including specific references rather than statements for the source and sink of a dependence, and distinguishing a node's type which is useful in selecting parallelizing transformations. It is extensible to special cases including constant subscripts, control branches, and nested loops [3].

5.2 Reconstruction of a Program from the IVD Graph

Reconstructing a program from an IVD graph is similar to algorithms for vectorization based upon partitioning the dependence graph into strongly connected components (PI-blocks) [19, page 206]. However, the IVD graph has already undergone transformations so that dependences which require loop distribution have been identified (loop edges).

Reconstructing a program from an IVD graph requires that some additional bookkeeping information be kept at each node, namely:

- The function used to calculate the right-hand side of the assignment for the node
- The array assigned by the node
- The alignment applied to the node

The algorithm for reconstructing a program from a graph is as follows:
Algorithm: IVDgraph_to_Program

- Partition the IVD graph into *loop subgraphs*, where each subgraph is connected, and none of the edges within the subgraph are loop edges.
- The partition induces a *loop graph*, where each node in the loop graph is a loop subgraph, and the edges in the loop graph are loop edges. The loop graph must be a DAG by construction, as shown below. Nodes in the loop graph are either SSRs (or cycles of SSRs), or subgraphs in which the distances on each edge are "+0" or "-0", or replication nodes.
- Depth label the loop subgraph
- For each depth level d
  - Merge all the subgraphs at that level that do not contain SSR's into a single loop subgraph **G**.
  - If there are no subgraphs containing an SSR, generate a single PARALLEL DO for the merged subgraph using the algorithm IVDgraph_to_Parallel_Do
  - Else if there is a single SSR subgraph, generate a serial DO loop, using the algorithm Node_to_Statement
  - If there is more than one node at a given depth in the loop graph, generate a PARALLEL SECTION, in which each case is a serial DO loop for that SSR, and one case may be a merged subgraph not containing SSRs.

Theorem: The algorithm IVDgraph_to_program generates a program with maximal parallelism
Proof: To prove that there is maximal parallelism we need to show that no parallel loops can be fused together, and that the only remaining serial loops are

single statement recurrences. The proof that no parallel loops can be fused together follows because all loops at the same level of the loop graph are fused. If two parallel loops are at different levels then there must be a path of loop edges between them in the loop graph. If the path is a single edge then the source of the edge must be from a replication loop. By construction, replication loops have a dependence to their successor. If the path is longer than length one edge then their must be an SSR loop node on the path. If the parallel loops are merged then one of the dependences between the SSR loop and the parallel loops will be reversed.

Algorithm: IVDgraph_to_Parallel_Do

By construction, each edge in the loop will be labeled with either +0 or −0. The nodes must be topologically sorted of the lines, so that

- The sink of any edge labeled +0 is in a statement following the statment corresponding to the source
- The sink of any edge labeled −0 is in a statement preceding the statment corresponding to the source

This preserves the dependences.

A topological sort exists provided their are no cycles with all edges labeled either +0 or −0. This is impossible as it implies a dependence cycle within a single iteration of the loop.

After the lines are sorted, a PARALLEL LOOP is generated with bounds extended as follows: the maximum positive alignment of statements (if any) in the loop is subtracted from the lower bound, and the the maximum negative alignment of statements (if any) in the loop is added to the upper bound. Code for each statement is generated by the algorithm Node_to_Statement.

Algorithm: Node_to_Statement

Generation of a statement is straightforward if no alignments have been applied: simply substitute the node labels of incoming edge into the expression tree for the node. If any alignment is needed, initially the statement is generated in the if protected form of alignment. If any alignment has been applied, every statement (whether aligned or not) needs to be if protected. If the original loop bounds were 1 and N, the if protection for a statment aligned by k (possibly zero) is: IF($I \geq 1-k$ AND $I \leq N+k$) ...

For the earlier example program, applying the algorithm IVDgraph_to_Program, to the IVD graph after transformation (Diagram 2), yields the following program:

```
DO I = 1,N
  D[I] = fd(A[I+1], D[I-2])
ENDDO
PARALLEL DO I = 1,N+4
  IF(I >= 3 AND I <= N+2) G[I-2] = fg(I-2)
  IF(I >= 2 AND I <= N+1) E[I-1] = fe(G[I-2])
  IF(I <= N) F[I] = ff(E[I-1])
  IF(I >= 5) CC[I-4] = fc(I-4)
  IF(I >= 4 AND I <= N+3)
    B[I-3] = fb(CC[I-4], D[I-4])
  IF(I >= 4 AND I <= N+3) C[I-3] = fc(I-3)
  IF(I >= 3 AND I <= N+2)
    A[I-2] = fa(C[I-3], B[I-3], E[I-1])
ENDDO
```

Example 1: Parallel Program

The IVD graph in diagram 2 had two loop subgraphs: the SSR containing the assignment to D, and the parallel loop containing the other assignments. The assignments are ordered based upon edge labels (+0 or −0), and the alignments are applied to the statements in the original example.

As noted earlier, iterations 1, 2, 3, 4, N+1, N+2, N+3, and N+4 could be peeled to eliminate the conditional statements with the parallel loop.

5.3 Reducing an IVD Graph

Our primary concern is the reduction of an IVD graph and the corresponding transformation of a program. In our model, maximal parallelism is arrived at as follows:

1. reduce all CYCLEs to DAGs, breaking edges either by replicating 'old' values (edges with positive distances) or by substituting inline expressions assigned in the loop. All remaining loops will be SSRs, or cycles of SSRs, and edges between them and other nodes in the IVD graph will be loop edges.
2. reduce DAGs to TREEs as necessary by replicating 'old' values or by replicating 'new' values (i.e. copying assignments).
3. reduce TREEs by aligning subscripts and then ordering lines via a depth-first traversal of the remaining forest.
4. generate a program from the graph, using the algorithm IVDgraph_to_Program earlier.
5. apply local optimizations to the generated program

Thus, the overall algorithm for reducing an IVD graph $\mathbf{G_{IVD}}$, consisting of a tuple {**NODES**, **EDGES**, **T**, **D**, **C**} is as follows:

Algorithm: Reduce_IVDgraph
Reduce Cycles:
if **C** is non-empty then
Reduce_Cycles(POSITIVE_EDGES)
if **C** is non-empty then
Reduce_Cycles(NEGATIVE_EDGES)
Reduce DAGS:
for each **d** in **D**
Reduce_Dag(**d**)
Align Trees:
for each **t** in **T**
Align_Tree(**t**)

Each of the three algorithms used in Reduce_IVDgraph steps are described below. Complete algorithms are given in [3].

There are a few local optimizations that can be applied to the generated program:

- Optimization of alignment.

  The conditional form of alignment, page 10, is the simplest to generate but least efficient. This form can be optimized by peeling all iterations that have conditionally executed statements.

- Optimization of replication.

  Replication can be optimized by merging the replication loop with the parallel loop that uses the replicated values using strip mining as described on page 5.

Sometimes replicated values can be eliminated altogether, if there is a transitive dependence caused by a loop edge. For example, the IVD graph generated for the program on page 7 contains the following structure:

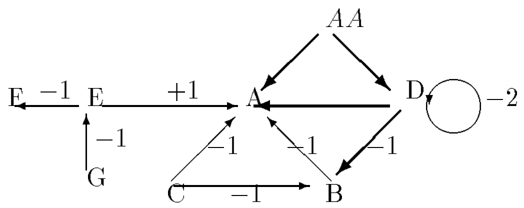


Diagram 3 : IVD Graph After Reduction of Cycles

The loop edges from AA to A and D are generated when the dependence $A \overset{+1}{\to} D$ is broken by replicating A (represented by AA) as explained below in section 5.3.2.

Because D is an SSR, the node D cannot be parallelized after the cycle $D \to B \to A \to D$ is broken. Thus, there is no need to replicate A for the serial loop D, and the dependence from AA to A can be deleted as it is transitive.

5.3.1 Reducing Cycles

In converting cycles to DAGs and trees, the lowest cost transformation is value replication. Determining the minimum number of nodes which must be replicated to break all cycles is an NP-complete problem; it is equivalent to determining the minimum number of edges which must be removed in converting an arbitrary graph to a DAG [11, page 192]. The heuristic of choosing nodes in the maximum number of cycles is a simple but effective approach.

Expression inlining can only shrink cycles to SSRs, or to cycles of nodes, each of which is an SSR. Cycle shrinking has a relatively high overhead, but minimizes the number of statements in sequential loops and increases fine-grain parallelism. If the distance in the recurrence is $-k$, then k iterations can be run in parallel. Reducing cycles to SSRs allows them to be run in PARALLEL SECTIONs with other SSRs.

A key concern is reducing intersecting cycles to disjoint cycles. *Intersecting cycles* have at least one edge in common. If one cycle completely contains the other, reducing the edges in both cycles will reduce the smaller to an SSR contained in the larger cycle. Otherwise, there is a DAG with ends in both cycles; breaking the DAG will break one of the cycles.

The algorithm for reducing CYCLES to DAGs is parameterized by whether the edges to be removed have positive distances or negative.

The algorithm first try to remove all cycles by calling the algorithm Reduce_Cycles(Positive), given below. The algorithm tries to break edges in cycles with positive distances, preferring edges in multiple cycles. Any remaining cycles are recurrences that cannot be broken, only shrunk. The same algorithm is applied to select edges for reducing multiple cycles by inlining expressions.

Before the algorithm is executed a check is made for any false CYCLES, in which all distances in the cycle are either $+0$ or -0, such cycles can be classified as TREES (they have no loop carried dependences, but may need to be aligned collectively with other tree nodes to which the cycle is connected).

After the algorithm has been applied, any remaining cycles (either SSRs or cycles of SSRs) need to be distributed into serial loops. This is done by simply converting each edge in the IVD graph to or from a remaining cycle into a loop edge. There will be no loops of such loop edges (such loops would be cycles in the IVD, which have already been shrunk. The example in diagram 2 on page 9 shows this. The SSR containing D has been distributed.

Algorithm: Reduce_Cycles (edge_type):

let **C** be the set of all cycles
let C be set of all nodes in cycles
let E be set of all edges in cycles between nodes in C, with sink not an SSR and distance is of edge_type (E is the 'reducible edges')
let MAX_CYCLES_{nodes} be max. # of cycles containing any node in C
let MAX_CYCLES_{edges} be max. # of cycles containing any edge in E
let C(I), I = 1..MAX_CYCLES_{nodes} be subsets of C whose members are all in I cycles
let E(I), I = 1..MAX_CYCLES_{nodes} be subsets of E whose members are all in I cycles
and whose direction is the same as edge_type
define rank(c) for a node c in C : I such that c is in C(I)
define rank(e) for an edge e in E : I such that e is in E(I)
while ($E \neq \{ \}$) do
  Select the set E' of edges in maximal cycles that point to nodes in maximal cycles:
  let N = MAX_CYCLES_{nodes}, $E' = \{ \}$
  while ($N \geq 1$ and $E \neq \{\}$) do
    if C(N) is not empty then
      let N2 = MAX_CYCLES_{edges}
      while ($N2 \geq 1$ and $E' = \{\}$) do
        let E' be subset of $E(N2)$ such that for $E1$ in E', E' has a sink which is in $C(N)$.
        N2 = N2 - 1
    end do
    N = N - 1
  end do
  Select an edge of E' with a sink of minimum rank:
  let $E1$ be element of E' whose source has minimum rank.
  if multiple edges of same rank, let $E1$ be edge of that rank to node with minimum distance from root.
  reduce edge E1:
  if edge_type = POSITIVE_EDGE
    call Delete_Cycle_Edge
  else
    call Reduce_Cycle_Edge
  recompute **C** , C, E, C(I)s, E(I)s.
end do

Although the above algorithm is only a heuristic it works well in practice for reducing cycles. When applied to the example earlier, only one edge has a positive direction ($D \rightarrow A$), and hence is selected for reduction first. The remaining edge in a cycle ($D \rightarrow D$) is an SSR, which cannot be reduced as shown by the algorithm below.

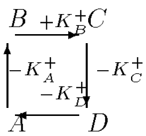
5.3.2 Reducing a CYCLE Edge, $A \rightarrow B$

Reducing an individual CYCLE edge can be done either by value replication or by expression substitution. Value replication is preferable, since it breaks the cycle and allows the code to be completely parallelized. These two algorithms are given below:

Algorithm: Reduce_Cycle_Edge : $A \rightarrow B$
  In assignment to B replace references to variable A with expression from assignment to A.
  for each edge $D \rightarrow A$
    add an edge from $D \rightarrow B$ with distance = sum of distances $D \rightarrow A \rightarrow B$;
    if $D == B$ mark B as an SSR.
    for each cycle containing $D \rightarrow A \rightarrow B$
      add a cycle replacing $D \rightarrow A \rightarrow B$ with new edge $D \rightarrow B$
      delete old cycle.
  delete edge $A \rightarrow B$
  if A is in no other cycles then
    if all remaining successors of A are TREE nodes with no other predecessors, make A a TREE node.
  else
    make A a DAG node;
    if A has no DAG predecessors, add A as a DAG root
    for each DAG successor of A, if they are classified as DAG roots
    reclassify them as non-root nodes.

For example, consider the following program fragment and its IVD graph:

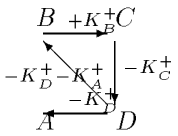
Program
DO I = 1,N
$$C[I] \;=\; fc(B[I+K_B^+],...)$$
$$B[I] \;=\; fb(A[I-K_A^+],...)$$
$$A[I] \;=\; fa(D[I-K_D^+],...)$$
$$D[I] \;=\; fd(C[I-K_C^+],...)$$
ENDDO



The corresponding program fragment and its IVD graph, after expression substitution for A into B is:

After Expression Substitution

```
DO I = 1,N
```
$$
\begin{aligned}
C[I] &= fa(B[I+K_B^+],\ldots) \\
B[I] &= fb(\\
& \quad fc(D[I-K_D^+],\ldots)\,|^{I}_{I-K_A^+}, \\
& \quad \ldots) \\
A[I] &= fc(D[I-K_D^+],\ldots) \\
D[I] &= fd(C[I-K_C^+],\ldots)
\end{aligned}
$$
```
ENDDO
```



The algorithm for breaking cycles is

Algorithm: Delete_Cycle_Edge:

Breaking a CYCLE edge $B \rightarrow C$ by value replication.

call Replicate_Value(B,C)
update the IVD graph node classification
for each cycle containing $B \rightarrow C$
  for each X in $\{B$ and predecessors along path of cycle back to $C\}$
      (B, A, D, and C in the example below)
    if X is in no other cycle then
      if all remaining successors of X are TREE nodes with no other predecessors, then
      make X a TREE node.
      else
        make X a DAG node
        if X has no DAG predecessors, add X as a DAG root
        for each DAG successor Y of X
          if Y is classified as a DAG roots then reclassify Y as a non-root node.
        end for
  end for
  delete cycle.
end for

The algorithm Replicate_Value(B, C) makes a copy of the value of B, into BB, which is represented by the following modifications to the IVD graph:

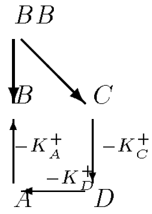
Algorithm: Replicate_Value(B, C)

create a new node BB, labeled with alignment 0 and the assignment $BB[I] = B[I]$
add a loop edge from BB to B and BB to C
delete edge $B \rightarrow C$

The algorithm for breaking cycles by replication can be illustrated with the same example above. The cycle would have been broken by replicating the value of the node which is the source of an edge with positive weight: $B \rightarrow C$:

Program After Replication

```
DO I = 1,N
```
$$BB[I] = B[I+K_B^+]$$
```
ENDDO
DO I = 1,N
```
$$
\begin{aligned}
C[I] &= fa(BB[I],\ldots) \\
B[I] &= fb(A[I-K_A^+],\ldots) \\
A[I] &= fc(D[I-K_D^+],\ldots) \\
D[I] &= fd(C[I-K_C^+],\ldots)
\end{aligned}
$$
```
ENDDO
```



Notice that value replication adds two loop edges, because of the dependence from BB to C and the anti-dependence from the read of B in the replication loop to the write of B in the subsequent loop.

5.3.3 Reducing a DAG

Reduction of a DAG involves elimination of redundant paths between two nodes. This can be done either by deleting an edge (value replication) or by replacing it with an edge to or from a new node (assignment replication), depending on the direction of the edge. The first check that needs to be made is for paths of identical distance. If there are two paths from node A to B with the same distance then no attempt needs to be made to eliminate the duplicate paths. The algorithm for aligning trees handles DAGS, assuming that redundant paths have identical distance.

Since a TREE of M nodes always has $M-1$ edges, a DAG of N nodes and E edges can always be converted to a TREE by removing exactly $E-N+1$ edges from the graph. In general, the lowest cost transformation is value replication, and positive edges can be removed by this means. Once positive edges have been removed to reduce the connectivity of the DAG, negative edges must be removed by assignment replication of each subtree via bottom-up recursion. (This has exponential cost when done inside the loop [7]. However, if assignments are split into separate parallel loops, enforcing a barrier synchronization between the expressions, the exponential cost is reduced to a linear cost). If two paths to a node in the DAG from a common dominating ancestor have the same dependence distance, then the second path is redundant (and need not be replicated).

The following algorithm is applied to convert each DAG subgraph in an IVD graph with cycles removed:

Algorithm: Reduce_Dag($\mathbf{d}$):

Let $\mathbf{d}$ be a tuple $\mathbf{d}_{Roots}$, $\mathbf{d}_{Nodes}$, $\mathbf{d}_{LeafNodes}$, $\mathbf{d}_{Edges}$ as described above.

$\mathbf{d}_{LeafNodes}$ consists only of TREE nodes (by construction).

Traverse the DAG, 'distance labeling' each node;
distance is maximum distance from any node
  in $\mathbf{d}_{Roots}$.
Order $\mathbf{d}_{LeafNodes}$ by distance.
while $\mathbf{d}_{LeafNodes}$ is not empty
  let **n** be a node in $\mathbf{d}_{LeafNodes}$ with maximum distance.
  if **n** has only one predecessor **m** in $\mathbf{d}_{Nodes}$ then
    convert **m** to a TREE node
    remove **m** from $\mathbf{d}_{Nodes}$
    add it to $\mathbf{d}_{LeafNodes}$
  else
    n *has more than one predecessor*
    let $\mathbf{d}_{JoinNodes}$ be the the set of nodes in $\mathbf{d}_{Nodes}$
    from which there are are two or more paths
      with unequal distances to **n**
    while $\mathbf{d}_{JoinNodes}$ is not empty
      Let **m** be a node in $\mathbf{d}_{JoinNodes}$.
      while there is a path from **m** to **n**
      with an edge $A \rightarrow B$ of positive distance
        delete edge
        call Replicate_Value(A,B)
      end while
    if there are multiple paths remaining to **n**
    all edges on these paths must be negative
    while there are $k > 1$ paths remaining
      let **m** be a predecessor of **n**
      delete a predecessor
      call Replicate_Subtree(**m**, **n**)
    end while
  remove **n** from $\mathbf{d}_{LeafNodes}$.
end while

Replicate_Subtree(**m**, **n**) makes a copy of the tree in the IVD graph rooted at **n**, then adds an edge from **m** to this new tree, then deletes the edge **m**$\rightarrow$ **n**. By construction, **n**must be a TREE node.

Theorem: Applying Reduce_Dag to a graph with no cycles yields an optimal sequence of replications for the corresponding loop.

Proof: Any reduction of a DAG to a tree requires removing the same number of edges. The algorithm always chooses the least cost edges to remove by replication. If there are cycles with edges all of which are negative, the only transformation that can be applied is replication of subtrees.

5.3.4 Reducing a DAG to a TREE

Reduction of a DAG involves elimination of redundant paths between two nodes. This can be done either by deleting an edge (value replication) or by replacing it with an edge to/from a new node (assignment replication), depending on the direction of the edge.

Theorem: A value replication corresponds to deleting an edge of positive weight in the dependence graph, and adding a loop edge for the replicated value.
This was shown earlier for value replication for reducing cycles.

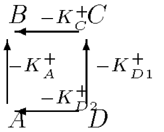
The converse also holds, that is deleting an edge of positive weight is equivalent to replication

Theorem: An assignment replication corresponds to replicating a subtree of the IVD graph.

The following example illustrates the general case, where all edges on the paths from D to A have negative weight:

Program

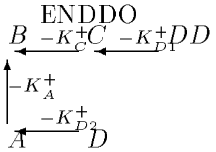
DO I = 1,N
$$C[I] = fc(D[I - K^+_{D1}], ...)$$
$$B[I] = fb(A[I - K^+_A, C[I - K^+_C], ...)$$
$$A[I] = fc(D[I - K^+_{D2}], ...)$$
$$D[I] = fd(...)$$
ENDDO



For the earlier definition of assignment replication, on page 5, program and the corresponding IVD graph after replication of the node D is:

Program

DO I = 1,N
$$C[I] = fc(DD[I - K^+_{D1}], ...)$$
$$B[I] = fb(A[I - K^+_A, C[I - K^+_C, ...)$$
$$A[I] = fc(D[I - K^+_{D2}], ...)$$
$$D[I] = fd(...)$$
$$DD[I] = fd(...)$$
ENDDO



The program obtained by converting the IVD graph with a replicated node D, DD above, is equivalent the program obtained by replicating the assignment.

Replication must be recursive, as if there is an edge $E \rightarrow D$, with distance k, then after replication E becomes a new join point in the graph, and the algorithm must be applied again. In our algorithm, replication is performed bottom-up, so that when D is replicated it is always a TREE node.

5.3.5 Reducing a TREE graph

The simplest series of transformations is that required by a loop represented as a TREE graph. In this case, a series of alignments are sufficient to parallelize the loop. Alignments can be used to convert all dependence distances to $+0$ or -0. We reduce a node of a TREE graph in the following manner.

Algorithm: Align_Tree(**t**)
 Mark all nodes in the tree **t**as being unaligned.
 Let **n**be a root of the tree **t**
 call Align_Node(**n**, 0)

Actually, any node of the tree could be chosen. The relative alignment of statments remains the same, although the loop bounds and expressions used in the if protection of statements are offset.

Align_Tree calls the recursive algorithm Align_Node:

Algorithm: Align_Node(**n**, alignment):
 Mark **n** as aligned, and
  set **n**.alignment = alignment
 for each node **m** which is a successor or predecessor
  of **n**that has not been marked as aligned
  add or subtract edge label (dependence distance)
  from predecessors and successors respectively
  if **m** is a predecessor of **n** then
   call Align_Node(**m**, alignment + edge label)
  else
   call Align_Node(**m**, alignment − edge label)
 end for

Theorem: Align_Node is equivalent to an alignment.

Proof: Let the node being aligned by be B:

$$B[I] = fb(A[I-K_A], C[I-K_C], ...)$$

Let D be a successor of B, and A be a predecessor of B in the graph:

$$D[I] = fd(B[I-K_B], ...)$$
$$A[I] = fa(C[I-K_C], ...)$$
$$C[I] = fc(...)$$

The corresponding IVD graph is:

$$\begin{tikzcd} & C \arrow[d, "-K_C"] & \\ D & B \arrow[l, "-K_B"'] & A \arrow[l, "-K_A"'] \end{tikzcd}$$

After alignment of B by K_A:

$$B_I \,|^I_{I+K_A} = fb(A[I-K_A], C[I-K_C], ...) \,|^I_{I+K_A}$$

By definition, the weight of the edge from T's parent to T is the difference in the subscripts of the assignment of the parent variable and the reference to that variable in the assignment to T.

The new dependence graph is:

$$\begin{tikzcd} & C \arrow[d, "-K_C+K_A"] & \\ D & B \arrow[l, "-K_B+K_A"'] & A \arrow[l, "\pm 0"'] \end{tikzcd}$$

The edge from A to B is -0 if the original dependence direction from A to B was $-$ve, and $+0$ otherwise.

In section 4.6 we gave a set of criteria for a sequence of alignments applied to a set of nodes $\{A, B, ...\}$ to result in an equivalent program. The criteria requires that all read accesses of the aligned nodes be in the same iteration as the writes. In IVD Graph terms this means applying alignments so that all distances are either $+0$ or -0. It is possible to do this to a subgraph **G** of the IVD Graph if and only if:

- Ignoring edge directions, **G** is a tree, or
- All cycles in the graph have zero distance, when the directions and signs of edges are reversed to make a directed cycle

6 Performance Analysis

The algorithm and approach introduced above produces a maximally parallel loop when directions and distances of all dependences are known. However, to be effective it needs to incorporate a performance model to estimate the performance improvements for the maximally parallel loop if any. Such a model needs to take into account:

1. The overhead of parallel constructs, and transformed statements
2. The effects of non-local memory access and cache misses

The development of such a model is an ongoing research project, at present targeted at the KSR-1.

It is relatively easy to develop an approximate model for the overhead of parallel constructs and transformed statments [3]. The model assumes

- Loops are executed large number of times (hence the overhead of peeling iterations can be ignored).
- Static scheduling is used so that the overhead of a parallel loop is independent of the number of iterations
- The relative cost of statements (assignments and expressions) is known

Unfortunately, in practice the effects of non-local memory and cache cannot be ignored[13]. A performance model needs to take into account

- New cache lines accessed (misses) by loop iterations

- Non-local memory blocks accessed by loop iterations

Estimating these requires a machine model. Even given an accurate machine model, these factors are affected by loop scheduling strategies (e.g., blocking for locality). There are a number of intra-iteration transformations that can improve locality, such as

- Coalescing arrays

  For example, coalescing two arrays $A(N, N)$ and $B(N, N)$ into a single array $AB(2, N, N)$.

- Prefetching data (either for this iteration or for subsequent iterations)

Inter-loop transformations that can affect locality are

- Interchanging loops
- Fusing loops, or groups loops into affinity regions [4]
- Loop interchange, or exchanging subscripts
- Loop skewing (this usually degrades locality)

The interplay between transformations for locality and transformations for parallelism is an open research issue.

A further research issue is the relationship between different program representations such as PDG/SSA and representations tailored for loop-carried dependences, such as the IVD graph. We are presently working on combining the PDG/SSA representation with the IVD graph to better represent control dependences and scalar optimizations.

7 Conclusion

Our algorithm provides a systematic basis for transforming and parallelizing loops containing complex dependences. The algorithms presented have been implemented and tested in PAT and are effective in reducing previously unparallelizable code. The approach is capable of handling loops with control dependences and nested loops.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley Publishing Company, Reading, Massachusetts, 1987.

[2] Bill Appelbe, Charlie McDowell, and Kevin Smith. Start/Pat: A parallel-programming toolkit. *IEEE Software*, 6(4):29–38, July 1989.

[3] Bill Appelbe and Kevin Smith. Analyzing loops for parallelism. Technical Report GIT-ICS-90/59, Georgia Institute of Technology, November 1990.

[4] William F. Appelbe and Bala Lakshmanan. Optimizing parallel programs using affinity regions. In *International Conference on Parallel Processing*, August 1993.

[5] Vasanth Balasundarum. *Itereactive Parallelization of Numerical Scientific Programs.* PhD thesis, Rice University, June 1989. Regular Sections summarize dependences in programs.

[6] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, Massachusetts, 1988.

[7] David Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Rice University, 1987. Rice Tech Report, COMP TR87-50.

[8] David Callahan. Recognizing and parallelizing bounded recurrences. In *Fourth Workshop on Languages and Compilers for Parallel Computing*, August 1991.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[10] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 1987.

[11] Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, San Francisco, California, 1979.

[12] Alan H. Karp. Programming for parallelism. *Computer*, 20(5):43–57, May 1987.

[13] Ken Kennedy and Kathryn McKinley. Optimizing for parallelism and data locality. In *International Conference on Supercomputing*, pages 323–334, July 1992.

[14] William Pugh. The Omega test, a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91*, pages 4–13, November 1991.

[15] Kevin Smith and Bill Appelbe. Interactive conversion of sequential to multitasking FORTRAN. In *International Conference on Supercomputing*, pages 225–234, June 1989.

[16] Kevin S. Smith. *PAT: An Interactive Fortran Parallelizing Assistant Tool.* PhD thesis, Georgia Institute of Technology, December 1988.

[17] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–482, October 1991.

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

[19] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, New York, 1990.