Yale University
Department of Computer Science

Program Optimization and Parallelization Using Idioms

Shlomit S. Pinter and Ron Y. Pinter

YALEU/DCS/TR-730
August 1989

Program Optimization and Parallelization Using Idioms

Shlomit S. Pinter[1,2]
Ron Y. Pinter[3,4]

Dept. of Computer Science
Yale University
New Haven, CT 06520

August 1989

Abstract

Programs in languages such as FORTRAN, Pascal, and C, were designed and written for a sequential machine model. Several methods to vectorize such programs and recover other forms of parallelism that apply to more advanced machine architectures have been developed during the last decade. We propose and demonstrate a more powerful translation technique for making such programs run efficiently on parallel machines which offer facilities such as parallel prefix operations as well as parallel and vector capabilities. This technique, which is global in nature and involves a modification of the traditional definition of the program dependence graph (PDG), is based on the extraction of parallelizable program structures ("idioms") from the given (sequential) program. The benefits of our technique extend beyond the abovementioned architectures, and can be viewed as a general program optimization method, applicable in many other situations. We show a few examples in which our method indeed outperforms existing analysis techniques.

1 Introduction

Many of the classical compiler optimization techniques [2] comprise the application of local transformations to the (intermediate) code, replacing sub-optimal fragments with better ones. One hopes, as is often the case, that repeating this process (until a fixed-point is found or some other criterion is met) will result in an overall better program. To a large extent, attempts to vectorize — and otherwise parallelize — sequential code [20] are also based on the detection of local relationships between data items and the control structures that enclose them. These approaches, however, are local in nature and do not recognize the structure of the computation that is being carried out.

A computation structure sometimes spans a fragment of the program which may include some irrelevant details (that might obscure the picture both to the human eye and to

[1] On sabbatical leave from the Dept. of Electrical Engineering, Technion — Israel Institute of Technology.
[2] Work supported in part by NSF grant number DCR-8405478.
[3] On sabbatical leave from the IBM Israel Scientific Center.
[4] Work supported in part by ONR grant number N00014-89-J-1906.

automatic recovery algorithms). Such "idioms" [16] may include inner-product calculations in numeric code, data structure traversal in a symbolic context, and patterns of updating shared values in a distributed application. Recognizing them is far beyond the potential of the classical, local methods.

Once recognized, such structures can be replaced lock, stock, and barrel by a new piece of code that has been highly optimized for the task at hand. This pertains to sequential target machines, but is most appealing in view of the potential gains in parallel computing. For example, a loop computing the convolution of two vectors, which uses array references, can be replaced by an equivalent fragment using pointers which was tailored by an expert assembler programmer. The same code can be replaced by a call to a BLAS [8, 13] routine that does the same job on the appropriate target machine. Better yet, if the machine at hand — for example, TMC's CM-2 [19] — supports summary operators, such as reduction which works in time $O(\log n)$ using a parallel prefix implementation [4, 12] rather than $O(n)$ on a sequential machine (where n is the length of the vectors involved), the gains could be even more dramatic.

In this paper we propose a method for extracting parallelizable idioms from scientific programs. We cast our techniques in terms of the construction and analysis of the *computation graph*, which is a modified extension of the program dependence graph (PDG) [9], as necessary for the required analyses. We describe how to form this new type of graph from a source program and provide algorithms that use it for optimization transformations. Using this method, we were able to extract structures that other methods fail to recognize, as reported in the literature, thereby providing a larger potential for speed-up. We believe that our techniques can also be fitted to other program analysis frameworks, such as symbolic evaluation and plan analysis.

Our technique is more involved than just symbolic manipulations and the look-up of patterns. Its power can be demonstrated by a small example, taken from [3], who gave up on trying to parallelize the following loop (which indeed cannot be vectorized):

```
      DO 100 I=1,N
      C(I)=A(I)+B(I)
      B(I+1)=C(I-1)*A(I)
100   CONTINUE
```

Known methods, and even a human observer, may not realize that this loop hides two scan operations that can be implemented in time $O(\log n)$ (n being the value of `N`) rather than $O(n)$. This kind of transformation can, however, be effected using our computation graph.

We first review other work on dependency analysis and evaluate its limitations for purposes of idiom extraction. Then, in Section 3, we provide a formal presentation of our technique, including the necessary algorithms. In Section 4 we exemplify our techniques, stressing the advantages over existing methods. Section 5 discusses applications other than parallelism and proposes further research.

2 Existing Optimization and Parallelization Methods

Three major program analysis methods for purposes of optimization and parallelization are most common: constructing and then using the program dependency graph (PDG), setting up and then solving a system of linear equations that reflect the relationship between array references, and (partial) symbolic evaluation. In this section we briefly review each method, reason why in itself none can be used to serve our purposes, and explain why we have decided to extend the PDG approach.

Much of the traditional program analysis work culminated in the definition and use of the PDG or its constituents, the control and data dependence graphs. Such a graph defines the flow of control in the program as well as the dependence between the variables being used in the program's statements. Analyzing the graph allows us to detect loop invariant assignments for purposes of vectorization, classify the type of other dependencies as loop carried and internal ones, and when applied to programs in single static assignment (SSA) form [7] it can be quite effective.

The analysis methods allowed by this framework, however, are very much tied to the original structure of the program. Also, the focus is on following the dependency between symbols rather than tracking data flow per se, and moreover there is no explicit representation of the operations that are applied to the data. Thus, this approach is not quite strong enough to allow us to extract patterns of data modification that are not readily apparent from the source program.

Another approach is that of capturing the data dependence between array references by means of dependence vectors [6, 11, 20]. Then methods from linear algebra can be brought to bear to extract wavefronts of the computation. The problem with this framework is that only certain types of dependencies are modeled, again — there is no way to talk about specific operations, and in general it is hard to extend.

Finally, various symbolic methods have been proposed [10, 14, 17], mostly for plan analysis of programs. These methods all follow the structure of the program rigorously, and even though the program is transformed into some normal form up-front and all transformations preserve this property, still these methods are highly sensitive to "noise" in the source program. More severely, the reasoning about array references (which are the mainstay of scientific code) is quite limited for purposes of finding reduction operations on arrays.

The PDG-based methods seem to be most flexible and amenable for extensions. Recently [5] they have also been shown to be semantically sound, removing a potential draw-back. Moreover, this approach offers the most extensive support for program analysis that is required for enabling the usage of our techniques, namely it allows application of classical optimizations, vectorization, and more. Thus, we have chosen to extend the PDG based framework and provide algorithms that can be used on the new structure.

3 Computation Graphs and Algorithms for Idiom Extraction

In this section we propose a new graph theoretic model to represent the data flow and dependencies between values in a program, namely the *computation graph*. Such graphs are labelled (at the nodes), directed, and acyclic by construction. After formally defining this model, we provide an algorithmic framework that uses this new abstraction in order to analyze the structure of programs and identify computations that can be parallelized or otherwise computed more efficiently than they appear in the source. We conclude this section by stating some properties of computation graphs and by outlining the correctness of our transformations.

To guide the reader through this section we use the sample program of Figure 1, which is written in FORTRAN. This example is merely meant to illustrate the definitions and algorithms, not to show off the advantages of our approach over other work; this will be done in Section 4.

```
      DO 10 I=1,N
      T = I+4
C     assume M is even
      DO 10 J=3,M
      A(I,J) = A(I,J) + T*A(I,J-2)
   10 CONTINUE
```

Figure 1: A sample FORTRAN program.

3.1 Modeling

We first list some assumptions concerning the form of the programs on which we perform our analysis. This form can be obtained by applying certain well known optimization transformations.

- The program has been converted to Single Static Assignment (SSA) form per [7] and the Program Dependence Graph (PDG) [9] has been constructed. This means that we can ignore — to the extent of looking at the relationship between variables in the program — the dependence of variables on the outcome of conditional statements.
- We assume that standard basic block optimizations (such as dead-code and dead-store elimination, common subexpression detection, etc.) are performed on basic blocks, per [2], Section 9.4, so that the resulting set of assignments to values is "clean". This implies that the expression DAG that is obtained represents only the dependencies between variables that are live upon entry to the blocks and those that are live at the exit, thus reflecting the substitution of temporaries in expressions that use them locally and the like. We further assume that the dependency between values is "simple",

meaning that each defining expression is of some predefined form. Typical restrictions could be that it contains at most two values and one operator or that it be a linear form; which restriction is imposed depends on the type of recognition algorithm we want to apply later (in Section 3.2).

- All the PDG based optimizations (on both structure and data) have been performed, including the recognition of loops wherever possible. We also assume that the possibility to vectorize and apply some other parallelizing transformations (such as scalar expansion) has been detected and annotated; we may use some of this information in our algorithms or may prefer to subsume it with our own techniques, as we shall see.

- Loops have been *normalized* with respect to their index, as defined below. Here we follow Munshi and Simons [15] who have observed that by sufficient unrolling, all loop carried dependencies can be made to occur only between consecutive iterations.

Definition 1 *A loop is in normal form if all of its loop carried dependencies are between consecutive iterations.*

A loop can be normalized as follows: let the *span* of a loop be the largest integer k such that some value set in iteration i still depends on a value set in iteration $i - k$. Then, to normalize a loop with span $k > 1$, the loop's body is copied (unrolled) $k - 1$ times and the loop's index is adjusted so it starts at 1 and is incremented appropriately at each iteration. Loops are normalized from the most deeply nested outwards.

In the example of Figure 1, the inner loop needs to be normalized by unrolling it once, whereas the the outer loop is already in normal form (it is vacuously so, since there are no loop carried dependencies). We also assume that the temporary scalar T has been expanded, thus obtaining the program in Figure 2.

```
      DO 10 I=1,N
      T(I) = I+4
C     assume M is even
      DO 10 J=1,M-2,2
      A(I,J+2) = A(I,J+2) + T(I)*A(I,J)
      A(I,J+3) = A(I,J+3) + T(I)*A(I,J+1)
   10 CONTINUE
```

Figure 2: The program of Figure 1 in normal form.

Given a program satisfying the above assumptions, we define its computation graph, $G = (V, E)$, in two stages: first we handle basic blocks (that do not contain conditional branching), and then we show how to represent normalized loops. A node $v \in V$ represents the instance of an assignment statement to a variable (this includes a variable's initializations, since we use SSA form), denoted $var(v)$, and is uniquely identified by the statement number. We draw an edge from u to v if the value computed in u is used in v, *i.e.* $var(u)$

appears on the right hand side of a later assignment to $var(v)$ and no u' on the path from u to v has $var(u) = var(u')$.

We further label each node v by the function performed on the arguments to obtain the value assigned to $var(v)$. The function is represented in terms of a numbering on the edges entering the node (*i.e.* the arguments) which is used to disambiguate the defining expression[1], as is done in expression trees and DAGs.

Using the array references verbatim as atomic variables, the basic loop constituting the body of the inner loop of Figure 2 gives rise to the computation graph shown in Figure 3. Since we shall be looking for potential reduction and scan operations, which apply to linear forms, we allow the functions at nodes to be ternary multiply-add combinations. Note also that the nodes denoting the initial values of the variables use $\perp$ as their function label (which could be replaced when the graph is embedded in a larger context, as we shall see).

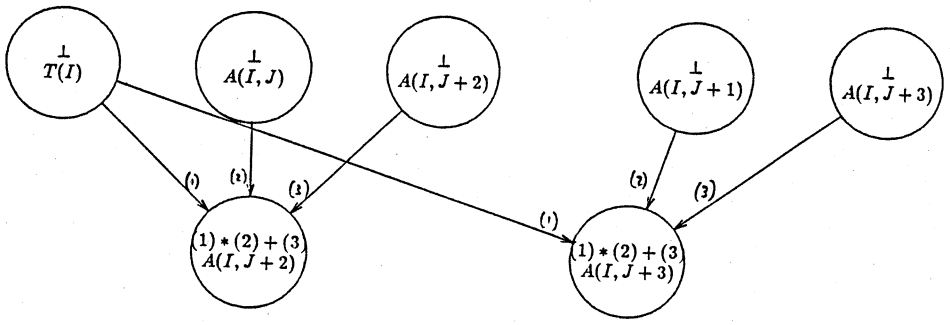

Figure 3: The computation graph of the basic block constituting the inner loop of the program in Figure 2.

Next we define computation graphs for normalized loops. Such graphs are obtained by replicating the DAGs representing the enclosed body[2] as follows. We generate three copies of the graph representing the body: one represents the initial iteration, one — a typical middle iteration, and one stands for the final iteration. Each node is uniquly identified by the statement number and the unrolled copy it belongs to (initial, middle, or final). Furthermore, array references that depend on the loop variable are changed by substituting the initial loop boundary, the name of the variable, and the final boundary in the index expressions respectively.

For every loop carried dependency [3, 20], we insert an edge from the appropriate vertex in the initial copy to the one in the middle copy, and likewise from the middle copy to the final copy. The loop is linked to its surroundings by (data) dependency edges where necessary (*i.e.* from initializations outside the loop and to subsequent uses).

[1] When the function is commutative, such as addition, this is not necessary.

[2] At the innermost level these are basic blocks, but as we go up the structure these are general computation graphs.

In order to cover the cases in which the loop body is executed fewer than three times, we splice each dependency edge that goes into the loop by adding a ϕ_{in}-node on it. From this ϕ_{in}-node we also draw an extra edge leading to the proper place in the *unrolled subgraph* representing the computations in which the loop is unrolled or is rolled less than three times. Such a node contains the proper ϕ-function (per [7]) which navigates the control flow for every execution. Similary, we splice every dependency edge leaving the loop, adding a ϕ_{out}-node, and draw an edge from the proper place of the unrolled graph to the new node (if that dependency appears in the unrolled graph). In the rest of the paper we omit the unrolled part of the graph since we are interested in parallelizing loops that have many iterations. that in case of nested loops there may be a need to use it.

Note that since nodes represent events rather than statements, all of the loop's anti-dependencies [3, 20] are reflected simply as data dependencies. The new graph spans a "signature" of the whole structure, and we call it the *summary form* of the loop. Note that once inner loops are transformed into their summary form, they are treated as part of the body of the enclosing loops and they may be — in turn — replicated similarly to yield another summary form. If the loop control variable is used in the computations within its body we treat it like any other variable with the proper initial value (node), and we add a statement (node) for incrementing the control variable.

Figure 4 shows the computation graph of the inner loop of the example in Figure 2. The graph comprises three copies of the graph from Figure 3 with the appropriate edges added. The graph for the whole program would consist of three copies of what is shown in Figure 4, with appropriate annotation of the nodes, but with no extra edges since there are no loop carried dependencies between the `I` iterations.

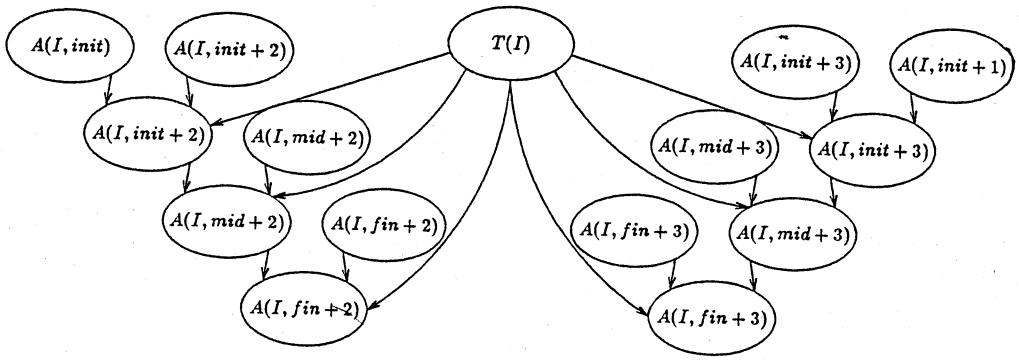

Figure 4: The computation graph of the inner loop of the program in Figure 2.

3.2 Algorithms

Having constructed the computation graph, the task of finding computational idioms in the program amounts to recognizing certain patterns in the graph. These patterns comprise graph structures such as paths or other particular subgraphs, depending on the target

idiom, and some additional information pertaining to the labeling of the nodes. The idiom recognition algorithm constitutes both a technique for identifying the graph patterns in the given graph as well as the conditions for when they apply, *i.e.* checking whether the context in which they are found is one where the idiom can indeed be used.

Overall, the optimization procedure consists of the following algorithmic ingredients:

- Matching and replacement of individual patterns can be achieved by using graph grammars to describe the rewrite rules. While rewriting, we also transform the labels (including the operators, of course), thus generating the target idioms. We make sure no side effects are lost by denoting forbidden entries and exits per [18].
- We need to provide a list of idioms and the graph rewriting rules that replace them. These should include structures such as reduction, scan, transposition, reflection, and FFT butterflies. Compositions thereof, such as inner product, convolution, and other permutations, can be generated as a preprocessing stage.
- At the top level, we (repeatedly) match patterns from the given list according to a predetermined application schedule until no more changes are applicable (or some other termination condition is met). This means that we need to establish a precedence relation among rules that will govern the order in which they are applied. This greedy tactic, which is similar to the conventional application of optimization transformations à la [2], is just one alternative. One could assign costs that reflect the merits of transformations and find a minimum cost cover of the whole graph at each stage, and then iterate.

We next elaborate on each of the above items, filling in the necessary details. First we discuss graph rewriting rules. Since we are trying to summarize information, these will be mostly reduction rules, *i.e.* shrinking subgraphs into smaller ones. More importantly, there are three characteristics that must be matched besides the skeletal graph structure: the operators (functions), the array references, and context, *i.e.* relationship to the enclosing structure.

The first two items can be handled by looking at the labels of the vertices. The third involves finding a particular subgraph that can be replaced by a new structure and making sure it does not interfere with the rest of the computation. For example, to identify a reduction operation on an array, we need to find a path of nodes all having the same associative operator label (*e.g.* multiplication, addition, or an appropriate linear combination thereof) and using consecutive (*i.e.* initial, middle, and final) entries in the array to update the same summary variable; we also need to ascertain that no intervening computation is going on (writing to or reading from the summary variable).

Both the annotations of the nodes as well as the guards against intervening computations are part of the graph grammar productions defining the replacement rules. Figure 5 provides two such rules to make this notion clear. Recall that we assume that at this point certain vectorization transformations, including scalar expansion, have occurred, so we use their results when looking for patterns (alternatively, we can generate these transformations, too, with appropriate expanding rewrite rules).

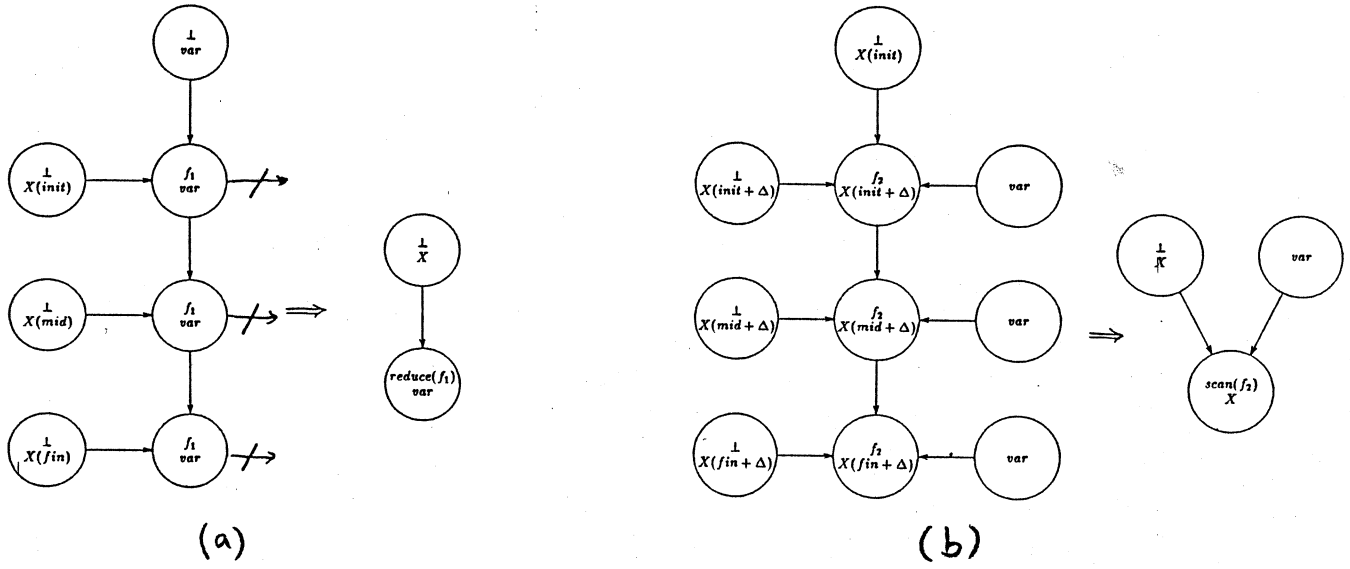

Figure 5: Matching and replacement rules for (a) reduction and (b) scan.

Once such rules are applied, the computation graph contains new types of nodes, namely summary nodes representing idioms. These nodes can, of course, appear themselves as candidates for replacement (on the left hand side of a rule), thereby enabling further optimization. Obviously, we would apply the rules for lower level optimizations first and only then use the others, but one should not get the impression that this procedure necessarily follows the loop structure of the given program. On the contrary, the computation graph as constructed allows the detection of structures that might otherwise be obscured, as we shall see in Section 4.

The result of applying the rules of Figure 5 to the graph of Figure 4 is shown in Figure 6. If we had started with a graph for the entire program of Figure 2, not just the inner loop, applying a vectorization rule to the result would have generated the program of Figure 7.

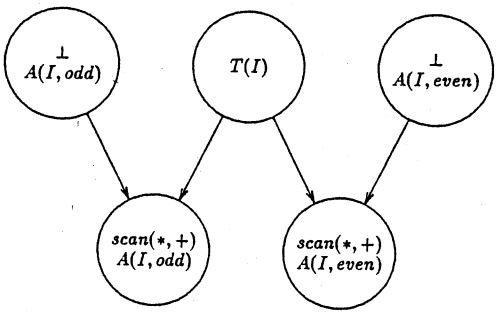

Figure 6: The computation graph resulting from applying the transformations of Figure 5 to the graph in Figure 4.

```
      DOALL 10 I=1,N
      T(I) = I+4
      SCAN(A(I,*),1,2,T(I),"*","+")
      SCAN(A(I,*),2,2,T(I),"*","+")
   10 CONTINUE
```

Figure 7: The program resulting from applying idiom extraction on the program of Figure 2. (We use an arbitrary template for `SCAN` which includes all the necessary information, including strides inside vectors or minors and the operations to be performed.)

In general, the order in which rules are applied and the termination condition depend on the rule set. If the rules are Church-Rosser then this is immaterial, but often they are competing (in the sense that the application of one would outrule the consequent application of the other) or are contradictory (creating potential oscillation). This issue is beyond the scope of this paper and we defer its discussion to general studies on properties of rewrite systems.

3.3 Correctness

In this section we argue that the computation graph faithfully represents the computation expressed by the program. A *computation* of a program is a partially ordered set of events that is derived from the program during its execution. Before evaluation we replace every conditional branch and loop header with the corresponding ϕ-node. The events of the evaluation are the executions of assignment statements and the results of conditionals (ϕ-nodes), and the ordering is defined by the execution order. We define the *dependency relation* to be the irreflexive transitive closure of the program's computation. The collection of all such relations of a program, with respect to the input domain, represents all the computations of the program.

Definition 2 *Given a computation graph of a program, we define a subgraph of the computation graph to be a representative of a computation of the program if the following four conditions hold: (i) it contains the source (root) nodes corresponding to the variables initialized by the computation, (ii) for every node which is not a ϕ-node it contains all the edges and nodes reachable from the node via non ϕ-nodes, (iii) for every ϕ-node reached only one exiting edge is in the subgraph, and (iv) the irreflexive transitive closure relation of the subgraph is a subset of the dependency relation.*

To establish the correctness of this representation, it is not difficult to prove the following.

Lemma. 1 *Every terminating computation of the program has a representative in the program's computation graph.*

Next we claim that the representatives are detailed enough in order to carry out our transformations. For non-looping computations, the computation graph defines the same ordering

relation as the computation of the program. Since the major area of optimization is array access in loops, we must justify the construction of the computation graph for loops. There is one key lemma (presented here without proof, which is straightforward), as follows:

Lemma. 2 *Unfolding three iterations of a normalized loop is enough to reveal its data dependence structure.*

We finally define the correctness conditions for transformations in our framework.

Definition 3 *A transformation is correct if for any given program every terminating computation of the transformed program has a representative in the computation graph of the original program, and each representative subgraph of the computation graph is consistent with the ordering of some computation.*

4 Examples

To exemplify the advantages of our idiom recovery method, we present here three cases in which other methods cannot speed the computation up but ours can (if combined properly with standard methods). We do not include the complete derivation in each case, but we outline the major steps and point out the key items.

The first example is one of the loops used in a Romberg integration routine. The original code (as released by the Computer Sciences Corporation of Hampton, VA) looks as follows

```
      DO 40 N=2,M+1
      KN2 = K+N-2
      KNM2 = KN2+M
      KNM1 = KNM2+1
      F = 1.0 / (4.0**(N-1) - 1.0)
      TEMP1 = WK(KNM2) - WK(KN2)
      WK(KNM1) = WK(KNM2) + F*TEMP1
   40 CONTINUE
```

but after substitution of temporaries, scalar expansion of `F`, and renaming of loop bounds (no normalization is necessary in this case), which are all straightforward transformations, we obtain

```
      DO 40 I=K+M,K+2*M
      F(I)= 1.0 / (4.0**(I-K-M+1) - 1.0)
      WK(I+1)=WK(I)+F(I)*(WK(I)-WK(I-M))
   40 CONTINUE
```

Now observe two facts: all the references to `WK(I-M)` are to elements that are not set in this loop (per the given bounds), and the computation of `F(I)` can be performed in a separate loop ("loop distribution"); again, these facts can be deduced with known techniques.

The loop computing F can be easily vectorized, so all we are left with is a loop containing a single statement which computes WK(I+1). If we draw the computation graph for this loop and allow operators to be linear combinations with a coefficient (F(I) in this case), we can immediately recognize the scan operation that is taking place.

The second example is taken from [3], who gave up on trying to parallelize it (and indeed it cannot be vectorized):

```
      DO 100 I=1,N
      C(I)=A(I)+B(I)
      B(I+1)=C(I-1)*A(I)
  100 CONTINUE
```

In this case, a 3-fold unrolling is required. Once this is done, we detect two independent chains in the computation graph, one including the computation of the even[7] entries in B and C, and the other computing the odd entries. If the data is arranged properly, the whole computation can be attained by two scan operations, taking time $O(\log n)$ (n being the value of N) rather than $O(n)$.

Finally, we parallelize a program computing the inner product of two vectors, which is a commonly occurring computation. Consider

```
      P=0
      DO 100 I=1,N
      P=P+X(I)*Y(I)
  100 CONTINUE
```

Traditional analysis of the program (in preparation for vectorization) replaces the loop body by

```
      T(I)=X(I)*Y(I)
      P=P+T(I)
```

Figure 8 shows the computation graphs of this transformed basic block and that of the relevant part of the loop. The vectorizable portion of the loop can be transformed into a statement of the form T=X*Y, and what is left can be matched by the left hand side of rule (a) of Figure 5. Applying the rule produces the graph corresponding to the statement REDUCE(P,1,1,T(I),"+").

The recognition of an inner product using our techniques will not be disturbed by enclosing contexts such as a matrix multiplication program. The framed part of the program in Figure 9 produces the same reduced graph that appears replicated after treating the two outermost loops.

[7] We use "even" and "odd" here just to distinguish the chains; the recognition procedure does not need to know or prove this fact at all.

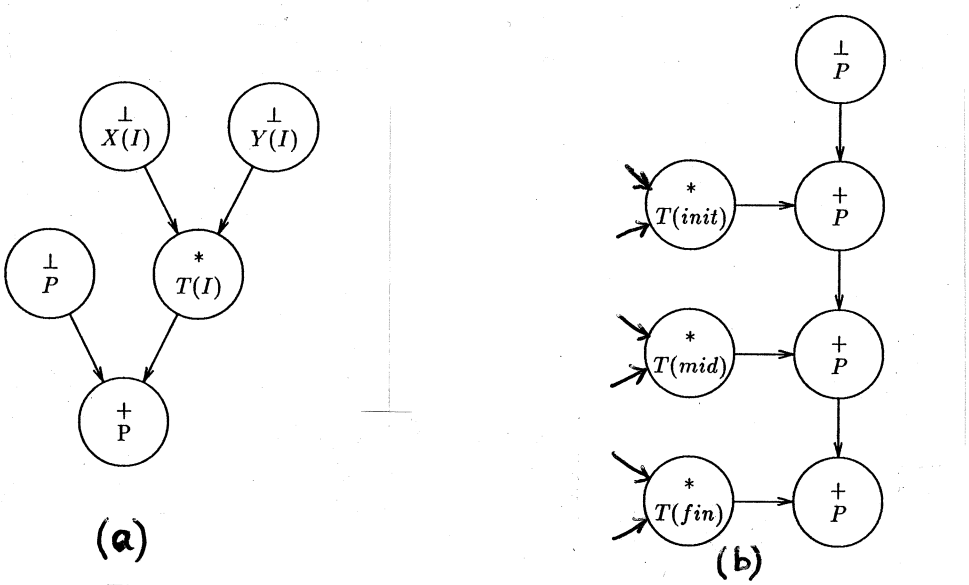

Figure 8: Transforming an inner product to a reduction.

```
      DO 100 I=1,N
      DO 100 J=1,N
      C(I,J)=0
      DO 100 K=1,N
      C(I,J)=C(I,J)+A(I,K)*B(K,J)
100   CONTINUE
```

Figure 9: The portion of a matrix multiplication that is recognized by the tarnsformation of Figure 8.

5 Other Applications and Future Work

Our primary objective is to cater for machines with effective support of data parallelism. Our techniques, however, are not predicated upon any particular hardware, but can rather be targeted to an abstract architecture or to language constructs that reflect such features. Examples of such higher level formalisms are *APL* functionals and idioms, the BLAS package (which has many efficient implementations on a variety of machines), vector-matrix primitives as suggested in [1], and languages such as Crystal, `C*` and `*lisp` which all support reductions and scans.

All in all, we feel that this paper makes both methodological and algorithmic contributions that should be further investigated. We believe that in addition to the constructs mentioned above, many other can be expressed as patterns and be identified as idioms. Such constructs need to be formulated in terms of computation graph patterns, and further — additional work on algorithmic methods other than repeated application of the rules (such as fixed-point computations) is also necessary. Finally, we intend to implement the techniques mentioned here as part of an existing parallelization package and obtain experimental results that would indicate how prevalent each of the transformations is.

References

[1] A. Agrawal, G. E. Blelloch, R. L. Krawitz, and C. A. Phillips. Four vector-matrix primitives. In *Symposium on Parallel Algorithms and Architectures*, pages 292–302. ACM, June 1989.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[3] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Fourteenth Annual Symposium on Principles of Programming Languages.* ACM, January 1987.

[4] G. Blelloch. Scans as primitive parallel operations. In *International Conference on Parallel Processing*, pages 355–362. IEEE, 1987.

[5] R. Cartwright and M. Felleisen. The semantics of program dependence. In *SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 13–27. ACM, 1989.

[6] M. C. Chen. A parallel language and its compilation to multiprocessor machines or VLSI. In *Thirteenth Annual Symposium on Principles of Programming Languages*, pages 131–139. ACM, January 1986.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Sixteenth Annual Symposium on Principles of Programming Languages*, pages 25–35. ACM, January 1989.

[8] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. on Mathematical Software*, 14(1):1–17, March 1988.

[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319–349, July 1987.

[10] P. Jouvelot and B. Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *International Conference on Supercomputing*, pages 186–194. ACM, June 1989.

[11] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the Association for Computing Machinery*, 14(3):563–590, July 1967.

[12] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the Association for Computing Machinery*, 27(4):831–838, October 1980.

[13] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. on Mathematical Software*, 5(3):308–323, September 1979.

[14] S. I. Letovsky. *Plan Analysis of Programs.* PhD thesis, Dept. of Computer Science, Yale University, December 1988. Available as YALEU/CSD/RR 662.

[15] A. Munshi and B. Simons. Scheduling sequential loops on parallel processors. Technical Report RJ 5546, IBM Almaden Research Center, 1987.

[16] A. J. Perlis and S. Rugaber. Programming with idioms in *apl.* In *APL'79*, pages 232–235. ACM, June 1979. *APL* Quote Quad, Vol. 9, No. 4.

[17] C. Rich and R. C. Waters. The programmer's apprentice: a research overview. *Computer*, 21(11):10–25, November 1988.

[18] M. Rosendahl and K. P. Mankwald. Analysis of programs by reduction of their structure. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph-Grammars and Their Applications to Computer Science and Biology*, pages 409–417. Springer-Verlag, 1979. Lecture Notes in Computer Science, Vol. 73.

[19] Thinking Machines Corporation, Technical Report HA87-4. *Connection Machine Model CM-2 Technical Summary*, April 1987.

[20] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982. Available as TR UIUCDCS-R-82-1105.