NASA Contractor Report 178277

ICASE REPORT NO. 87-23

ICASE

EXPLOITING LOOP LEVEL PARALLELISM IN NONPROCEDURAL DATAFLOW PROGRAMS

Maya B. Gokhale

Contract No. NAS1-18107

April 1987

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

EXPLOITING LOOP LEVEL PARALLELISM IN NONPROCEDURAL DATAFLOW PROGRAMS

Maya B. Gokhale
Department of Computer and Information Sciences
University of Delaware

ABSTRACT

In this paper, we discuss how loop level parallelism is detected in a nonprocedural dataflow program and how a procedural program with concurrent loops is scheduled. In addition, we discuss a program restructuring technique which may be applied to recursive equations so that concurrent loops may be generated for a seemingly iterative computation. A compiler which generates C code for the language described below has been implemented. We describe the scheduling component of the compiler and the restructuring transformation.

This research was supported in part by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18107 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665 and in part by UDRF Grant LTR860114.

Exploiting Loop Level Parallelism in Nonprocedural Dataflow Programs

Maya B. Gokhale
Department of Computer and Information Sciences
University of Delaware

1 Introduction

Loop level parallelism has been recognized as having major impact in the performance of parallel programs on MIMD machines. Most parallel languages contain some sort of *forall* construct (for example, see [7] or [12]) and much effort has been directed towards detecting forall loops in sequential programs (for a small sampling of the literature in this area, see [1], [3], [4], [8], [9], [13], [16]) and automatically generating parallel versions of the sequential programs.

In this work, we also address the question of automatically generating forall loops, but from a different perspective. Our starting point is a very high level dataflow language PS (similar to [2], [17], [19], and [18]), attractive because of its functional semantics which greatly facilitate program restructuring.

To demonstrate this, we show how we can transform certain forms of subscript expressions used to index recursively-defined arrays so that concurrent loops may be generated instead of sequential ones. Although these transformations are also applicable to sequential programs, a more careful analysis of reassignment and aliasing may be required in the sequential program. In some cases, the programmer's use of aliasing and reassignment may prevent the transformations from being applied.

In addition to the benefits of its functional semantics, the language PS is attractive for the close correspondance in form between equations used to describe numerical algorithms and equations in PS. In fact, we may consider the language to be an internal (albeit textual) representation of

equations such as Equation 1 below. Our ultimate goal is a translator of equations in the form of (1), perhaps as TEXor Postscript files, to modules in this language.

To date we are implementing a compiler for PS, currently 24,000 lines of Pascal, which generates declarations and functions in the C language. The compiler automatically collects equations into groups for which are generated C *for* loops. Each loop is annotated to indicate whether it is an iterative or concurrent *for*. The subscript transformation described in Section 4 has been developed independently, and is being integerated into the compiler.

In this paper, we describe the scheduling phase of the compiler, with particular emphasis on

- differentiating between parallel and sequential loops,
- memory reuse in the generated imperative code, and
- array transformation to facilitate parallel loop generation and memory reuse.

2 The PS Language

The Problem Specification (PS) language [6] is a very high level dataflow language. A program in this language consists of one or more *module* descriptions, where a module is simply a functional unit, taking 0 or more input parameters and returning 1 or more result. Internal to a module the data declarations resemble Pascal or Modula-2. There may be user-defined types or variables declared. Standard Pascal data types are provided (primitive types, enumerations, arrays, records). In place of the procedural code, however, PS has a *define* section consisting of equations defining values for all non-input variables. The equations may be entered in any order. An equation in PS is a restricted form of mathematical equation in that the left hand side of the "=" is either a single variable or a list of variables, and the right hand side is an expression of the same arity and type as the the left hand side. A *scheduling* phase of the compiler derives from the data dependency graph of the equations an ordering for the procedural code which is emitted.

Example: Let us take a a simplified version of standard relaxation, where for non-boundary elements and for $k > 1$,

$$A^{(k)}[i,j] = (A^{(k-1)}[i,j-1] + A^{(k-1)}[i-1,j] + A^{(k-1)}[i,j+1] + A^{(k-1)}[i+1,j])/4 \qquad (1)$$

Note that in this example, all element values are taken from the previous iteration. In PS the superscripts (iteration number) and subscripts (array element) are not differentiated. All of them are put in as subscripts:

```
A[K,I,J] = ( A[K-1,I,J-1]
           +A[K-1,I-1,J]
           +A[K-1,I,J+1]
           +A[K-1,I+1,J] ) / 4;
```

To make a PS module out of this fragment, we first write the module header:

```
Relaxation: module (InitialA: array[I,J] of real;
                    M: int; maxK : int):
                    [newA:array[I,J] of real];
```

InitialA is the input array of dimension $M \times M$, maxK is the number of iterations desired, and newA is the array returned as the module result.

Next, we define types and local variables:

```
type
   I,J  = 0 .. M+1;  K = 1 .. maxK;

var
   A: array [K] of array[I,J] of real;
```

We have defined subrange types I and J as ranging from 0 to M + 1, so that the boundary may be padded with 0's. K is the superscript from the initial formula. Since A has dimensionality which is the sum of subscripts and superscripts, [1] it is declared as a local 3-dimensional array.

Now we insert the equations. The initial value of A is simply the input array InitialA. The result NewA is the value of the maxK'th element of A.

```
  A[1] = InitialA; (* the first grid is input *)
  newA  = A[maxK]; (* the grid returned is
                      from the last iteration *)
```

Next we give the equation defining the other elements of A, including the boundary values. This equation uses an if expression to determine whether an element is a boundary value or an interior value.

```
      A[K,I,J] = if (I = 0)  (* carry over boundary points *)
                 or (J = 0)
                 or (I = M+1)
                 or (J = M+1)
                 then  A[K-1,I,J]
                 else ( A[K-1,I,J-1]
```

[1] In keeping with other single assignment languages, a value is never changed. Rather a new value is generated from a computation involving the old value.

```
+A[K-1,I-1,J]
+A[K-1,I,J+1]
+A[K-1,I+1,J] ) / 4;
```

The entire module is shown in Figure 1.

3 Scheduling the Equations of a Module

The PS compiler consists of three components,

- the "front end" which does syntax and semantic analysis and stores the entire program in an internal form
- the scheduler, which, on a module by module basis, builds a data dependency graph, analyzes the graph and generates a flowchart of execution ordering including, if necessary, iterative and parallel loops
- the code generator which generates procedural code from the flowchart.

In this paper we concentrate on the scheduler, beginning with a description of the dependency graph.

3.1 The Dependency Graph

The dependency graph $G = (N, E)$, where the set of nodes N contains the data items and equations of the module, and E contains directed edges between nodes. A directed edge is drawn from node i to node j if data produced in i is used in j. Thus the graph is simply a dataflow graph, showing the flow of data from producer to consumer. There exist data dependency edges from all variables on the right hand side of an equation to the equation, and from the equation to the variable on the left hand side. In addition, data dependency edges are drawn from variables defining a subrange bound to variables using that subrange. For example, a data dependency edge is drawn from `M` to `InitialA`, to `A`, and to `NewA`, since the bounds of these arrays depend on `M`. A data dependency edge is drawn from `maxK` to `A` for the same reason. Besides the data dependency edges, certain hierarchical edges also are drawn. These are used to show the relationship between the fields of a record and the record itself, and do not concern us further in this example.

Each node and each edge is annotated with a list of labels. There is a node label for each dimension of the node, eg. an array A[K,I,J] has three node labels, describing respectively, the dimensions K, I and J. The edge labels contain information about the subscript expression used to reference the source node. Figure 2 show in further detail the attributes of the edge labels.
The dependency graph for the `Relaxation` Module is shown in Figure 3.

```
(*$m+v+x+t-*)
Relaxation: module (InitialA: array[I,J] of real;
                    M: int; maxK : int):
                    [newA:array[I,J] of real];

type
   I,J  = 0 .. M+1;  K = 2 .. maxK;

var A: array [1 .. maxK] of array[I,J] of real;
(* A denotes the succession of grids *)

define
 (*eq.1*) A[1] = InitialA; (* the first grid is input *)
 (*eq.2*) newA  = A[maxK];      (* the grid returned is from
                                   the last iteration *)
 (*eq.3*) A[K,I,J] = if (I = 0)  (* carry over boundary points *)
                     or (J = 0)
                     or (I = M+1)
                     or (J = M+1)
                     then  A[K-1,I,J]
                     else ( A[K-1,I,J-1]
                           +A[K-1,I-1,J]
                           +A[K-1,I,J+1]
                           +A[K-1,I+1,J] ) / 4;
end Relaxation;
```

Figure 1: The Relaxation Module

- Position in Target of this Source Subscript
- Subscript Expression Type
  - "I" as in A[I]
  - "I - constant" as in A[I-2]
  - any other expression
- Offset amount. Applicable only to "I - constant" subscript expression

Figure 2: Edge Label Attributes

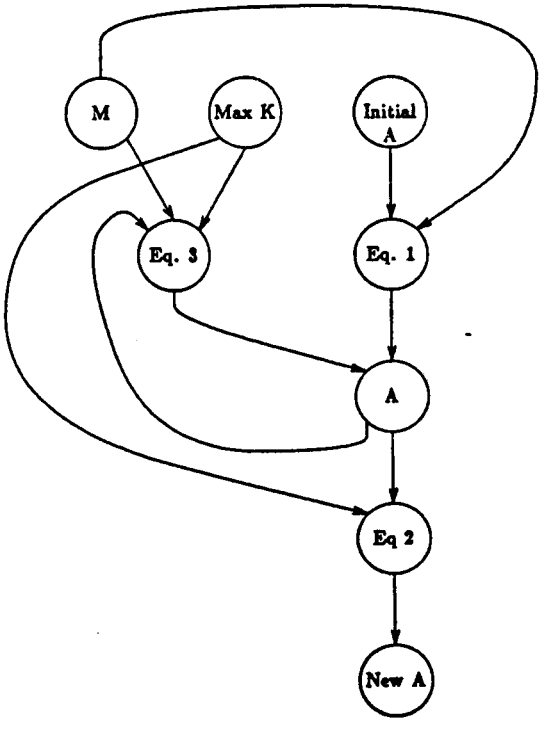

Figure 3: Dependency Graph for `Relaxation` Module

- Descriptor type: either Dependency Graph Node or Subrange Type
- If Subrange Type,
  - Is an iterative loop to be generated from this subrange or is a parallel loopto be generated?
  - List of descriptors which are nested within this subrange

Figure 4: A Flowchart Descriptor

3.2 Scheduler Output

If the scheduler can determine an execution ordering for the equations, it generates a flowchart describing both the order of equations and the loop nesting structure in which the equations are embedded. The flowchart, then, is used by the code generator to emit the procedural code. The flowchart is simply a list of descriptors. A descriptor may indicate either a dependency graph node or a subrange type. The code generator, on encountering the former, emits code for the data item or the equation. The presence of the latter means that a *for* loop over the indicated subrange is to be generated. A subrange type descriptor also contains a list of descriptors which are contained within the scope of the loop. Thus the flowchart is a recursive structure which reflects the nesting structure of the generated program. The format of a descriptor is shown in Figure 4.

3.3 The Scheduling Algorithm

The scheduling algorithm described here is a variant of the algorithm of [15]. Our algorithm is most similar to [5], which generates a schedule for subsequent code generation to a procedural data flow language. The algorithm described below does distinguish between iterative and parallel loops, but performs poorly in other respects, such as combining into a single loop those equations which though not recursively related, nevertheless depend on the same subscript(s). See [11] for a scheduling algorithm which produces only iterative loops, but does combine non-recursively related equations which depend on the same subscript(s), and [5] for the algorithm which distinguishes between iterative and parallel loops, but does not combine iterative components which depend on the same subscript(s).

The scheduler consists of two mutually recursive procedures. The first, Schedule-Graph, takes as input a dependency graph, and returns a flowchart. The second, Schedule-Component, schedules a Maximally Strongly Connected Component (MSCC) of a dependency graph, and returns a flowchart.

Schedule-Graph operates as follows:

1. Find the MSCC's of the graph $\{M_i\}, i = 1 \ldots n$, where n is the number of MSCC's.
2. Initialize the flowchart to null.
3. For each M_i,
   (a) Call Schedule-Component with M_i as input
   (b) Concatenate the result returned by Schedule-Component onto the flowchart.
4. Return the flowchart.

Schedule-Component, in turn, does the following:

1. If the component consists of exactly one data node, exit with a null schedule.
2. Pick an unscheduled node dimension to use as loop subscript.
   (a) If there are no more dimensions left to be scheduled and the graph contains more than 1 node, then signal error and return: the equations cannot be scheduled by this algorithm.
   (b) If there are no more dimensions left to be scheduled and the graph contains exactly 1 node then return as flowchart that single node.
3. Otherwise verify that the subrange associated with that dimension appears in a consistent position in each node of the component, [2] and that the only subscript expressions used in that dimension are either "I" or "I - constant".
4. Delete edges in M_i which contain subscript expressions of type "I - constant" in the dimension being scheduled.[3]
5. Mark the dimension as scheduled.
6. Create a flowchart descriptor for a Subrange Type. If "I - constant" edges were deleted, record that an iterative loop is to be generated, otherwise parallel.
7. Call Schedule-Graph with the subgraph which results from deleting the "I - constant" edges.
8. Concatenate the result returned by Schedule-Graph onto the Subrange Type flowchart descriptor created above, and return the resulting list.

Let us apply the scheduling algorithm to the relaxation dependency graph (Figure 3). The component graph and the flowchart for each component are shown in Figure 5. Input to Schedule-

[2] For example, in the equation `A[I,J] = A[I,J-1] + A[J,I]` the subscripts `I` and `J` are not in a consistent position.

[3] We can delete these recursive edges and still generate a correct schedule because if the loop iterates from the low bound of the subrange to the high bound, a reference to (for example) A[I-2] will refer to an element of A which was produced two iterations back.

Component	Node(s)	Flowchart
1	InitialA	null
2	m	null
3	maxK	null
4	eq.1	DOALL I (DOALL J (eq.1))
5	A, eq.3	DO K (DOALL I (DOALL J (eq.3)))
6	eq.2	DOALL I (DOALL J (eq.2))
7	newA	null

Figure 5: Component Graph and Corresponding Flowchart

Graph is the entire graph. The MSCC's of the graph are shown in Figure 5. Schedule-Graph calls Schedule-Component successively with component. The third column of Figure 5 shows the flowchart returned by Schedule-Component for each of the components. Components 1, 2, 3, and 7 result in a null flowchart being returned by Schedule-Component at step 1 of the algorithm.

For Component 4, Schedule-Component chooses the first dimension to schedule (I) and recursively calls Schedule-Graph with that component as input. Schedule-Graph in turn recursively calls Schedule-Component with the same component, the first dimension of which is marked scheduled. Schedule-Component now marks the second dimension (J) scheduled, and calls Schedule-Graph with the same component as input. Schedule-Graph calls Schedule-Component. Now, by step 2b of Schedule-Component, eq.1 is returned as result to Schedule-Graph, which returns it to Schedule-Component, which concatenates it onto a "DOALL J" and returns "DOALL J eq.1" to Schedule-Graph, which returns it to Schedule-Component, which concatenates it onto "DOALL I" and returns "DOALL I DOALL J eq.2" to the original call of Schedule-Graph.

Component 6 is processed in exactly the same way as Component 4. Component 7, however, is a multi-node MSCC. Schedule-Component picks the first dimension (K) to schedule first. The other two cannot be chosen because of subscript expressions "J + 1" and "I + 1" (see Schedule-Component step 3). All edges having subscript expression "K - 1" are deleted by step 4, and Schedule-Graph is called recursively. The subgraph now has two components, eq.3 and A. Eq.3 can be scheduled in the I and J dimensions as outlined above for eq.1. The flowchart for A is null. Thus the flowchart for Component 5 consists of an inner two level DOALL and an outer DO. The final schedule returned by the outermost call to Schedule-Graph is shown in Figure 6.

3.4 Virtual Dimension

The code generation phase generates C declarations and assignment statements. For each variable, either input parameter, output parameter, or local variable, an equivalent C declaration is gener-

```
DOALL I (
  DOALL J (
    eq.1
          )
        )
DO K (
     DOALL I (
    DOALL J (
       eq.3
    )
  )
)
DOALL I (
  DOALL J (
    eq.2
          )
        )
```

Figure 6: Flowchart for the Relaxation Module

ated. Then, using the flowchart, the code generator emits *for* loops and assignment statements.

Thus array declarations are generated for each of the arrays InitialA, NewA, and A. Now it is obvious that allocating a three dimensional array for A is unnecessary,[4] since in C and other imperative languages the same function can be performed by a two dimensional array with reassignment. The k'th dimension of A can be thought a "virtual" dimension rather than one physically allocated in its entirety.

In general, a data node dimension is defined to be physical if the number of elements allocated at that dimension of the generated variable is the same as the number declared in the PS declaration. A data node dimension is virtual if the dimension is mapped to a "window" of elements, and the width of the window is smaller that the PS declared size. For the array A, a window of two elements is needed, the current one K, and the previous, K-1.

The scheduler recognizes virtual data node dimensions during the Schedule-Component phase. For each data node which is a local variable N_r in the component M_i, the node dimension being scheduled is marked virtual if each edge from N_r to a node of type equation is in one or both of the following forms:

1. The edge has subscript expression "I" or "I - constant" in the dimension being scheduled, and the target is in M_i

2. The edge goes to a node outside the component, and the edge has a subscript expression of the form "N", where "N" has been used as the upper bound of the subrange defining that dimension. This type of edge indicates that only the last element at that dimension is used outside the loop.

In the case of our example, local variable A is virtual in dimension 1. The other two dimensions are not virtual for two reasons: first, they have edges with subscript expression "I + constant", and second, there are edges going out out of the component which don't have the second form of subscript expression in those dimensions. Therefore the scheduler marks the first dimension of data node A virtual with window two, thereby directing the code generator to allocate only two instances rather than maxK instances.

4 A Restructuring Transformation

We now look at a slightly modified version of Equation 1. We will show that what appears to be a strictly iterative formulation can, with a shift of coordinate system, still result in a parallel loop, and that the "iteration" superscript need not really be an iteration subscript in the generated program.

[4] It is also obvious that generating a declaration for NewA is unnecessary. Our solution to this problem is beyond the scope of this paper.

Let us now solve the more standard (but still simplified) relaxation, for $k > 1$,

$$A_{i,j}^{(k)} = (A_{i,j-1}^{(k)} + A_{i-1,j}^{(k)} + A_{i,j+1}^{(k-1)} + A_{i+1,j}^{(k-1)})/4 \quad (2)$$

This results in equation 3 of the module becoming:

```
A[K,I,J] = if (I = 0)  (* carry over boundary points *)
           or (J = 0)
           or (I = M+1)
           or (J = M+1)
           then  A[K-1,I,J]
           else ( A[K,I,J-1]
                 +A[K,I-1,J]
                 +A[K-1,I,J+1]
                 +A[K-1,I+1,J] ) / 4;
```

Now when we apply the scheduling algorithm, we find that deleting the K-1 edges leaves two recursive edges, so that both the I and the J loop must be iterative.

The resulting schedule is shown in Figure 7.

The virtual dimension analysis gives the same result as in the previous version: the first dimension of A is virtual with window of two elements. Note that each dimension is scheduled independently, so that we do not detect the fact that only one array is needed.

At this point,we take a closer look at the data dependencies of A. The dataflow graph for A in which each array element is a node (rather than the form used above in which there is a single node for the entire array) shows that all elements whose indices satisfy the equation

$$2K + I + J = t, \;\; t = 1 \ldots 2 \times maxK + 2 \times M$$

can be computed at one time.

In this section we demonstrate a technique by which such parallelism can be detected. In particular, we will show how to find linear solutions to a set of inequatities representing the recursive array dependencies. See [10] for a more complete treatment of the subject for constant offset recursive equations, and [14] for an extension to the method which handles certain forms of symbolic offsets in recursive equations.

Our fundamental constraint is that data must be produced before it can be used. Thus $A[K, I, J]$ cannot be created until after $A[K-1, I, J]$,$A[K, I, J-1]$, $A[K, I-1, J]$, $A[K-1, I, J+1]$, and $A[K, I+1, J]$ are available.

We define the time of creation for each array element as a linear combination of the indices. For the recursively defined array A, this gives us the *time equation*

$$t(A[K, I, J]) = aK + bI + cJ.$$

```
InitialA
DOALL I (
  DOALL J (
    eq.1
          )
        )
DO K (
     DO I (
    DO J (
       eq.3
    )
  )
)
DOALL I (
  DOALL J (
    eq.2
          )
        )
```

Figure 7: Flowchart with Revised Eq.3

Our first problem is to solve for the coefficients a, b, and c.

We now represent the problem's dependence ordering with (strict) inequalities involving time. In this case, the time for $A[K,I,J]$ must come after the time for $A[K-1,I,J]$, etc. which gives us five *dependence inequalities*:

$$\begin{aligned} aK+bI+cJ > a(K-1)+bI+cJ &\Rightarrow a>0 \\ aK+bI+cJ > aK+bI+c(J-1) &\Rightarrow c>0 \\ aK+bI+cJ > aK+b(I-1)+cJ &\Rightarrow b>0 \\ aK+bI+cJ > a(K-1)+bI+c(J+1) &\Rightarrow a>c \\ aK+bI+cJ > a(K-1)+b(I+1)+cJ &\Rightarrow a>b \end{aligned}$$

Now we can find the least integers a b, and c for which these dependence inequalities will hold. In this case, we get $a=2$ and $b=c=1$, and arrive at the time equation $2K+I+J$ cited above.

All array elements $A[K,I,J]$ such that $2K+I+J=t$ will be defined at time t. For given t, these entries comprise a "hyperplane". As t is increased from 0 to $t_{Max}=K_{Max}+I_{Max}+J_{Max}$, we find a sequence of such hyperplanes which cover every point in the array.

We now define a new array A' related to A so that $A'[K',I',J'] = A[K,I,J]$ and have $A'[K',I',J']$ be constructed at time K'. Thus, we transform the coordinates K,I,J for the array A into coordinates K',I',J' such that $K'=t=2K+I+J$. A method for obtaining the I' and J' dimensions after K' has been determined is given in [10]. In this example, we find that $I'=K$ and $J'=I$. Specifically,

$$\begin{aligned} K' = 2K+I+J \qquad & I' = K \qquad & J' = I \\ K = I' \qquad & I = J' \qquad & J = K'-2I'-J' \\ A[K,I,J] = \; & A'[K',I',J'] = \; & A'[2K+I+J,K,I], \\ A'[K',I',J'] = \; & A[K,I,J] = \; & A[I',J',K'-2I'-J'] \end{aligned}$$

Using these equalities, we derive the following recursive equation using A'.

$$\begin{aligned} A'[K',I',J'] &= A[I',J',K'-2I'-J'] \\ &= A[I'-1,J',K'-2,I-J'] \\ &\quad \text{if } J'=0 \vee K'-2I'-J'=0 \vee J'=M+1 \vee K'-2I'-J'=M+1 \\ &= A'[K'-2,I'-1,J'] \\ &\quad \text{if } J'=0 \vee K'-2I'-J'=0 \vee J'=M+1 \vee K'-2I'-J'=M+1 \end{aligned}$$

$$
\begin{aligned}
&= (A[I',J',K'-2I'-J'-1]+A[I',J'-1,K'-2I'-J'] \\
&\quad +A[I'-1,J',K'-2I'-J'+1]+A[I'-1,J'+1,K'-2I'-J'])/4 \\
&\quad \text{otherwise} \\
&= A'[K'-1,I',J']+A'[K'-1,I',J'-1] \\
&\quad +A'[K'-1,I'-1,J']+A'[K'-1,I'-1,J'+1] \\
&\quad \text{otherwise by simplification}
\end{aligned}
$$

Applying the scheduling algorithm to the subgraph of this recursive equation gives us an outer iteration, as before. However, once the "K' - constant" edges have been deleted, the I and J dimension can be scheduled as parallel loops (as in the example based on Equation 1) rather than iterative. In fact, the schedule is identical to that of Figure 6.

In addition, by using the transformed array A' instead of the original array A in the scheduling algorithm, we now find that the first dimension of A' is virtual, since the only references are to $K'-1$ or $K'-2$. The window size is three, so that we can allocate an array $3 \times I'_{Max} \times J'_{Max} = 3 \times maxK \times M$ rather than $2 \times M \times M$, the space allocation of the purely iterative version.

In the final code which is generated, there are several alternatives in how the transformed array is treated. We can flag arrays which have undergone this transformation, and replace each reference to $A'[K',I',J']$ by $A[I',J',K'-2I'-J']$. Alternatively, we could redefined A as a function which retrieves the proper entry from A'. With a little more intelligence, we could rotate the input array into $A'[1]$, work entirely with the transformed array A' in the recurrence, and unrotate back into the return parameter. The latter approach is preferable because a regular pattern of array reference is established for the iteration which can be optimized (with respect to stride length) in procedural multiprocessor compilers.

5 Conclusion

We have presented a nonprocedural dataflow language and shown how the compiler for the language creates the flowchart of a procedural loop program with iterative and forall loops. We have shown how opportunities for storage reuse are detected by the scheduler, and that subscript transformation may be performed so that 1) an apparently iterative formulation can be transformed into a parallel one from which a parallel loop can be generated, and 2) storage reuse can be applied to the transformed array.

Implementation effort is focussed on the following topics:

- Integration of the array subscript restructuring algorithm into the compiler.
- a graphical front end, which can translate Equation 1 or Equation 2 into PS.
- Improvement of the scheduler to better merge iterative loops.

Acknowledgements Many thanks to those who have participated in the design and implementation of PS: Thomas J. Myers, Sue Samet, Eric Su, Todd Torgersen, Yuh-Dong Tsai, Ed Wyatt.

6 Bibliography

[1] Allen, J., "Dependence Analysis for Subscripted Variables and its Application to Program Transformations," Ph.D dissertation, Rice University, 1983.

[2] Ashcroft, E. and Wadge, W., "Lucid, A Nonprocedural Language with Iteration," *Communications of the ACM*, July 1977.

[3] Banerjee, et. al. Time and parallel processing bounds for Fortran-like loops," *IEEE Trans. on Computers*, Sept. 1979.

[4] Cytron, R., "Doacross: Beyond Vectorization for Multiprocessors," *International Conference on Parallel Processing* 1986, pp. 836-844.

[5] Gokhale, M., "Generating Parallel Programs from Nonprocedural Specifications," 4th Jerusalem Conference on Information Technology 1984.

[6] Gokhale, M., "Algorith Specification in a Very High Level Language," Institute for Computer Applications in Science and Engineering Report 86-67, 1986.

[7] Jordan, Harry, "The Force on the Flex: Global Parallelism and Portability," Institute for Computer Applications in Science and Engineering Report 86-54, 1986.

[8] Kuck, D., et. al., "Dependence Graphs and Compiler Optimization," *Proc. of 8th Annual Symposium on Principles of Programming Languages*, 1981.

[9] Kuck, et. al., "Analysis and transformation of programs for parallel computation," Proc. of 4th Int'l Computer Software and Applications Conference, IEEE, 1980.

[10] Lamport, L., "The Parallel Execution of Do Loops," *Communications of the ACM* February 1974.

[11] Lu, K.-S., "MODEL Program Generator: System and Programming Documentation," Technical Report U. of Pennsylvania, 1982.

[12] McGraw, Jim, "SISAL: Streams and Iteration in a Single Assignment Language," Lawrence Livermore National Laboratory Report M-146, Revision 1, 1985.

[13] Midkiff, S. P., et. al., "Compiler Generated Synchronization for DO Loops," *International Conference on Parallel Processing* 1986, pp. 544-551.

[14] Myers, T. and Gokhale, M., "Parallel Scheduling of Recursively Defined Arrays," *Journal of Symbolic Computation*, to appear. Also available as Institute for Computer Applications in Science and Engineering Report 86-66, 1986.

[15] Pneuli, A. and Prywes, N., "Scheduling Equational Specifications and Nonprocedural Programs," Chapter 13 in *Automatic Program Construction Techniques*, MacMillan 1984.

[16] Polychronopoulos, C. D., et. al., "Execution of Parallel Loops on Parallel Processor Systems," *International Conference on Parallel Processing* 1986, pp. 519-527.

[17] Prywes, N., et. al., "Compilation of Nonprocedural Specifications into Computer Programs," IEEE *Transactions on Software Engineering*, May 1983.

[18] Prywes, N. et. al., "Programming Supercomputers in an Equational Language," First International Conference on Supercomputing Systems, 1985.

[19] Shi, Y., " Very-High Level Concurrent Programming," Ph.D dissertation in Computer and Information Sciences, U. of Pennsylvania, 1984.

Standard Bibliographic Page

1. Report No. NASA CR-178277 ICASE Report No. 87-23	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle EXPLOITING LOOP LEVEL PARALLELISM IN NONPROCEDURAL DATAFLOW PROGRAMS		5. Report Date April 1987 6. Performing Organization Code
7. Author(s) Maya B. Gokhale		8. Performing Organization Report No. 87-23
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		10. Work Unit No. 11. Contract or Grant No. NAS1-18107
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		13. Type of Report and Period Covered Contractor Report 14. Sponsoring Agency Code 505-90-21-01

15. Supplementary Notes

Langley Technical Monitor:
J. C. South

Final Report

Submitted to Proceedings of the International Conference on Parallel Processing

16. Abstract

In this paper, we discuss how loop level parallelism is detected in a nonprocedural dataflow program and how a procedural program with concurrent loops is scheduled. In addition, we discuss a program restructuring technique which may be applied to recursive equations so that concurrent loops may be generated for a seemingly iterative computation. A compiler which generates C code for the language described below has been implemented. We describe the scheduling component of the compiler and the restructuring transformation.

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
parallel scheduling, dataflow, automatic program generation, program transformation, hyperplane	62 - Computer Systems Unclassified - unlimited

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	19	A02

For sale by the National Technical Information Service, Springfield, Virginia 22161

NASA Langley Form 63 (June 1985)