PARAFASE RESTRUCTURING OF FORTRAN CODE FOR PARALLEL PROCESSING*

Atul Wadhwa  
Sverdrup Technology, Inc.  
(Lewis Research Center Group)  
NASA Lewis Research Center

ABSTRACT

Because of the existence of large and cumbersome computer codes, there is a need to find new and more efficient ways of performing structural computations. Today, there is a heavy emphasis on emerging parallel processing methods. A research effort is in progress at NASA Lewis to develop these methods to reduce time and cost of program execution. Restructuring FORTRAN codes to take advantage of parallel processing architecture is a part of this effort. An automatic code restructurer, Parafrase, is used to meet this effort. Parafrase, developed at the University of Illinois, transforms a FORTRAN code, subroutine by subroutine, into a parallel code for a vector and/or shared-memory multiprocessor system. Parafrase is not a compiler; it transforms a code and provides information for a vector or concurrent process.

Parafrase uses a data dependency to reveal parallelism among instructions. The data dependency test distinguishes between recurrences and statements that can be directly vectorized or parallelized. A number of transformations are required to build a data dependency graph.

The purpose of this presentation is to give an overview of the Parafrase restructuring approach. Specifically, key aspects of the Parafrase program (such as data dependence tests and machine-dependent transformations) will be discussed.

*Work performed on-site at the Lewis Research Center for the Structural Dynamics Branch.
ELEMENTS OF PARALLEL PROCESSING

Some current computer software packages written in sequential codes (i.e., existing FORTRAN) have an undesirable turnaround time. Parallel processing can minimize execution time by employing vector or concurrent events in the computing process. The most common terms characterizing parallel processing are vector and multiprocessing.

In vector processing a loop can be vectorized if each statement of the loop can be executed for the entire index set of the loop before executing the next statement and producing the same result. Multiprocessing refers to a system with two or more processors. There are basically two types of multiprocessing systems, a shared-memory system and a message-passing system. In the shared-memory system, all the processors exchange information through the shared-memory; while in the message-passing system, each processor has its own private memory, and each can communicate and synchronize with others through some network connection.
Parafrase is a restructuring tool that transforms a code, subroutine by subroutine, to take advantage of the parallelism available in a particular machine (Kuck et al., 1984). Parafrase accepts an input program in FORTRAN, analyzes its data dependency, then targets the detected parallelism on vector processor (Single Execution Array (SEA)) or multiprocessor (Multiple Execution Scalar (MES)). To determine if a loop can be parallelized, Parafrase builds a graph of the data dependencies; the nodes represent program statements and the edges represent data and control dependencies. Parafrase output is useful in analyzing and evaluating parallel programs.
DATA DEPENDENCY TEST

Detecting parallelism in a code requires data dependency testing, which reveals information about data computation and use in the program. The data dependency test determines whether or not a statement uses a value that was computed on previous iteration. There are four types of dependencies: flow, antidependence, output dependence, and control dependence (Wolfe, 1982). These dependencies must be considered to detect recurrences.

\[
\begin{align*}
S1: & \quad A(I) = B(I) + C(I) \\
S2: & \quad D(I) = A(I) + 5 \\
S3: & \quad C(I + 1) = A(I) + B(I) \\
S4: & \quad \text{IF } D(I) > 10 \text{ THEN} \\
S5: & \quad C(I + 1) = B(I) + 5
\end{align*}
\]
Parafrase uses a data dependency test to detect parallelism. A number of machine-independent passes are required to build an effective data dependency graph. Some of the important passes are as follows: DO loop normalization, induction variable substitution, statement forward substitution, and dead-code elimination.
The first of machine-independent passes is the DO-loop normalization (Polychronopolus, 1986). A DO loop normalization transforms loops in such a way that the induction variables of each loop increase by one, starting from one, to some upper bound. Every old induction variable within the loop is replaced by the new induction variable.

**ORIGINAL CODE**

```
DO 20 I = 1,100
   NI = I
   DO 10 J = 1,100,3
      NI = NI + 2
      X(J) = Y(J)*Z(NI)
      Y(J+1) = Y(J) + Z(NI)
      X(J) = X(J) + Y(J)
   10 CONT
20 CONT
```

**REVISED CODE #1**

```
DO 20 I = 1,100
   NI = I
   DO 10 J = 1,(100 + 2)/3
      NI = NI + 2
      X(J*3 - 2) = Y(J*3 - 2)*Z(NI)
      Y(J*3 - 2) = Y(3*J - 2) + Z(NI)
      X(J*3 - 2) = X(J*3 - 2) + Y(J*3 - 2)
   10 CONT
20 CONT
```
INDUCTION VARIABLE SUBSTITUTION

Induction variables are used inside loops to simplify subscripts to linear functions of loop index variables. Detection and elimination of these variables reduce the number of operations. This transformation may also allow vectorization and parallelization of the loop, which would have been impossible because of the dependence cycle (when two statements are closely coupled). The discovery of induction variables is required since the data dependency test needs the array subscripts to be in terms of the loop index variables.

REVISED CODE #1

DO 20 I = 1,100
   NI = I
   DO 10 J = 1,(100 + 2)/3
      NI = NI + 2
      X(J*3 - 2) = Y(J*3 - 2)*Z(NI)
      Y((J*3 - 2) + 1) = Y(3*J - 2) + Z(NI)
      X(J*3 - 2) = X(J*3 - 2) + Y(J*3 - 2)
   10 CONT
20 CONT

REVISED CODE #2

DO 20 I = 1,100
   NI = I
   DO 10 J = 1,(100 + 2)/3
      NI = I + 2*J
      X(J*3 - 2) = Y(J*3 - 2)*Z(NI)
      Y((J*3 - 2) + 1) = Y(3*J - 2) + Z(NI)
      X(J*3 - 2) = X(J*3 - 2) + Y(J*3 - 2)
   10 CONT
20 CONT
STATEMENT FORWARD SUBSTITUTION

A statement forward substitution replaces integer expressions and constants into subscripts. A scalar variable that is assigned a value, and is used in a subscript, is replaced by an expression. The statement forward substitution eliminates the need for compiler or user temporaries. This transformation provides more information for the data dependency test.

REVISED CODE #2

DO 20 I = 1,100
   NI = I
   DO 10 J = 1,(100 + 2)/3
      NI = I + 2*J
      X(J*3 - 2) = Y(J*3 - 2)*Z(NI)
      Y((J*3 - 2) + 1) = Y(3*J - 2) + Z(NI)
      X(J*3 - 2) = X(J*3 - 2) + Y(J*3 - 2)
   10 CONT
20 CONT

REVISED CODE #3

DO 20 I = 1,100
   NI = I
   DO 10 J = 1,(100 + 2)/3
      NI = I + 2*J
      X(J*3 - 2) = Y(J*3 - 2)*Z(I + 2*J)
      Y((J*3 - 2) + 1) = Y(3*J - 2) + Z(I + 2*J)
      X(J*3 - 2) = X(J*3 - 2) + Y(J*3 - 2)
   10 CONT
20 CONT
DEAD-CODE ELIMINATION

The dead-code elimination removes the statements whose output is never used. This transformation reduces the number of computations by eliminating unnecessary calculations. These transformations convert as many subscripts as possible to a linear function of DO loop induction variables.

REVISED CODE #3

DO 20 I = 1,100
   NI = I
   DO 10 J = 1,(100 + 2)/3
       NI = I + 2*J
       X(J*3 - 2) = Y(J*3 - 2)*Z(I + 2*J)
       Y((J*3 - 2) + 1) = Y(3*J - 2) + Z(I + 2*J)
       X(J*3 - 2) = X(J*3 - 2) + Y(J*3 - 2)
   10 CONT
20 CONT

REVISED CODE #4

DO 20 I = 1,100
   DO 10 J = 1,(100 + 2)/3
       X(J*3 - 2) = Y(J*3 - 2)*Z(I + 2*J)
       Y((J*3 - 2) + 1) = Y(3*J - 2) + Z(I + 2*J)
       X(J*3 - 2) = X(J*3 - 1) + Y(J*3 - 2)
   10 CONT
20 CONT
VECTORIZATION AND PARALLELIZATION

After building the data dependency graph, Parafrase starts restructuring a code from serial to parallel form. If the data dependency relation prevents loop vectorization or parallelization, then several machine-dependent transformations on the loop would be attempted. Loop interchanging and loop fission are two of the important passes.

AFTER BUILDING A DATA DEPENDENCY GRAPH, PARAFRASE STARTS TO RESTRUCTURE THE PROGRAM FROM SERIAL TO PARALLEL FORM. FOR THESE TRANSFORMATIONS A NUMBER OF PASSES ARE TARGETED FOR A VECTOR (SINGLE EXECUTION ARRAY) OR A MULTIPROCESSOR (MULTI-EXECUTION SCALAR) SYSTEM. TWO OF THESE PASSES ARE AS FOLLOWS:

1. LOOP INTERCHANGING
2. LOOP FISSION
   a) INSIDE-OUT LOOP DISTRIBUTION
   b) DO-LOOP SPREADING
The first transformation is loop interchanging, a switching of inner and outer loops. Loop interchanging may be used to vectorize the inner loop or to parallelize the outer loop. Loop interchanging is impossible when two or more statements of the loop are dependent with "<" and">" directions.

**REVISED CODE #4**

```plaintext
DO 20 I = 1,1100
  DO 10 J = 1,(100 + 2)/3
    X(J*3 - 2) = Y(J*3 - 2)*Z(I + 2*J)
    Y((J*3 - 2) + 1) = Y(3*J - 2) + Z(I + 2*J)
    X(J*3 - 2) = X(J*3 - 1) + Y(J*3 - 2)
  10 CONT
20 CONT
```

**REVISED CODE #5**

```plaintext
DO 10 J = 1,102/3
  DO 20 I = 1,1100
    X(J*3 - 2) = Y(J*3 - 2)*Z(I + 2*J)
    Y((J*3 - 2) + 1) = Y(3*J - 2) + Z(I + 2*J)
    X(J*3 - 2) = X(J*3 - 1) + Y(J*3 - 2)
  20 CONT
10 CONT
```
Some loops can be divided into two or more loops, a process known as loop fission. If the statement forward substitution does not remove the data dependence cycle, then the loop fission may be allowed to vectorize or parallelize a part of the loop. Inside-out loop distribution is used for the case of the vector operation machine (Allen and Kennedy, 1982), and Doall Loop Distribution is used for the multiprocess machine.

REVISED CODE #4

```
DO 20 I = 1,100
   DO 10 J = 1,(100 + 2)/3
      X(J*3 - 2) = Y(J*3 - 2)*Z(I + 2*J)
      Y(J*3 - 2) = Y(J*3 - 2) + Z(I + 2*J)
      X(J*3 - 2) = X(J*3 - 2) + Y(J*3 - 2)
   10 CONT
20 CONT
```

FINAL REVISED VECTOR CODE

```
DO 20 J = 1,100
   DO 10 I = 1,102/3
      X(J*3 - 2) = Y(J*3 - 2)*Z(I + 2*J)
      Y(J*3 - 2) = Y(J*3 - 2) + Z(I + 2*J)
      X(J*3 - 2) = X(J*3 - 2) + Y(J*3 - 2)
   10 CONT
20 CONT
```
DOALL LOOP DISTRIBUTION

Doall loop distribution for a multiprocessor machine is a spreading of the loop iteration across multiple processors. This transformation detects if each loop iteration can be executed independently of the others.

REVISED CODE #5

DO 10 J = 1,102/3
   DO 20 I = 1,100
      X(J*3 - 2) = Y(J*3 - 2)*Z(I + 2*J)
      Y((J*3 - 2) + 1) = Y(3*J - 2) + Z(I + 2*J)
      X(J*3 - 2) = X(J*3 - 1) + Y(J*3 - 2)
   10 CONT
   20 CONT

FINAL REVISED CONCURRENT CODE

DOALL 30 J = 1,102/3
   DO 10 I = 1,100
      X(J*3 - 2) = Y(J*3 - 2)*Z(I + 2*J)
      X(J*3 - 2) = X(J*3 - 1) + Y(J*3 - 2)
   10 CONT
   30 CONT

DOALL 30 J = 1,102/3
   DO 20 I = 1,100
      Y((J*3 - 2) + 1) = Y(3*J - 2) + Z(I + 2*J)
   20 CONT
   30 CONT
Parafrase was used to restructure a rotor dynamics SOLVE subroutine, as well as initialization INIT subroutine. The estimate of speedup values has been computed by Parafrase. The restructured subroutines were executed on CFT, a CRAY compiler, which detected more vector operations than the original code.

**SPEEDUP COMPARED WITH NUMBER OF VECTOR PROCESSORS**

![Graph comparing speedup with number of vector processors for SOLVE and INIT subroutines.]

**SPEEDUP COMPARED WITH NUMBER OF MULTIPROCESSORS**

![Graph comparing speedup with number of multiprocessors for SOLVE and INIT subroutines.]

CD-88-31957

CD-88-31958

1-444
SPEEDUP OF A RESTRUCTURED CODE

Kuck, et al., has applied Parafrase to EISPACK subroutines, an eigenvalue/eigenvector conjecture. After restructuring them, Parafrase calculated the speedup values of subroutines.

SPEEDUP VALUES OBTAINED BY KUCK'S GROUP ON EISPAK

<table>
<thead>
<tr>
<th>SUBROUTINE NAME</th>
<th>NUMBER OF PROCESSORS, 32</th>
<th>NUMBER OF PROCESSORS, 256</th>
<th>NUMBER OF PROCESSORS, 2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELMBAK</td>
<td>31.9</td>
<td>242.0</td>
<td>668.0</td>
</tr>
<tr>
<td>ELMHES</td>
<td>31.7</td>
<td>33.6</td>
<td>33.6</td>
</tr>
<tr>
<td>ETRAN</td>
<td>29.3</td>
<td>71.3</td>
<td>84.5</td>
</tr>
<tr>
<td>TRED1</td>
<td>31.0</td>
<td>235.0</td>
<td>240.0</td>
</tr>
<tr>
<td>TRED2</td>
<td>18.3</td>
<td>36.5</td>
<td>39.5</td>
</tr>
<tr>
<td>CBABK2</td>
<td>30.0</td>
<td>53.5</td>
<td>57.4</td>
</tr>
<tr>
<td>COMBAK</td>
<td>31.9</td>
<td>248.5</td>
<td>721.0</td>
</tr>
<tr>
<td>CORTB</td>
<td>32.0</td>
<td>254.0</td>
<td>1250.0</td>
</tr>
<tr>
<td>CORTH</td>
<td>32.0</td>
<td>252.0</td>
<td>501.0</td>
</tr>
<tr>
<td>BANDV</td>
<td>31.0</td>
<td>98.0</td>
<td>98.0</td>
</tr>
</tbody>
</table>

CD-88-31959
Most existing code compilers take advantage of parallel processing, but don't perform code restructuring. To achieve effective and efficient use of parallel processing architecture, existing codes have to be restructured. Parafrase is a FORTRAN code-restructuring tool. It is not a compiler. It produces information for vector or shared-memory processing systems. Parafrase has been applied to subroutines of the Rotor Rub Dynamics code. The restructured output code has been executed on a CRAY compiler, which found more vector operations than the original code.

- To achieve fast execution, restructuring of a sequential code is needed for parallel processing
- Parafrase can be used as a restructuring tool
- Parafrase optimizes a code for the data dependency test
- The output of Parafrase can be modified for vector or shared-memory architecture
REFERENCES


