PARAFRASE 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.

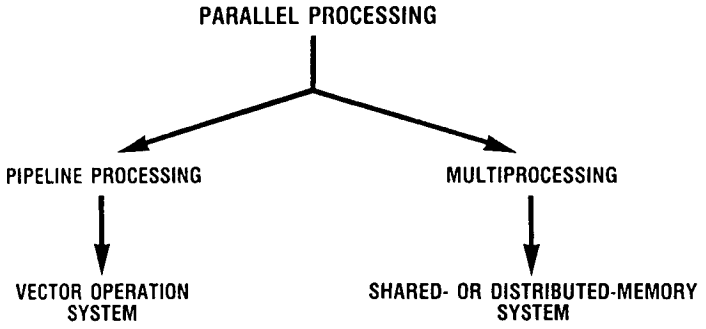
---

*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.



CD-88-31937

VECTOR PROCESSING

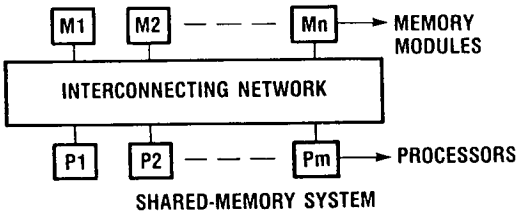
SERIAL PROCESS

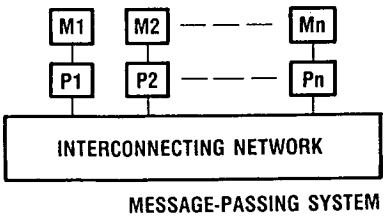
```
DO 10 I = 1,N
  A(I) = B(I) + C(I)
  B(I) = N*C(I)
10 CONTINUE
```

VECTOR PROCESS

```
A(1:N) = B(1:N) + C(1:N)
B(1:N) = N*C(1:N)
```

MULTIPROCESSING

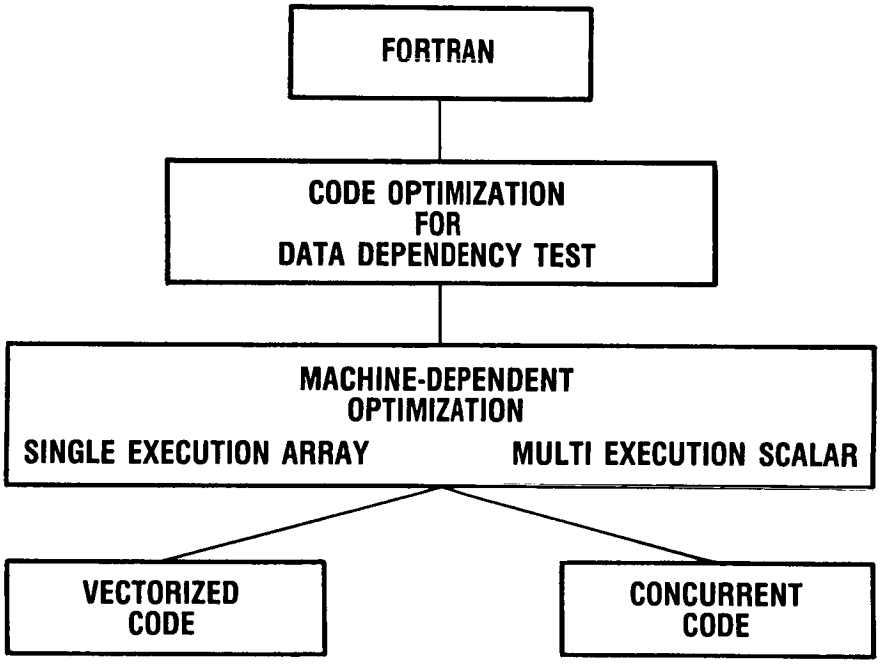




CD-88-31938

PARAFRASE: AN AUTOMATIC CODE RESTRUCTURER

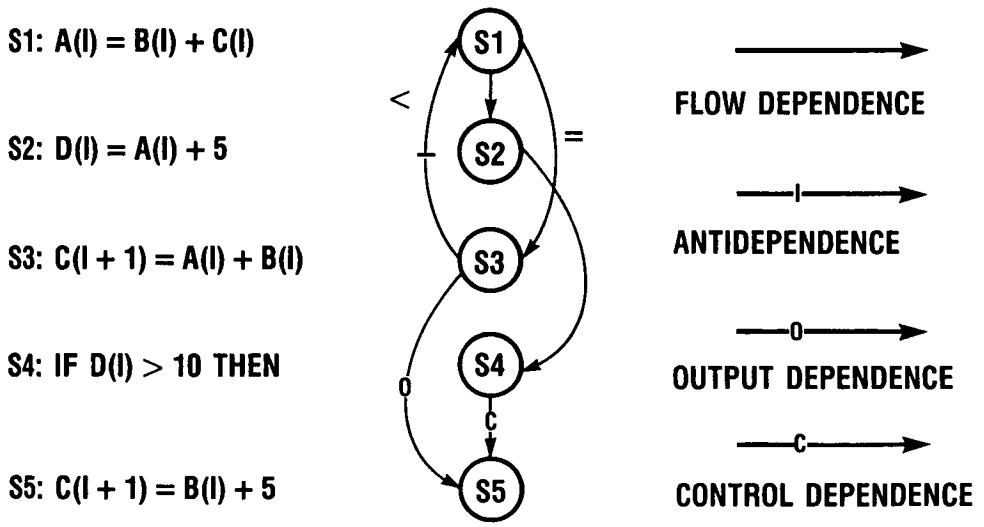
Parafrase is a restructuring tool that transforms a code, subroutine by subroutine, to take advantage of the parallelism available in a particular machine (Kuck et al., 1984). Parafrase accepts an input program in FORTRAN, analyzes its data dependency, then targets the detected parallelism on vector 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.



CD-88-31939

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.



CD-88-31940

TRANSFORMATION FOR DATA DEPENDENCE TEST

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.

A NUMBER OF TRANSFORMATIONS ARE REQUIRED TO BUILD A DATA DEPENDENCY GRAPH. THESE TRANSFORMATIONS ARE MACHINE-INDEPENDENT TRANSFORMATIONS. SOME OF THE IMPORTANT PASSES ARE

1. **DO-LOOP NORMALIZATION**
2. **INDUCTION VARIABLE SUBSTITUTION**
3. **STATEMENT FORWARD SUBSTITUTION**
4. **DEAD-CODE ELIMINATION**

CD-88-31941

DO LOOP NORMALIZATION

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) + 1) = Y(3*J - 2) + Z(NI)
    X(J*3 - 2) = X(J*3 - 2) + Y(J*3 - 2)
  10 CONT
20 CONT
```

CD-88-31942

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
```

CD-88-31943

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
```

CD-88-31944

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
```

CD-88-31945

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**

CD-88-31850

LOOP INTERCHANGING

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

```
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
```

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)
20 CONT
10 CONT
```

CD-88-31951

INSIDE-OUT LOOP DISTRIBUTION

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) + 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
```

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)
10 CONT
  DO 10 J = 1,102/3
    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
```

CD-88-31953

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
```

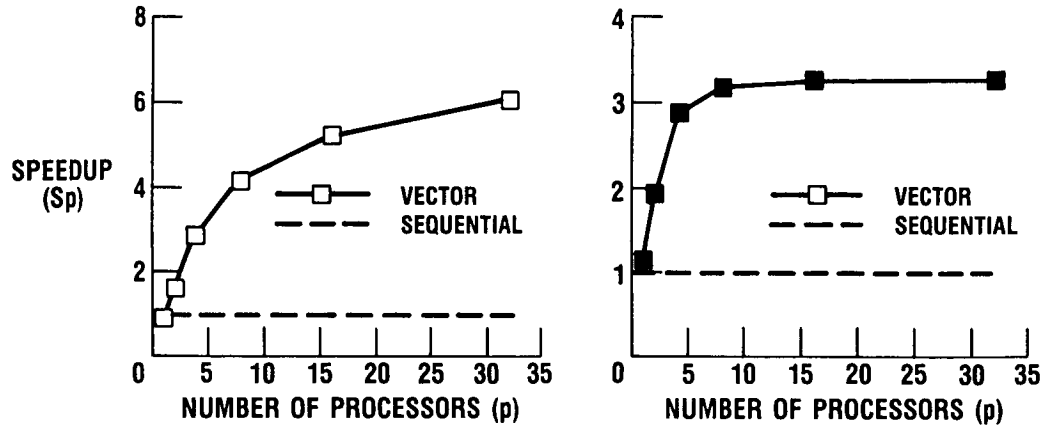
FINAL REVISED CONCURRENT CODE

```
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
```

CD-88-31955

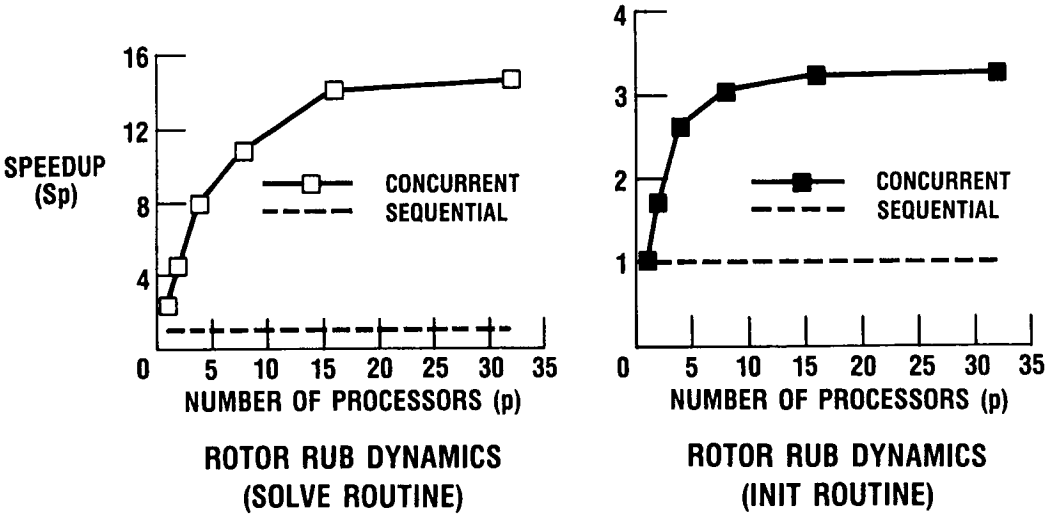
Parafrase was used to restructure a rotor dynamics SOLVE subroutine, as well as initialization INIT subroutine. The estimate of speedup values has been computed by Parafrase. The restructured subroutines were executed on CFT, a CRAY compiler, which detected more vector operations than the original code.

SPEEDUP COMPARED WITH NUMBER OF VECTOR PROCESSORS



CD-88-31957

SPEEDUP COMPARED WITH NUMBER OF MULTIPROCESSORS



CD-88-31958

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

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

CD-88-31959

SUMMARY

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**

CD-88-31956

REFERENCES

Allen, J.R., and Kennedy, K., 1982, PFC: A Program to Convert Fortran to Parallel Form, Report MASC-TR82-6. Rice Univ., Houston Texas.

Kuck, D.J. et al., 1984, "The Effect of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance," Proceedings of the 1984 International Conference on Parallel Processing, R.M. Keller, ed., IEEE, New York, pp. 129-138.

Kuck, D.J., 1978, The Structure of Computers and Computations, John Wiley and Sons, New York.

Leu, J.S., Agrawal, D.P., and Mauney, J., 1987, "Modeling of Parallel Software for Efficient Computation-Communication Overlap," to be published.

Midkiff, S.P., and Padua, D.A., 1986, "Compiler Generated Synchronization for Loops," Proceedings of the 1986 International Conference on Parallel Processing, K. Hwang, S.M. Jacobs, and E.E. Swartzlander, eds., IEEE, New York, pp. 544-551.

Padua, D.A., Kuck, D.J., and Lawrie, D.H., 1980, "High-Speed Multiprocessors and Compilation Techniques," IEEE Transaction on Computers, Vol. 29, No. 9, pp. 763-776.

Polychronopolus, C.D., and Banerjee, U., 1986, "Speedup Bounds and Processor Allocation of Parallel Programs on Multiprocessor Systems," Proceedings of the 1986 International Conference on Parallel Processing, K. Hwang, S.M. Jacobs, and E.E. Swartzlander, eds., IEEE, New York, pp. 554-551.

Wolfe, M.J., 1982, "Optimizing Supercompilers for Supercomputers," Ph.D. Thesis, University of Illinois at Urbana-Champaign, Illinois.