Organization of Parallelizing Compilers

(병렬화 컴파일러의 구조)

B.-M. Chang* J. K. Lee** D. Chi***
(창병모, 이재경, 지동해)

Wide variety of the architectural complexity of parallel computers often makes it difficult to develop efficient programs for them. One of approaches to improve this difficulty is to program in familiar sequential languages such as Fortran or C and to parallelize sequential programs into equivalent parallel programs automatically. This paper presents an organization of parallelizing compiler which transforms sequential programs into equivalent parallel programs. The parallelizer consists mainly of syntax analysis, control and data flow analysis, dependence analysis, program transformations, and parallel code generation. In particular, the program restructuring in this parallelizer maximizes loop parallelism

I. Introduction

Developing efficient programs for many parallel computers is difficult because of the architectural complexity of those machines. Furthermore, the wide variety of machine organizations often makes it more difficult to port an existing program than to reprogram completely.

One of approaches to improve this situation is to program in familiar sequential languages such as Fortran or C and to parallelize sequential programs into equivalent parallel programs automatically [1-3]. This approach frees the programmer from concerns about the specific characteristics of the target parallel machine. There are the following reasons why the parallelization of sequential programs is important :

- there are many sequential programs that would be convenient to be executed on parallel computers.

* 병렬처리연구실 post-doc 연구원
** 병렬처리연구실 선임연구원
*** 병렬처리연구실 실장

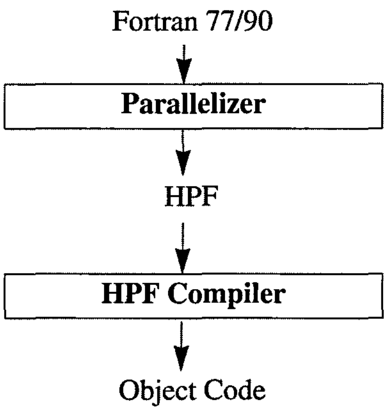


Fig. 1. Fortran parallelizer.

• powerful parallelizers should facilitate programming by allowing to develop programs in a familiar sequential programming language such as Fortran or C.

• much of experience about parallelization should be applicable to the other translation problems.

This paper presents an organization of parallelizing translator, which transforms sequential programs into equivalent parallel programs. As an example, we will develop a parallelizer which transforms programs in Fortran 77 or 90 into equivalent parallel programs in High Performance Fortran (HPF) [4]. HPF is a newly designed language for parallel processors, which is a superset of Fortran 90. It provides directives for data layout, two parallel loops FORALL and INDEPENDENT DO loops, and additional intrinsics for data parallel operations. The transformed parallel programs will be compiled into object code by parallel backend compiler. This relationship is shown in Fig. 1. Since this paper aims to introduce an overall organization and some theoretical foundations of the parallelizer, they are not applied only to Fortran parallelizer, but also to parallelizer for other procedural languages. So, Fortran syntax is not necessarily used in the following.

We also give an overview of techniques for parallelizers which transform sequential programs into equivalent parallel programs. The parallelizer consists mainly of syntax analysis, control and dataflow analysis, dependence analysis, program transformations, and parallel code generation. Parallelizers for high-performance parallel processors maximize parallelism and memory locality, mostly by tracking the properties of arrays using loop dependence analysis, while most optimizations for uniprocessors reduce the number of instructions executed by the program. In particular, the program restructuring in this compiler maximizes loop parallelism.

The organization of the rest of this paper is as follows. The next section gives a brief overview of organization of parallelizing compilers. Section III presents a description on analysis part for dataflow analysis, dependence analysis, etc. Section IV presents program restructuring techniques to improve parallelism. Section V describes a unified restructuring framework which maximizes loop parallelism. Section VI concludes the paper.

II. Organization of Parallelizers

We have presented a diagram of the various parts of a parallelizer in Fig. 2. We have tried to present one reasonable organization.

The first two phases, lexical analysis and parsing, constitute the front-end of the compiler. The front-end converts the original source programs into more convenient internal data structures and checks whether the static semantic constraints of the language have been properly satisfied. The parser generally produces an *abstract syntax tree*(AST), a symbol table, and information needed for basic interprocedural analysis.

The next stage of parallelizers is analysis, which consists of several phases of analyses. The first step is to produce *control-flow graph* (CFG) by *control flow* analysis. The CFG converts the different kinds of control transfer constructs in the program into a single form that is easier for the compiler to manipulate. After control flow has been dealt with, *dataflow analysis* and *dependence analysis* are to be done in order to examine how data is being used in the program. There are a variety of representations for

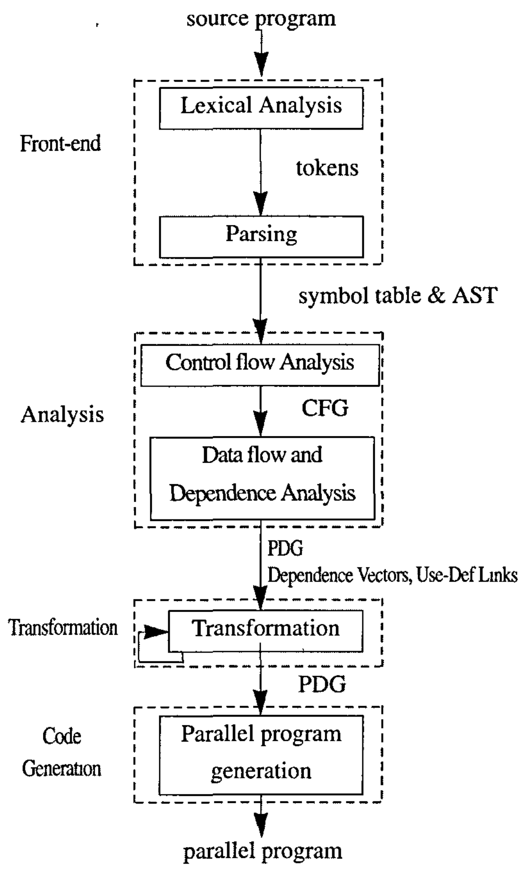
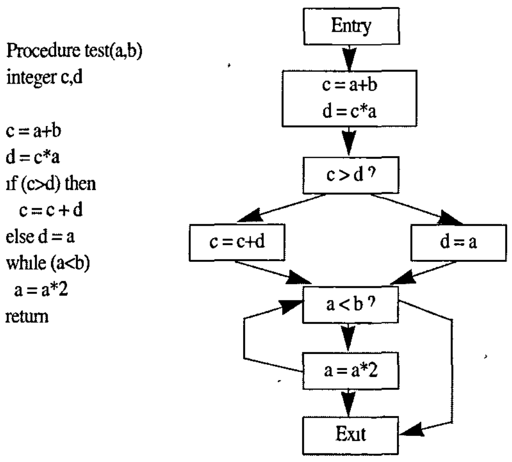


Fig. 2. Organization of a parallelizer.



```
Procedure test(a,b)
integer c,d

c = a+b
d = c*a
if (c>d) then
  c = c + d
else d = a
while (a<b)
  a = a*2
return
```

Fig. 3. Control flow graph.

capturing flow information such as *program dependence graphs* [5], *static single-assignment form* [6] and *dependence vectors* .

With this information, compilers can often automatically detect parallelism in loops, or report the user specific reasons why a particular loop can not be executed in parallel. Additional performance improvement in parallelism and memory locality can be achieved by a series of restructuring transformations, such as loop interchange, reversal, skewing, tiling (or blocking), etc.

Once the program has been fully transformed, the last stage of compilation is to convert it into an equivalent parallel program, which is in a parallel language like HPF, or in a sequential language with runtime library for parallel processing.

III. Analysis

1. Control Flow Analysis

There are a wide variety of representations that have been developed to simplify analysis. The most basic is the control flow graph (CFG) [7]. The CFG is a directed graph containing one node for each basic block in the program, plus two distinguished nodes called **Entry** and **Exit**. Each node has an edge to every node to which it can transfer control. The **Entry** node has an edge to every basic block that represents an entry point of the code; there is an edge to **Exit** from any basic block that can cause an exit. Figure 3 shows a procedure and its control flow graph.

The control flow analysis builds the control flow graph of the program and determines the *interval* regions of this graph. The control flow graph is the basis for dataflow analysis and for the construction of control dependence graph. An interval is a single-entry, strongly connected region in which every cycle includes the entry node. The interval partitioned control flow graph has been used as a space-and time-efficient basis for dataflow analysis and program transformations.

2. Dataflow Analysis

The parallelizer performs interprocedural analyses before intraprocedural dataflow analysis. Interprocedural dataflow analyses are done based on the *call graph* , which consists of nodes representing routines and edges representing calls, with one edge for each call site. Parallelizer generally performs the following three interprocedural analyses :

- Interprocedural alias analysis computes the set of potentially aliased variable pairs for each procedure in the program.
- Interprocedural constant propagation determines when a formal parameter will

always have the same constant value upon entrance to its procedure. This information sharpens the intraprocedural constant propagation analysis. In parallelizing compilers, the result of constant propagation will improve the sharpness of dependence analyses to come.

Interprocedural side effects analysis determines modified variables and used variables by each call.

Within procedure or functions, dataflow analysis, constant propagation, and linear induction variable detection are generally performed in parallelizers. Dataflow analysis [8] was one of the first strategies for analyzing program behavior. It is usually performed on a control flow graph; it attempts to track the flow of data through the program's variables and to characterize the values of variables at various points of execution. The primary purpose of dataflow analysis is the consideration of def-use and use-def chains [7]. Def-use chaining determines, for every definition of variables, the list of all possible uses of that definition. Use-def chaining is defined in the same manner. The reaching definitions and live variable global dataflow problems are solved by the interval analysis dataflow technique [9]. In order to increase precision of the def-use and def-def chains, the interprocedural alias analysis is done in the program before the dataflow analysis is started. Def-use and use-def chains will be a basis of dependence analysis to be described in the next section.

Intraprocedural constant propagation is a well-known optimization which determines the definitions which are integer constants to improve the subsequent phases of linear induction variable detection, data dependence analysis, and potential parallel process identification. In order to accomplish this goal, our constant propagator uses interprocedurally determined constants. The linear induction variable detection finds mutually defined linear induction variables and also determines the scope of such variables for dependence analysis.

3. Dependence Analysis

Among the various forms of analysis, parallelizing compilers heavily rely on *dependence analysis* [10, 11]. A dependence is a relationship between two computations that places constraints on their execution order. Dependence analysis identifies these constraints, which are then used to determine whether a particular transformation can be applied without changing the semantics of the computation. There are two kinds of dependences: *control dependence* and *data dependence* .

Definition 1 (Data dependence)

Given two statements S and T, such that S precedes T in the sequence,

- *T is flow dependent on S, denoted by $S\delta^f T$, iff there exists a simple variable v such that S assigns a value to v and T fetches the value of v(def-use ordering).*
- *T is anti dependent on S, denoted by $S\delta^a T$, iff there exists a simple variable v such that T assigns a value to v and S fetches the value of v(use-def ordering).*
- *T is out dependent on S, denoted by $Sd^o T$, iff there exists a simple variable v such that both T and S assign a value to v.*
- *T is data dependent on S, denoted by $S\delta T$, iff $S\delta^f T$, $S\delta^a T$, or $S\delta^o T$.*

The control dependence relation represents the part of the control structure of the source program that is important to determine which transformations are valid. The definition that is most frequently used today is that of Ferrante *et al.* [5]. They define control dependence relation in terms of control-flow graphs (CFG).

Definition 2 (Control dependence)

In control-flow graphs, a node T of a control-flow graph is said to be control dependent on a node S if

1) there is a path from S to T whose internal nodes are all postdominated by T; and

2) T does not postdominate S.

Intuitively, the control dependence means that the outcome of S determines whether or not T executes.

For data-dependence computation in actual programs, the most common situation occurs when we are comparing two variables in a single loop and those variables are elements of a one-dimensional array, with subscripts linear in the loop index variable, as in the following model:

```
do I = p, q
S:    X(a * I + a_0) = ...
T:    ... = ... X(B * I + b_0) ...
end do
```

Here, X is a one-dimensional array; p, q, a, a_0, b, and b_0 are integer constants known at compile time; and both a and b are nonzero. We want to find out if the output variable of statement S and the input variable of statement T cause a data dependence between T and S.

The instance of the variable $X(aI + a_0)$ for an index value $I = i$ is $X(ai = a_0)$, and the instance of the variable $X(bI + b_0)$ for an index value $I = j$ is $X(bj + b_0)$. These two instances will represent the same memory location iff

$$ai - bj = b_0 - a_0. \qquad (1)$$

Since i and j are values of the index variable I, they must be integers and lie in the range :

$$p \leq i \leq q$$
$$p \leq j \leq q. \qquad (2)$$

Suppose (i,j) is an integer solution to (1) that also satisfies (2). If $i \leq j$, then the instance $S(i)$ of S is executed before the instance $T(j)$ of T in the sequential execution of the program. Hence, $S(i)$ first puts a value in the memory location defined by both $X(ai + a_0)$ and $X(bj + b_0)$, and then $T(j)$ uses that value. This makes the instance $T(j)$ flow dependent on the instance $S(i)$, and the statement T flow dependent on the statement S. Similarly, if $i > j$, then $S(i)$ is antidependent on $T(j)$ and S is antidependent on T.

Now we consider nested loops. Figure 4 shows a generalized perfect nest of d loops. The body of the loop nest reads and writes elements of the m-dimensional array a. The functions f_i and g_i map the current values of the loop iteration variables to an integer which indexes the i^{th} dimension of a. The generalized loop can give rise to any type of data dependence.

An iteration can be uniquely named by a vector of d elements $\vec{I} = (i_1, ..., i_n)$, where $l_p \leq i_p \leq u_p$. The outermost loop corresponds to the leftmost index.

We will describe how to find loop-carried dependence between the two references to a, and how to represent those dependences. Clearly, a reference in iteration $\vec{J}$ can only depend upon another reference in iteration $\vec{I}$ that was executed before it, not after it. We denote $\vec{I} < \vec{J}$ if the iteration $\vec{I}$ is executed before $\vec{J}$ in sequential execution. A reference in some iteration $\vec{J}$ depends upon a reference in iteration $\vec{I}$ when the values of the subscripts are the same in different iteration $\vec{I}$ and $\vec{J}$. If no such $\vec{I}$ and $\vec{J}$ exist, the two references are independent across all iterations of the loop.

```
do $i_1 = l_1, u_1$
  do $i_2 = l_2, u_2$
   . . .
      do $i_n = l_n, u_n$
        $a[f_1(i_1, ..., i_n), \cdots, f_m(i_1, \cdots, i_n)] = ...$
          $... = a[g_1(i_1, ..., i_n), ..., g_m(i_1, ..., i_n)]$
      end do
   . . .
  end do
```

Fig. 4. General loop nest.

Example 1 Each iteration of the inner loop writes the element $a[i, j]$. There is a dependence if any other iteration reads or writes the same element. Consider iterations $\vec{I} = (1,3)$ and $\vec{J} = (2,2)$. There is a flow dependence from iteration $\vec{I}$ to iteration $\vec{J}$ since iteration $\vec{I}$ writes the value $a[1,3]$ and this is read in the iteration $\vec{J}$.

```
do $i = 2, n$
  do $j = 1, n-1$
    $a[i,j] = a[i,j] + a[i-1, j+1]$
  end do
end do
```

When $\vec{I}\ \delta\ \vec{J}$, the *dependence distance* is defined as $\vec{J} - \vec{I} = (j_1 - i_1, ..., j_n - i_n)$. In the above example, the dependence distance is $\vec{J} - \vec{I} = (1, -1)$ When a dependence distance is used to describe the dependences for all iterations, it is called a distance vector [12, 13]. A legal distance vector must be *lexicographically positive*, meaning that the first nonzero element of the distance vector must be positive.

In some cases it is impossible to determine the exact dependence distance at compile-time. A *direction vector*[10] is commonly used to describe such dependences. For a dependence $\vec{I}\ \delta\ \vec{J}$, the direction vector $\vec{d} = (d_1, \ldots, d_n)$ where

$$d_p = \begin{cases} < \text{if } I_p < J_p \\ = \text{if } I_p = J_p \\ > \text{if } I_p > J_p \end{cases}$$

It is well known that a linear diophantine equation like (1) has a solution iff the gcd of the coefficients on the left-hand side divides the right-hand side. This fact can sometimes be used to settle a data-dependence question : it is called *gcd test*. There is a generalized gcd test using this property that works for a system of linear diophantine equation [11].

The λ-test [14] is an approximate test that tries to decide if there is a real solution to the whole system of data dependence equations satisfying the constraints. It assumes that no subscript tested can be formed by a linear combination of other subscripts.

The *I*-test [15] combines the approximate method and the gcd test. It isolates the case in which the approximate method is exact, and therefore can decide if there is an integer solution in that case. It is applicable when the array is one dimensional, and the coefficients of the data-dependence equation are small in a sense.

The Ω-test [16] uses an extension of the Fourier-Motzkin method to integer programming. Although its worst-case time complexity is exponential, it is claimed to be a fast and practical method for performing data dependence analysis.

IV. Program Transformation

Since parallelizing compilers were introduced, a lot of transformation techniques have been proposed to expose parallelism and improve memory locality [17]. In this section we describe one framework that is being actively investigated based on unimodular matrix theory[18]. It is applicable to any loop nest whose dependences can be described with a distance vector; a subset of the loops which requires a direction vector can also be handled. It is well known that loop transformations such as interchange, reversal, and skewing are useful in parallelization. These loop transformations can be modeled as elementary matrix transforma- tions; combinations of these transformations can be simply represented as products of the elementary transformation matrices. The optimization problem is thus to find the unimodular transformation that maximizes an objective function given a set of constraints.

Let us consider a loop interchange transformation to illustrate the unimodular transformation model. A loop interchange transformation maps iteration(i, j) to iteration(j, i). In matrix notation, this is written as

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} j \\ i \end{bmatrix}$$

The elementary permutation matrix thus performs the loop interchange transformation on the iteration space. A unimodular matrix has three important properties.

- it is square.
- it has all integer components.
- the absolute value of its determinant is one.

Because of these properties, the product of two unimodular matrices is unimodular, and the inverse of a unimodular matrix is unimodular.

Since a unimodular matrix performs a linear transformation on the iteration space, $T\vec{p_2} - T\vec{p_1} = T(\vec{p_2} - \vec{p_1})$. Therefore, if $\vec{d}$ is a distance vector in the original iteration space, then $T\vec{d}$ is a distance vector in the transformed iteration space.

There are three elementary transformations :

Permutation : A permutation σ on a loop nest transforms iteration $(p_1, \ldots, p_n)$ to $(p_{\sigma_1}, \ldots, p_{\sigma_n})$. This transformation can be expressed in matrix form as I_σ, the $n \times n$ identity matrix I with rows permuted by σ. The loop interchange is an n=2 example of the general permutation transformation. Loop interchanges exchange the position of two loops in a loop nest, generally moving one of the outer loops to the innermost position. It is one of the most powerful transformations and can improve performance in many ways. Among many advantages of loop interchange, it may be performed to improve parallelization by moving the independent loop with the largest range into the innermost position. General rule for loop interchange is in Fig. 5.

```
do $i = l_i, u_i$
    do $j = s * i + l_j, s * i + u_j$
        ...
    end do
end do
```

(a) Original loop

```
do $j = s * l_i + l_j, s * u_i + u_j$
do $i = max(l_i, (j - u_j + s - 1)/s), min(u_i, (j - l_j + s - 1)/s)$
      ...
    end do
end do
```

(b) Interchanged loop

Fig. 5 Loop Interchange: a kind of permutation.

Reversal : Reversal of the i-th loop is represented by the identity matrix, but with the i -th diagonal element equal to -1 rather than 1. For example, the matrix representing loop reversal of the outermost loop of a two-deep loop nest is

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

```
do $i = 2, n - 1$
   do $j = 2, m - 1$
      $a[i, j] = (a[i - 1, j] + a[i, j - 1] + a[i+1, j]$
               $+ a[i, j + 1])/4$
   end do
end do
```

(a) original code : dependences {(1,0), (0,1)}

```
do $i = 2, n-1$
   do $j = i+2, i+m-1$
      $a[i,j-1] = (a[i-1, j-1] + a[i, j-i-1] + a[i+1, j-1]$
               $+ a[i, j-i+1])/4$
   end do
end do
```

(b) skewed code : dependences {(1,1), (0,1)}

Fig. 6. Skewing.

Skewing : Skewing loop I_j by an integer factor f with respect to loop I_i maps iteration

$(p_1, \ldots, p_{i-1}, p_i, p_{i+1}, \ldots, p_{j-1}, p_j, p_{j+1}, \ldots, p_n)$

to

$(p_1, \ldots, p_{i-1}, p_i, p_{i+1}, \ldots, p_{j-1}, p_j + fp_i, p_{j+1}, \ldots, p_n)$

The transformation matrix T that produces skewing is the identity matrix with the element $t_{j,i}$ equal to f rather than zero. Since $i<j$, T must be lower triangular. In the transformed code, the same quantity fp_i is subtracted from every use of the iteration variable of the loop I_j. Note that the computation in the transformed code is still done in the same order as in the original code.

For example, the transformation from Fig.1 (a) to Fig. (b) is a skew of the inner loop with respect to the outer loop by a factor of one, and then subtracts the same quantity from every use of the inner iteration variable inside the loop. The transformed code is equivalent to the original, but the effect on the iteration is to align the diagonal wavefronts of the original loop nest so that for a given value of j, all iterations in i can be executed in parallel.

All these elementary transformation matrices are unimodular matrices.

When transforming loops, the following principle must be satisfied :

Principle 1 *When the transformed code is executed in lexicographic order, all data dependences are satisfied if the transformed distance vectors are lexicographically positive. An unimodular transformation for a loop nest is* legal *iff all the transformed distance vectors are lexicographically positive.*

For example, loop interchange of the loop nest in Example 1 is illegal since the transformed distance vector (-1,1) is illegal. This means that the original dependencies are not satisfied in the transformed loop.

V. Parallelization Techniques

A major goal of parallelizing compilers is to discover and exploit parallelism in loops. This section describes a new model of transformations to maximize loop parallelism [18]. The model is important in the sense that it enables the choice of an optimal transformation without an exhaustive search. The derivation of the optimal compound transformation consists of two steps. The first step puts the loops into a canonical form called *fully permutable loop*, and the second step tails it to specific architecture.

A loop nest is said to be *fully permutable* if all the components of the distance vectors are nonnegative. This implies that any loop permutation would render the transformed dependences lexicographically positive and is thus legal. Full permutability is an important property for parallelization and locality.

We will show that n-deep loops have at least n-1 degrees of parallelism, exploitable at both fine and coarse granularity. The algorithm consists of two steps :

- transformation of the original loop nest into a fully permutable loop nest.
- transformation of the fully permutable loop nest to exploit coarse and/or fine-grain parallelism according to the target architecture.

Loops with distance vectors have a special property that they can always be transformed into a fully permutable loop nest via skewing.

Theorem 1 [18] *Consider a loop nest with lexicographically positive distance vectors. The loops in the loop nest can be made fully permutable by skewing.*

Iterations of a loop can be executed in parallel iff there are no dependences carried by that loop. To maximize the degree of parallelism is to transform the loop nest to maximize the number of DOALL loops.

Theorem 2 Let $(I_1, \ldots, I_n)$ *be a loop nest with lexicographically positive dependences* $\vec{d} \in D$. I_i *is parallelizable iff* for all $\vec{d} \in D$, $(d_1, \ldots, d_{i-1}) > \vec{0}$ or d_i=0.

To determine whether a loop can be executed in parallel, its loop-carried dependences must be examined. The obvious case is when there are no dependences carried across iterations by the loop. We consider several loops which is parallelizable. Figure 7 (a) shows that the outer

```
do i = 1, n
   do j = 2, n
      a[i, j] = a[i, j - 1] + c
   end do
end do
```

(a) Outer loop is parallelizable

```
do i = 1, n
   do j = 2, n
      a[i, j] = a[i - 1, j] + a[i - 1, j + 1]
   end do
end do
```

(b) Inner loop is parallelizable

Fig. 7. Dependence conditions for parallelizing loops.

loop is parallelizable because the distance vector for the loop is (0,1). In Figure 7 (b), the distance vectors are (1,0),(1,-1), so the inner loop is parallelizable by Theorem V.

In general, the loops in the canonical format can be trivially transformed to give the maximum degree of fine grain parallelism. It is shown in [18] that a nest of n fully permutable loops can be transformed to code containing at least n-1 degrees of parallelism.

Theorem 3 [18] *Applying the wavefront transformation to a fully permutable loop nest maximizes the degree of parallelism with the nest.*

$$\begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

For an n-dimensional loop nest, the maximal degree of parallelism is n-1 if there are any dependencies carried within the loop nest, n otherwise.

For example, Fig.6 (a) itself is fully permutable. Applying the wavefront transformation to the original loop make the inner loop in the transformed loop parallelizable. As a matter of fact, this wavefront transformation is equivalent to applying the skewing in Fig.6 (b) and interchange in sequence. The transformed loop is as follows :

```
do $j = 4, m+n-2$
   do $i = max(2, j-m+1), min(n-1, j-2)$
      $a[i, j-1] = (a[i-1, j-1] + a[i, j-i-1] + a[i+1, j$
        $-1] + a[i, j-i+1])/4$
   end do
end do
```

VI. Conclusion

We have given an overall organization and theoretical foundations of parallelizer. Currently, Fortran parallelizer is being developed as a part of the development of the highly parallel computer. It would be the first parallelizer which transforms Fortran programs into HPF programs. After it has been implemented, performance statistics on it will be presented.

References

[1] R Cytron, J Ferrante, and V. Sarkar, "Experiences using control dependence," in *PTRAN*, "Languages and Compilers for Parallel Computing," Edited by D. Gelernter, A. Nicolau, and D. Padua, 1990, MA: MIT Press, pp 186-211.

[2] G.R. Luecke, J Coyle, W. Haque, J. Hoekstra, H. Jespersen, and R. Schmidt, A comparative study of KAP and VAST: two automatic preprocessors with Fortran 8x output, *Supercomputer* 29, vol. V, no. 6, pp 15-25, 1988.

[3] C D. Polychronopoulos, M B Firkar, M R. Hagighat, C. L. Lee, B.P. Leung, and D.A. Schouten, "The structure of Parafrase-2. an advanced parallelizing compiler for C and Fortran," *Languages*

and Compılers for Parallel Computıng, Edıted by D. Gelernter, A. Nıcolau, and D Padua, 1990, MA: MIT Press, pp. 423-453 vol. 81, no. 2, Feb. 1993, pp 288-304

[4] D B Loveman, "Hıgh Performance Fortran," *IEEE Parallel and Dıstrıbuted Technology*, Feb 1993, pp 25-42.

[5] J. Ferrante, K. J. Ottensteın, and J. D Warren, "The program dependence graph and its use in optımızatıon," *ACM Transactıons on Programmıng Languages and Systems*, vol 9, no 3, July 1987, pp. 310-349.

[6] R Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F K Zadeck, "Effıcıently computıng statıc sıngle assıgnment form and the control dependence graph," *ACM Transactıons on Programmıng Languages and Systems*, vol. *13*, no. 4, Oct. 1991, pp. 451-490

[7] F.E.Allen and J.Cocke, "A program dataflow analysis procedure," *Communications of ACM*, vol 19, no. 3, Mar. 1976, pp. 137-146

[8] S. S. Muchnıck and N. Jones, *Program Flow Analysıs*. Englewood Clıffs, New Jersey. Prentıce-Hall, 1981.

[9] J. T. Schwartz and M Sharir, A desıgn for optımızatıons of the bit vectorıng class. Computer Scıence Report no. 17, New York Unıversıty, Sept. 1979.

[10] M. J. Wolfe, "Optımızıng Supercompılers for Supercomputers," Ph D thesis, Unıversıty of Illınois at Urbana-Champaıgn, 1982.

[11] U. Banerjee, "An ıntroductıon to a formal theory of dependence analysıs," *J. Supercomputing*, vol 2, no 2, Oct. 1988, pp 133-149.

[12] D J Kuck, *The structure of Computers and Computatıons*, vol. 1, New York John Wiley and Sons, 1978 .

[13] Y Muraoka, Parallelısm Exposure and Exploıtatıon ın Programs, Ph D. thesıs, Unıversıty of Illınoıs at Urbana-Champaign, *Technıcal Report* 71-424, 1971.

[14] J Lı, P Yew, and C Zhu, "An effıcıent data dependence analysıs for parallelızıng compılers," and M. Chen, "Compılıng communication-effıcıent programs for massıvely parallel machines," *IEEE Trans on Parallel and Dıstrıbuted Systems*, vol. 1, no. 1, Jan. 1990, pp 16-34

[15] X. Kong, D Klappholz, and K Psarıs, "The test. An ımproved dependence test for automatıc parallelızation and vectorızation," *IEEE Trans on Parallel and Distrıbuted Systems*, vol. 2, no 3, July 1991, pp 342-349

[16] W. Pugh, "The Omega test: A fast and practıcal ınteger programmıng algorıthm for dependence analysis," *Proc. of Supercomputıng* '91, Nov. 1991.

[17] D F Bacon, S L Graham, and O J. Sharp, "Compıler Transformatıons for Hıgh-Performance Computıng,"*Technıcal Report* UCB/CSD-93-781, Berkeley. Unıversıty of Calıfornıa, 1993.

[18] M. E. Wolf and M S Lam, "A Loop Transformatıon Theory and an Algorıthm to Maximıze Parallelısm," *IEEE Transactıons on Parallel and Dıstrıbuted Systems*, vol 2, no. 4, Oct. 1991, pp 452-471