ISSN:0128-7680
Pertanika J. Sci. & Technol. 3(2): 361-381 (1995)

Transformation of Sequential Programs into Parallel Forms

Md Yazid Mohd Saman
Department of Computer Science
Faculty of Science and Enviromental Studies
UPM 43400 Serdang, Selangor, Malaysia
yazid@cs.upm.my

Received 24 November 1993

ABSTRAK

Salah satu tugas yang perlu dilakukan oleh pengaturcara ketika menulis aturcara selari ialah mengenal pasti bahagian yang akan dilaksanakan secara selari. Proses ini memakan masa dan selalu memberikan kesalahan. Satu cara lain ialah pengaturcara menulis aturcara jujukan terlebih dahulu dan ini akan diterjemahkan ke dalam bentuk selari oleh pengkompil selari. Lingkaran dalam aturcara merupakan bahagian yang mudah untuk diterjemahkan kepada bentuk selari. Kertaskerja ini membincangkan teknik penterjemahan yang boleh dilaksanakan ke atas aturcara jujukan terutama lingkaran, untuk dilaksanakan secara selari. Teknik ini berdasarkan set Bernstein.

ABSTRACT

One of the main tasks of a programmer when writing parallel programs is to identify the parts that are to be executed in parallel. This process is very time-consuming and error prone. As an alternative, one can write its sequential version and then transform it into the parallel form by a parallelizing compiler. The loops in the sequential programs offer the best opportunities for parallelism. This paper presents the transformation techniques that can be applied to sequential programs, especially the loops, in order to parallelize them. These techniques are based on the Bernstein sets.

Keywords: sequential programs, parallelism, transformation, Bernstein sets

INTRODUCTION

Programming for parallel computers is not as easy as writing the equivalent sequential programs. One of the programmer's main tasks is to identify parts of the programs that are to be executed in parallel. One way to develop a parallel program is to write its sequential version in the initial stage. This program is then transformed into its parallel form by a sophisticated software tool such as the parallelizing compiler. Some researchers have developed complete parallel programming environments such as the Parafrase (Polychronopoulos 1988) and KAP (Kuck *et al.* 1984).

The parts of a sequential program which offer the best opportunities for parallelism are the loops (Allen 1988; Banerjee 1988; Wolfe 1989; Md Yazid 1993a). The iterations in a loop can be parallelized if all of them are independent, i.e. there is no data dependence between them. Inter-iteration data dependences are usually caused by references by an iteration to array elements modified by others. In the case of scalar variables in the loops, data dependences occur if they are involved in store operations in the different iterations.

To detect any data dependences in a loop, the statements in its body have to be analysed in the data dependence analysis (Md Yazid 1993a). In order to execute the iterations with data dependences concurrently, the loops have to be modified to eliminate those dependences. In cases where data dependences cannot be removed, synchronization statements are inserted in the loops. This paper proposes techniques that can be applied to sequential programs in order to transform and execute them in parallel. These techniques are based on the Bernstein method consisting of the Bernstein sets (BSs), the Bernstein tests (BTs) and the Bernstein loop tests (BLTs) (Bernstein 1966; Williams 1978; Md Yazid 1993a). This paper shows how information provided by the BSs and the results of the BLTs can be used in making the decision for the transformation process.

PARALLELIZATION OF PROGRAMS

Loop transformation has been a major focus in the parallelization of sequential programs and before any transformation can be performed, extensive analysis has to be carried out to determine the data dependence between the iterations. One common method to perform this analysis is to develop the data dependence graph (DDG). The DDG is then used to decide on which transformation techniques are to be applied. Most researchers use this technique (Kuck *et al.* 1984; Padua and Wolfe 1986; Wolfe 1989). The other method bases the analysis on the Bernstein sets (Bernstein 1966; Md Yazid 1993a, 1993b). In the parallelization of sequential programs, apart from modifying the codes to eliminate data dependences, insertion of proper synchronization constructs may also be carried out so that the iterations of the loops can be executed in parallel. There are two types of loop transformations: parallelization and vectorization (Padua and Wolfe 1986; Polychronopoulos 1988; Zima and Chapman 1990). Vectorization is a process of transforming loops into vector codes for vector computers. On the other hand, parallelization is a transformation process mainly targeted for shared-memory parallel machines. In this paper, the BSs and the BLTs become the bases for the program transformations.

DEFINITIONS OF FETCH AND STORE DIRECTIONS

The BLTs are tests to determine parallelism in loops; their application will produce results that can be classified as the forward/store (FS), forward/fetch (FF), backward/store (BS) or backward/fetch (BF). These definitions will be used to decide on how the loops can be transformed. The basic definitions of BSs and the BLTs are given in the appendix and their detailed discussions can be found alsewhere (Bernstein 1966; Williams 1978; Md Yazid 1993a).

Results of the Test $(XYZi \cap XYZj)$

This test detects any dependences caused by simultaneous store operations, so the letter X will be appended to each classification.

(i) X-forward/store (XFS) - if the forward dependence involves a store as for the array a below:

```
a[i] := p;          OR          a[i+1] := q;
a[i+1] := q;                    a[i] := p;
```

(ii) X-backward/store (XBS) - if the backward dependence involves a store as for the array a below:

```
a[i-1] := q;        OR          a[i] := q;
a[i] := p;                      a[i-1] := p;
```

Results of the Test $(XYZi \cap XYZj)$

This test detects any data dependences due to fetch first and store later operations (that is Y) or dependences due to store first and fetch later operations (that is Z). Hence a letter Y or Z will be appended to each classification.

(i) Y and Z-forward/store (YFS and ZFS) if the forward dependence involves a store as for the array a below. The values fetched are new values.

```
YFS                     ZFS
p := a[i];              a[i+1] := p;
a[i+1] := q;            q := a[i];
```

(ii) Y and Z-forward/fetch (YFF and ZFF) if the forward dependence involves a fetch as for the array a below. The values fetched are old values.

```
YFF                     ZFF
q := a[i+1];            a[i] := p;
a[i] := p;              q := a[i+1];
```

(iii) Y and Z-backward/store (YBS and ZBS) if the backward dependence involves a store as for the array a below. The values fetched are old values.

```
YBS                     ZBS
p := a[i];              a[i-1] := p;
a[i-1] := p;            q := a[i];
```

(iv) Y and Z-backward/fetch (YBF and ZBF) if the backward dependence involves a fetch as for the array a below. The values fetched are new values.

```
YBF                     ZBF
q := a[i-1];            a[i] := p;
a[i] := p;              q := a[i-1];
```

BS-BASED TRANSFORMATIONS

In this section, transformation methods based on the contents of the BSs are discussed. Consider the loop and its corresponding BSs shown in *Fig. 1.* The variable c is involved in store and fetch operations and this causes data dependence between iterations. Undefined results may be stored or fetched if the loop is parallelized. *Fig. 1(b)* shows that the variable c is a member of Z set. If the iterations are to be executed in parallel, care must be taken to ensure that the value of c is properly maintained.

```
for i:=1 to n do
begin
        ...;
        c := a[i] + a[i+1] + a[i+2];
        if c > m[i] then m[i] := c;
        ...;
end;
```

(a) a loop with scalar data dependences

W	X	Y	Z
a[=] a[<1] a[<2]	-	m[=]	c

(b) Bernstein sets.

Fig. 1. Scalar variables data dependences

Based on the BS types, the following transformation decisions can be made. Note that the BSs are derived from the whole loop body.

Bernstein set	Transformation decisions
(i) Variables in the W set	Scalar variables in this set do not have any effect since they involve only fetch operations, so each iteration can be executed in parallel.
(ii) Variables in the X set	Scalar variables in this set involve only store operations; this could give undefined values by the concurrent execution of the iterations in the loop. Since their values are not fetched by any iteration, they become redundant and may be moved out of the loop. However, their values in the last iteration have to be computed and saved for later references. Alternatively, they can be renamed with array variables or declared as local variables.
(iii) Variables in the Y set	The values of the variables in this set are first fetched and later stored, so the data dependences can be eliminated by scalar renaming (discussed below) or they are declared as local variables for each iteration.
(iv) Variables in the Z set	The values of these variables are stored and later fetched, thus they can be treated as local variables in each iteration or eliminated by scalar renaming or scalar forward substitution.

Scalar Forward Substitution
In this technique, each occurrence of a variable is substituted with its corresponding expression (Aho *et al.* 1986; Polychronopoulos 1988; Zima and Chapman 1990). In the following example:

```
for i       := 1 to n do
begin
            ...
S1:         x := i*i - k;
            ...
S2:         sum[x] := sum[x,x+5] + a[i];
            ...
end
```

the variable x causes the data dependence. Since its value is first evaluated, its occurrence in the succeeding statements can be replaced by the expression evaluated earlier. This eliminates the data dependence. Hence, statement S1 can be eliminated and statement S2 modified to:

```
sum[i*i - k] := sum[i*i - k,i*i - k+5] + a[i];
```

Since the variables involved must be initially stored and later fetched, they must be members of the Z set. If the variables involve a new store operation, then care has to be taken that the new value substituted is the latest assigned expression. A main disadvantage of this technique is that it increases the run-time needed to repeatedly evaluate each expression.

Scalar Renaming (Expansion)
In this technique, the scalar variable is given a different name to eliminate any data dependences (Aho *et al.* 1986; Polychronopoulos 1988; Zima and Chapman 1990). This is usually done by renaming it with a temporary array variable. For example, consider the following loop.

```
for i:=1 to n do
begin
            ...
            a := 10;
            x := a + b;
            ...
end;
```

The variables a and x which cause the data dependences can be renamed with array names such as NEWa[i] and NEWx[i] respectively.

The transformed loop, which does not have any dependences, is as follows .

```
forall i:=1 to n do
begin
        ...
        NEWa[i] := 10;
        NEWx[i] := NEWa[i] + b;
        ...
end;
```

This technique applies to scalar variables in the X, Y and Z sets of the BSs. However, if the variables in Y set are involved in a statement such as a := a + 1 (i.e., a is called a reduction variable), then they cannot be renamed as in the above method since each iteration uses and updates the value. One major shortcoming of this technique is that it increases the use of variables in the program.

Constant Propagation

This is a common technique in optimizing compilers which determines the values of constants in programs. These values are then propagated throughout the programs (Aho *et al.* 1986; Polychronopoulos 1988; Zima and Chapman 1990). This technique eliminates the need for run-time evaluation of the values. Consider the following example.

```
for i:=1 to n do
begin
    S1:pi := 3.142;
    S2:twopi := 2*pi;
    S3:arr[i] := twopi * rad[i] * rad[i];
end;
```

The variables pi and twopi can be determined as constants since they are assigned constant values. Consider the BSs of the loop body.

	W	X	Y	Z
S1:	-	pi	-	-
S2:	pi	twopi	-	-
S3:	twopi rad[i]	arr[i]	-	-

From the first BSs, pi is assigned a constant value since there are no other variables in the other sets. This value is then fetched and later stored in twopi, thus showing that twopi is also a constant. Therefore, the loop can be transformed into vector instructions as follows.

```
pi := 3.142;
twopi := 6.284;
arr[1:n] := twopi * rad[1:n] * rad[1:n];
```

BLT-BASED TRANSFORMATION OF LOOPS

In this section, some transformation methods suitable for loops containing array variables are discussed. These methods are standard techniques for program transformation and are widely discussed in the literature (Polychronopoulos 1988; Wolfe 1989; Zima and Chapman 1990; Lewis and El-Rewini 1992). However, the rules for transformation are based on the contents of the BSs and the results of BLTs (defined earlier as XFS, XBS, YFS, ZFS, YFF, ZFF, YBS, ZBS, YBF and ZBF).

Loop distribution

In this technique, a loop is broken into several loops to distribute the control over groups of statements in its body. This is particularly convenient for vectorization. As an example, the single loop below has a YFF dependence caused by array a.

```
for i:=1 to n do
begin
S1:   c[i] := a[i+2] * b[i];
S2:   a[i] := b[i] + c[i];
end
```

It can be transformed to become two loops as follows.

```
for i:=1 to n do
      c[i] := a[i+2] * b[i];

for i:=1 to n do
      a[i] := b[i] + c[i];
```

Then they can be vectorized to become:

```
c(1:n) = a(1:n+2) * b(1:n)
a(1:n) = b(1:n) + c(1:n)
```

The semantics of the statements are preserved since the fetched element of a[i+2] in S1 contains its old value. In the following example, the array a has a XFS data dependence.

```
for i:=1 to n do
begin
        a[i] := p;
        a[i+1] := q;
end;
```

It can also be distributed into two loops to become:

```
for i:=1 to n do
        a[i] := p;

for i:=1 to n do
        a[i+1] := q;
```

The vectorized statements are as follows.

```
a[1:n] := p;
a[2:n+1] := q;
```

In general, the BLT-based rules for this transformation are as follows.

a. The variables are store dependent This usually appears in initialization loops.
b. The statements do not have forward and backward dependences (i.e., in a cycle) such as in the following example. Statements S1 and S2 contain an array a that has a forward dependence and array b that has a backward dependence. This means that they cannot be separated into different loops.

```
for i:=1 to n do
begin
S1:     a[i+1] := b[i-1] + c[i];
S2:     b[i] := a[i] * d[i];
S3:     c[i] := b[i] + d[i];
end;
```

However, the statement S3 has an equal dependence with S1 and S2 and hence the whole loop can be distributed as follows.

```
for i:=1 to n do
begin
S1:    a[i+1] := b[i-1] + c[i];
S2:    b[i] := a[i] * d[i];
end;

for i:=1 to n do
S3:    c[i] := b[i] + d[i];
```

c. The variables are not YBF or YFS because loop distribution destroys the dependences and the new values of the arrays are not properly accessed.

Statement reordering

This method involves exchanging the textual positions of two statements in a loop body. The following loop cannot be vectorized due to the presence of a data dependence on array a which is YBF. The values fetched are new values that are assigned by the other statement, except for the first element.

```
for i:=1 to n do
begin
        c[i] := a[i-1] - 4;
        a[i] := b[i] * 2;
end;
```

If the statements are reordered, the loop becomes:

```
for i:=1 to n do
begin
        a[i] := b[i] * 2;
        c[i] := a[i-1] - 4;
end
```

Now the data dependence of array a has a ZBF dependence where the fetched values of a[i-1] are new values. Thus, the loop can be distributed and vectorized.

```
a[1:n] = b[1:n] * 2
c[1:n] = a[0:n-1] - 4
```

In general, the conditions for this kind of transformation, based on the results of BLTs, are as follows.

a. For the scalar variables, they are members of the W or X sets only and not members of the Y and Z sets.

b. For the array variables, the types of dependence are YFS and YBF. These dependences involve new values that are being fetched initially. Hence, the statement with the fetch operation may be reordered to appear later in the sequence of iteration execution.

c. For the array variables, there are NO equal (=) reference directions after the BLTs have been applied, i.e., there are no loop-independent dependences on any variables such as the array a in the following example. Note that array b has a YFS dependence.

```
for i:= 1 to n do
begin
        a[i] := b[i];
        b[i+1] := a[i] + ...
end
```

Loop Interchange

In this technique, any two levels of a perfectly nested loop are exchanged. For the example below, the BLTs indicate that the inner loop is unparallelizable, since there is a forward dependence for the j loop for array a[=,<]. If the loop statements are interchanged, the new inner loop can then be parallelized.

```
for i:=1 to n do
begin
        for j:=1 to n do
        a[i,j+1] := a[i,j] * b[i,j];
end
```

The interchanged version is as follow.

```
for j:=1 to n do
                forall i:=1 to n do
                a[i,j+1] := a[i,j] * b[i,j];
```

After interchanging, the outer j loop will be executed sequentially while the inner i loop can be executed concurrently. This is suitable for vectorization as in the following form.

```
for j:=1 to n do
        a[i:n,j+1] := a[i:n,j] * b[i:n,j];
```

However, for parallelization, the earlier version (before interchanging) is preferable since then there are overheads incurred in executing the

outer loop concurrently. For a nested loop such as in the above example, this will involve more than one direction for the array variables, that is, a vector of reference directions is needed. Let D be a vector of data reference directions (DRDs):

D = (d1,d2,...,dn) with di = (<,>,=,*),
for all $1 \leq i \leq n$,n= number of dimensions

The necessary condition for this kind of transformation is that, given a nested loop of n dimensions, two loop statements with array directions A[...,di,...,dj,...] cannot be interchanged if one of them (i.e., di or dj) has a forward direction and the other one has a backward direction. This means that if an array has directions such as A[...,<,...,>,...], it indicates that the two loops are not interchangeable. The forward, backward and equal reference directions are derived by the BLTs.

Loop Unrolling
This technique makes one or more copies of the loop body and thus increases the stride. This reduces the control overhead in executing the loops as in the following example.

```
for i := 1 to 1000 do
        a[i] := b[i+2] * a[i-1];
```

It can be unrolled to become a loop with a stride of 2 which has only 500 (i.e., 50% fewer) iterations to be generated.

```
for i := 1 to 1000 step 2 do
begin
      a[i] := b[i+2] * a[i-1];
      a[i+1] := b[i+3] * a[i];
end
```

A similar technique called loop replication makes copies of statements in loop body without changing the stride (Allen *et al.* 1987). Consider the following example.

```
for i := 1 to n do
begin
      a[i] := b[i] * c[i];
      d[i] := a[i] * a[i-1];
end
```

It has a ZBF dependence caused by operations on array elements a[i] and a[i-1]. To be able to perform array alignment, as discussed above, the first statement can be replicated and the array element a[i] in the second and third statement renamed to NEWa[i].

```
for i := 1 to n do
begin
        a[i] := b[i] * c[i];
        NEWa[i] := b[i] * c[i];
        d[i] := NEWa[i] * a[i-1];
end
```

The loop can then be distributed and aligned as follows, thus eliminating all data dependences.

```
for i := 1 to n do
        NEWa[i] := b[i] * c[i];
forall i := 0 to n do
begin
        if (i > 0) then a[i] := b[i] * c[i];
        if (i < n) then d[i+1] := NEWa[i+1] * a[i];
end
```

Array Renaming (Variable Copying)

Scalar renaming is useful for eliminating data dependences involving scalar variables. Array variables, can also be renamed to eliminate data dependences. Consider the following example where there is a data dependence on array a which is ZFF.

```
for i:=1 to n do
begin
        a[i] := b[i] + c[i];
        d[i] := a[i] + a[i+1];
end;
```

It can be transformed into the following version after renaming the array a[i] to NEWa[i], thus eliminating the data dependence.

```
for i:=1 to n do
        NEWa[i] := a[i];
forall i:=1 to n do
begin
        a[i] := b[i] + c[i];
        d[i] := a[i] + NEWa[i+1];
end;
```

The loops can then be vectorized as follows.

```
NEWa[1:n] := a[1:n];
a[1:n] := b[1:n] + c[1:n];
d[1:n] := a[1:n] + NEWa[2:n+1];
```

For this technique, the array variables must be of the types YFF, ZFF, YBS or ZBS as derived by the BLTs, to enable them to be renamed with different array names. This is because the forward reference is a reference to an old value. In the previous example, a[i+1] refer to old values of a and thus can be renamed.

Other Transformation Techniques

(i) Index set splitting

In this technique, the loop is divided into two or more loops with partial size. The loops can then be executed concurrently. This technique requires an extraction by the BLTs of the distance of the dependence in order for the loop bound to be split properly.

(ii) Node splitting

This method breaks expressions occurring in statements into several parts. This involves a lower level treatment of expressions in statements of the loops. The arrays involved must be those which do not contribute to any results in the BLTs.

(iii) Loop blocking.

If the BLTs discover data dependences in a loop with dependence distances $\geq$ 2 then loop blocking transformation can be used to parallelize it (Padua and Wolfe 1986; Polychronopoulos 1988). It creates doubly nested loops out of a single loop, by organizing the computation in the original loop into chunks of approximately equal size.

(iv) Array alignment

Array alignment is a technique which involves adjusting the array reference to eliminate the data dependences. It transforms a loop-carried dependence into a loop-independent dependence. Variables involved are usually of the types YBS and ZBS such as a[i-1] which can be aligned to a[i]. This is allowed since the backward reference is a reference to an old value.

Insertion of Synchronization Statements

In some cases, data dependences cannot be eliminated at all. When array variables with complex array subscripts, such as coupled subscripts or array subscripts or those other than (i1const), are met, usually data dependence is assumed to exist. Since the dependences are difficult to eliminate, the synchronization instructions such as **LOCK** and **UNLOCK** (for shared memory machines) are used (Midkiff and Padua 1987). These instructions create the critical regions in which only one process is able to execute its critical region at a time.

PROBLEMS AND FUTURE WORK

Table 1 gives a summary of the transformation techniques that can be performed, based on the results of the BLTs. The techniques discussed above have some limitations. In this section these problems and future work will be discussed.

Problems

Regarding the formation of data reference directions (DRDs) in the BLTs, the subscripts that are analysed are simple expressions of the forms [$i \pm$ constant]. Shen *et al.* (1989) have shown that there are other forms of complex subscript expressions commonly found in programs, although they are not found as frequently as the simple expressions allowed by BLTs. These complex expressions include coupled subscripts (i.e., loop indices appearing at any level), nonlinear subscripts, array subscripts and symbolic subscripts. In this case, the BLTs will assume that there are data dependences and no transformation can be carried out.

The handling of complex subscript expressions has been studied by several researchers using numerical methods (Banerjee 1988; Wolfe 1989). Solutions for array subscripts have been discussed by Polychronopoulos (1988). Apart from these problems, loops sometimes contain non-uniform loop indexing, procedure calls and conditional statements. This increases the complexity of the loop analysis for data dependence. Procedure calls can be handled by inter-procedural analysis (IPA) (Md Yazid 1993c). The problem caused by non-uniform loop indexing needs modification before any analysis and transformation can be performed.

An example of such a loop is as follows.

```
for i := 100 downto 1 do
for j := 5 to 100 step 3 do
```

This can be overcome by normalizing the loop indexing so that each loop will start from 1 with a stride of 1. This, however, will complicate the subscript expressions in the loop body and thus making it more

TABLE 1
Loop transformations and the dependence types

Dependence Types	Transformation
X-Forward/Store (XFS)	Loop distribution
X-Backward/Store (XBS)	Loop distribution
Y-Forward/Store (YFS)	Statement reordering
	Loop blocking
Y-Forward/Fetch (YFF)	Loop distribution
	Array renaming
	Loop blocking
Y-Backward/Store (YBS)	Loop distribution
	Array renaming
	Array alignment
Y-Backward/Fetch (YBF)	Statement reordering
Z-Forward/Store (ZFS)	Loop distribution
Z-Forward/Fetch (ZFF)	Statement reordering
	Array renaming
	Loop blocking
Z-Backward/Store (ZBS)	Statement reordering
	Array renaming
	Array alignment
Z-Backward/Fetch (ZBF)	Loop distribution

difficult to be handled by the BLTs. Another problem that is encountered by the dependence analysis is symbolic subscripts where the subscripts contain variables (Haghighat 1990). Sometimes, this can be solved by constant propagation. However, in some cases, the actual values of the symbolic expressions are only known at run-time. One simple solution is to generate conditional vectorized statements (Padua and Wolfe 1986). Consider the following example.

```
for i:=1 to n do
      a[i+k] := a[i]/b[i] + c[i];
```

If the value of k is not known at compile time, the translation could look like the following code.

```
if (k < 1 or k>= n) then
          a[k+1:n+k] := a[1:n] / b[1:n] + c[1:n]
else
```

```
for i:=1 to n do
        a[i+k] := a[i]/b[i] + c[i];
```

The presence of complex control flow in loops also poses problems for the BLTs. This creates the control dependence between two or more statements in a loop. This dependence prevents the execution of one statement while executing the other (Allen *et al.* 1983). One simple solution to this problem is to convert them into data dependences. Logical variables are introduced to control execution of statements. Consider the following example.

```
for i := 1 to n do
    if a[i] > 0 then
        a[i] := b[i] + c[i];
```

The control dependence can be removed as follows.

```
for i := 1 to n do
begin
    t := a[i] > 0;
    if (t) then a[i] := b[i] + c[i];
end
```

The vectorized form of the above loop makes use of the where statement, as in Fortran 8x; it is as follows.

```
t[1:n] = a[1:n] > 0;
where (t[1:n]) a[1:n] = b[1:n] + c[1:n];
```

Implementation

Currently, a project is being carried out to implement the above transformation techniques. A software tool called UNIPAR is being developed that will accept a subset of a sequential Pascal program and transform it into parallel Pascal such as Sequent Pascal and Multi-Pascal (Lester 1993). Details of its background and implementation are discussed elsewhere (Md Yazid 1995).

CONCLUSION

Once the data dependences in loops have been ascertained in the data dependence analysis, they can be parallelized by modifying the codes to remove the dependences. This paper has discussed the transformation techniques that can be carried out on loops in sequential programs in order to parallelize them. Combinations of the techniques may be used to do the transformations. This process must ensure that the semantics of the loops are maintained. For those loops whose dependences cannot be eliminated,

certain synchronization constructs such as LOCK and UNLOCK may be inserted and the loop iterations may still be run in parallel.

The rules for transformation developed in this paper are based on the BLTs as well as the contents of the BSs. Other researchers have been concentrating on performing the transformation based on the data dependence graphs. Although the algorithms for the DDGs are well known, they need extra space for their representation. The BLTs-based techniques can be applied directly from the results of the BLTs. As a conclusion, this paper has shown that the BSs and the results of the BLTs are very useful in making decisions on the type of transformation methods to be carried out in the parallelization of programs.

ACKNOWLEDGEMENTS

The supervision of Prof. D.J. Evans of Parallel Algorithms Research Centre, UK in conducting the above research is greatly appreciated. Financial assistance was provided by Universiti Pertanian Malaysia.

REFERENCES

AHO, A.V., R. SEITH and J.D. ULLMAN. 1986. *Compilers Principles, Techniques and Tools.* Reading, MA: Addison-Wesley.

ALLEN, F.E. 1988. Compiling for parallelism: an overview. In: *Parallel Systems and Computations* ed. G. Paul and G.S. Almasi. p. 3-13. North-Holland.

ALLEN, J.R., K. KENNEDY, C. PORTERFIELD and J. WARREN. 1983. Conversion of control dependence to data dependence. In: *Proceedings of 10th ACM Symposium on Principles of Prog. Lang*, p. 177-189. Austin, Texas: ACM Press.

ALLEN, R., D. CALLAHAN and K. KENNEDY. 1987. Automatic decomposition of scientific programs for parallel execution. In: *Proceedings of 14th ACM Symposium on Principles of Prog. Lang*, p. 63- 76. Austin, Texas: ACM Press.

BANERJEE, U. 1988. *Dependence Analysis for Supercomputing.* Boston, Mass: Kluwer.

BERNSTEIN, A.J. 1966. Analysis of programs for parallel processing. *IEE Trans. on Elec. Comp.* **1 EC-15**: 757-763.

HAGHIGHAT, M.R. 1990. Symbolic dependence analysis for high performance parallelizing compilers. CSRD Rpt. No. 995 MSc thesis, University of Illinois USA.

KUCK, D.J., R.H. KUHN, B. LEASURE and M. WOLFE. 1984. The structure of an advanced retargetable vectorizer. In: *Tutorial Supercomputers: Design and Applications*, ed. K. Hwang. p. 163-178. IEEE Computer Soc.

LESTER, B.P. 1993. *The Art of Parallel Programming.* Englewood Cliffs, NJ: Prentice-Hall.

LEWIS, T.G. and H. EL-REWINI. 1992. *Introduction to Parallel Computing.* Englewood Cliffs, NJ: Prentice Hall.

MIDKIFF, S.P. and D.A. PADUA. 1987. Compiler algorithms for synchronization. *IEEE Trans. on Computers* **C-36-12**: 1485-1495.

MD YAZID MOHD SAMAN. 1993a. The Bernstein method for data dependence analysis. Technical Report SAK/TR-003/1993, Dept. of Computer Science, UPM.

MD YAZID MOHD SAMAN. 1993b. The Bernstein method for automatic parallelization of programs. Technical Report SAK/TR-001/1993, Dept. of Computer Science, UPM.

MD YAZID MOHD SAMAN. 1993c. Inter-procedural analysis. Technical report SAK/TR-002/1993, Dept. of Computer Science, UPM.

MD YAZID MOHD SAMAN. 1995. UNIPAR: A software tool for parallelizing Pascal programs. Technical report SAK/TR-004/1995, Dept. of Computer Science, UPM.

PADUA, D.A. and M.J. WOLFE. 1986. Advanced compiler optimizations for supercomputers. CACM **29(12)**: 1184-1201.

POLYCHRONOPOULOS, C.D. 1988. *Parallel Programming and Compilers.* Boston, Mass: Kluwer.

SHEN, Z., Z. LI and P. YEW. 1989. An empirical study on array subscripts and data dependencies. In: *Proceedings of International Conference on Parallel Processing,* p. II-145-152.

WILLIAMS, S.A. 1978. Approaches to the determination for parallelism for computer programs. PhD thesis, Loughborough University of Technology, UK.

WOLFE, M. 1989. *Optimizing Supercompilers for Supercomputers.* London: Pitman.

ZIMA, H.P. and C. CHAPMAN. 1990. *Supercompilers for Parallel and Vector Computers.* ACM Press.

APPENDIX

DEFINITION 1.

Bernstein sets (BSs) consist of four sets defined as follows:

a. W set - set of variables fetched during execution of task
b. X set - set of variables stored during execution of task
c. Y set - set of variables which involves a fetch and one of the succeeding operations is a store
d. Z set - set of variables which involves a store and one of the succeeding operations is a fetch

DEFINITION 2.
Bernstein tests (BTs) between two tasks i and j, are tests to determine whether they can be run concurrently or not, i.e., if they satisfy all of the following three conditions:

(Xi OR Yi OR Zi) AND (Wj OR Yj OR Zj)= $\underline{\emptyset}$
(Wi OR Yi OR Zi) AND (Xj OR Yj OR Zj)= $\underline{\emptyset}$
(Xi OR Yi OR Zi) AND (Xj OR Yj OR Zj)= $\underline{\emptyset}$

DEFINITION 3.
Bernstein loop tests (BLTs) are tests to determine whether loop iterations can be run concurrently or not. In these tests data reference directions (forward direction <, backward direction > and equal direction =) are introduced in the Bernstein Sets. The loop must satisfy all of the following three conditions in order to be parallelised.

BLT1:WYZi AND XYZj
where 1 <= i,j <= n, n = number of tasks
BLT2:XYZi AND XYZj
where 1 <= i,j <= n, n = number of tasks

Note that XYZi and WYZi denote (Xi OR Yi OR Zi) and (Wi OR Yi OR Zi) respectively. To determine the existence of data dependences in a loop that contains n tasks Si and Sj (where 1 <= i,j <= n):

a. for scalar variables:
BLT1 (Si , Sj) <> Ø for all i and j
BLT2 (Si , Sj) <> Ø for all i and j, i < j

b. for array variables:

BLT1 (Si,Sj) <> Ø or produces forward/backward directions with non-zero dependence distances, for all i and j and BLT2 (Si,Sj) <>Ø or produces forward/backward directions with non-zero dependence distances, for all i and j and i < j.