AN APPROACH OF PARALLEL COMPUTATION ON FACTORING PROGRAMS

by

JIEH-SHWU LIN

B.S., Chinese Cultural College(Taiwan, R.O.C), 1978

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements of the degree

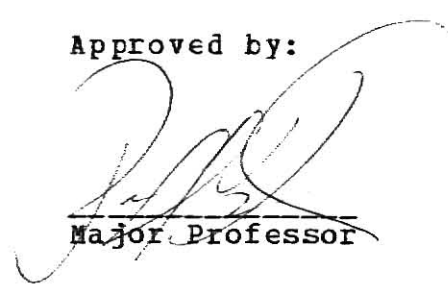
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1982

Approved by:



Major Professor

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my major advisor, Dr. Paul S. Fisher, for innumberable ways he provided ideas, assistance, patience, and consideration throughout my research project.

Thanks are due also to the committee members, Dr. David Gustafson, and Dr. Rodney Bates. I appreciate the time they took out of their busy schedules to serve on this committee.

A special thank-you is given to Dr. James Case for his interest and enthusiasm in providing many suggestions on this work.

Finally, specially gratitude is given to my mother, father, and my entire family for their love and support to make possible the completion of this report.

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

A program is a collection of computations related to each other with the needed of data values that are generated and consumed. Obviously, the execution order of the computations is not simply stated by the program but rather by the data dependencies, i.e., both static and dynamic structures are vital factors which effect the program executions, and are even more important in the parallel computations.

The main reason for today's parallelism is obtained by the follcwing motives(21):

1. Increase the speed of computation beyond the limit imposed by technological limitations.
2. Reduction of turnaround time of jobs.
3. Reduction of memory and time requirements for hoursekeeping chores.
4. An increase in simultaneous service to many users.
5. Improved performance in a uniprocessor multiprogrammed environment.

Within an individual program, parallelism can exist at several levels, i.e., the independent programs can be processed concurrently which is synonymous with the

conventional meaning of multiprocessing in the multiprogrammed environment. We are considered to intra-program parallelism as opposed to the inter-program parallelism. Intra-program parallelism refers to the type of processing in which a single program can be partitioned into tasks that can be performed in parallel.

On the other hand, natural sequences seem to have a regularity which is neither statistical nor analytic, but combinatorical. In Chapter III, a new kind of combinatorical regularity of natural sequences of symbols, called factoring, presented by Case & Fisher(7), is introduced.

In this paper, we will first introduce some parallel execution algorithms (Chapter II), followed by a factoring technique, and then give some examples to show how the program, after properly been factored, can be executed in parallel.

CHAPTER II

PARALLEL COMPUTATION

2.1 INTRODUCTION

A sequential program has the statements coded serially as if they are to be executed by a single processor one at time. However, in this section, some schemes for parallel executions of sequential programs will be described.

In a sequential program, statements can be considered as only the following two catagories:

(i) Assignment Statements --

An assignment statement has the form of "X <-- Expr", where X is a variable defined by the arithmetic or logical expression named Expr.

(ii) Branch Statements --

A branch statement is defined by either "IF Cond THEN Statements" or "GO TO Statements", where Cond is a conditional expression.

Therefore, any loop statements in a dynamic structured program, can be written as simple assignments and branch statements by iteration.

2.2 BENSTEIN'S SCHEME FOR PARALLEL COMPUTATICN

There are some schemes for the parallel execution of a sequential program which require a procedure to detect parallelism in a program before its execution. Here, we like to introduce Benstein's Algorithm.

Benstein(6,22) presented a model for FORTRAN language and a set of conditions to determine whether or not two successive portions of a given program can be executed in parallel and still produce the same results. He considers two machine models, one in which every processor communicates directly with a common main memory and one in which each processcr has a slave memory to take care of the information coming from or going to main memory. The variables are developed into four categories:

1) The location is only fetched during execution.
2) The location is only stored during execution.
3) The location which is first fetched and then stored.
4) The location which is first stored and then fetched.

(These are denoted as Wi, Xi, Yi, Zi sets respectively.)

If the data on successive three portions of a program can be satisfied by the relations:

(1) $(W1 \cup Y1 \cup Z1) \cap (X2 \cup Y2 \cup Z2) = \phi$

(2) $(X1 \cup Y1 \cup Z1) \cap (W2 \cup Y2 \cup Z2) = \phi$

(3) $X1 \cap X2 \cap (W3 \cup Y3) = \phi$

then their relationships are independent, so the first two portions can be executed in parallel.

However, these schemes have two restrictions: (i) the parallel executable statements share limited variable space (ii) the processors executing them do not communicate with each other. Ramamoorthy & Leung represented a scheme which relaxes the restrictions.

2.3 RAMAMCORTHY & LEUNGS' SCHEME FOR PARALLEL EXECUTION

They(22) proposed a scheme having monitors which preserve the inherent precedence relation among statements. There are some static information about the use of variable which will be generated before execution to aid the monitoring procedure. Also, the monitoring process requires dynamic information to trace the execution order of statements.

The precedence among statements which must be preserved is related to both the execution order and data dependency. They presented a monitoring process which is aided by two pieces of information: (i) the reference table which indicates how a variable is used in the program statements, and (ii) a stack of trace vectors which keeps track of the execution order of statements.

Given an example assignment statement block as following:

```
S1: A(I,J) = 1.0
S2: X = X + I
S3: I = I - 1
S4: J = J - 1
S5: Y(I) = I + X
S6: Z(I,J) = I * J
```

every element in the table is denoted as RT(X,Si), where X is a variable name and Si is a statement label, such as

RT(X,Si) = 00, if X is not referenced in Si.

RT(X,Si) = 01, if X is read in Si.

RT(X,Si) = 10, if X is updated in Si.

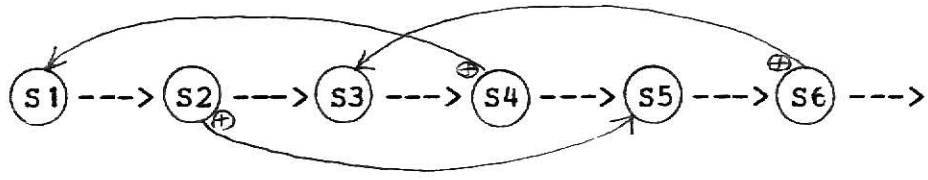
RT(X,Si) = 11, if X is read and updated in Si.

the reference table for the above program block then can be shown as:

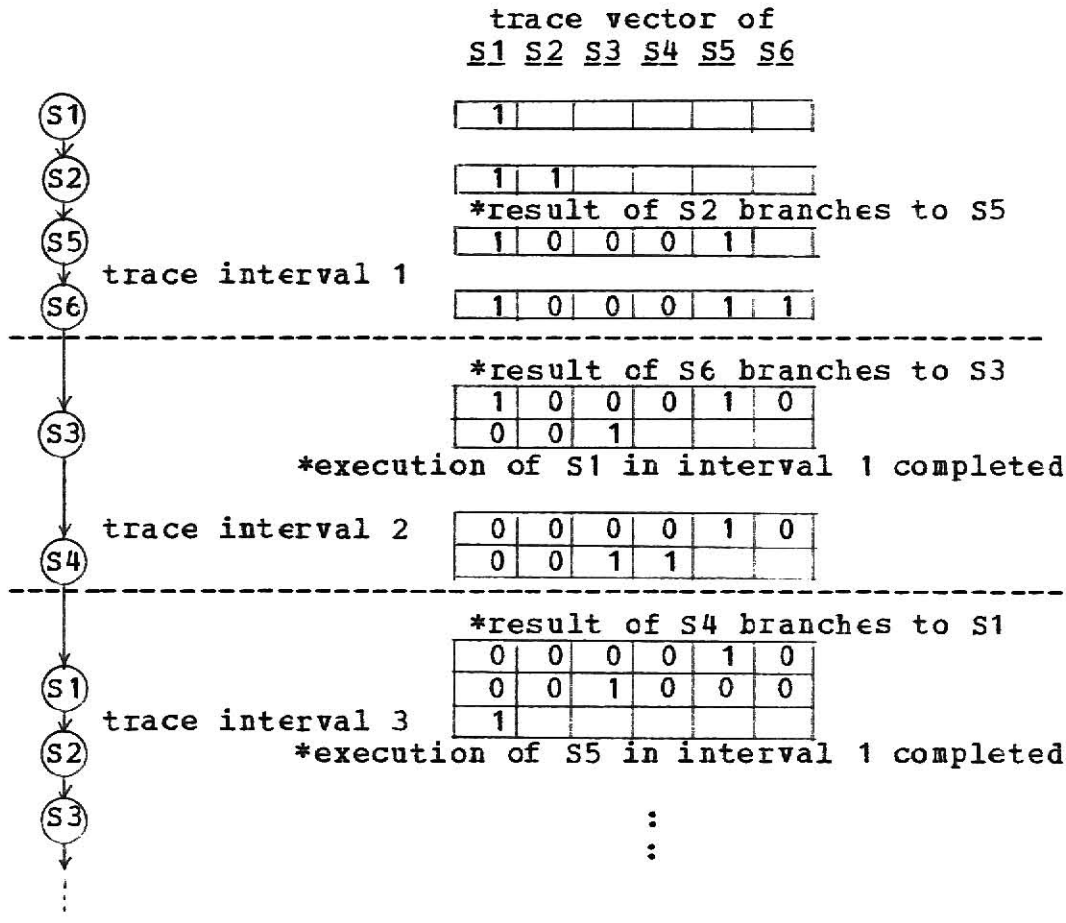
	S1	S2	S3	S4	S5	S6
A	10	00	00	00	00	00
I	01	01	11	00	01	01
J	01	00	00	11	00	01
X	00	11	00	00	01	00
Y	00	00	00	00	10	00
Z	00	00	00	00	00	10

Trace vectors convey two messages: (i) They show whether a statement in the execution route has been completed or not. (ii) They reveal the execution order of statements. An element of the trace vector is denoted as TV(u,Si), where Si is a statement label and u indicates a trace interval. A new trace interval is added when a backward branch is in effect. The element TV(u,Si)=1 menas that Si is being executed in interval u. TV(u,Si)=0 implies the execution of Si is completed or Si does not appear in the execution route in interval u.

Given an example program graph:



the execution route and the possible sequence of trace vectors will be:



The trace vectors will be updated frequently during execution to keep track of the execution order of statements. We consider three cases in updating the trace

vectors.

(1) When a statement Si is fetched for execution, the corresponding TV(u,Si) will be set to one. There is no change in trace interval.

(2) When a statement Sm is fetched as a result of forward branching from statement Sl, the statements between Sl and Sm are not covered in the execution route. Hence, for all Sl < Si < Sm, TV(u,Si) will be reset to zero and TV(u,Sm) = 1. There is no change in trace interval.

(3) A backward branch from statement Sm to Sl means that some statement Sk, such that Sl < Sk < Sm, may precede Sl in execution order. To account for this fact, a new trace interval u+1 will be created when Sl is fetched for execution.

Statements in different trace intervals then can be executed at the same time.

By using a variable reference table and a stack of trace vectors one can determine the execution order of statements, and obtain the shared variable sets of statements. One then can preserve the direct precedence relations and execute a sequential prcgram correctly.

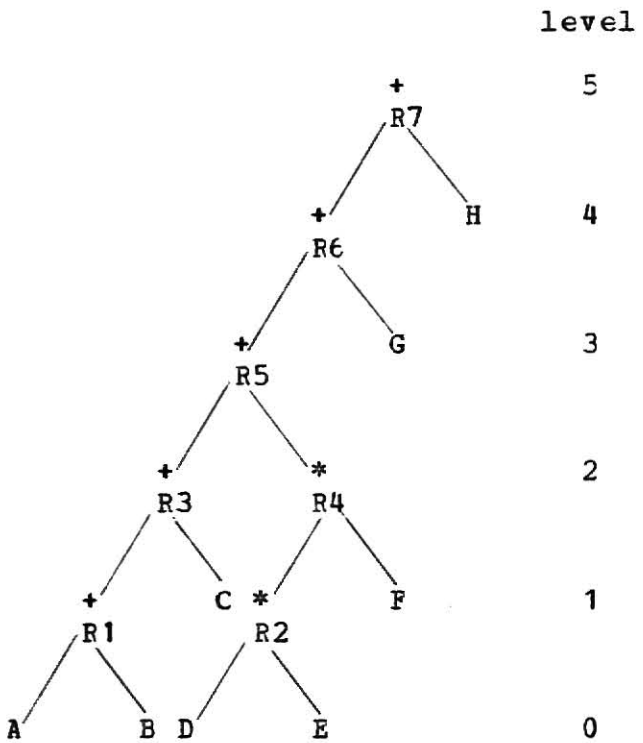
2.4 DETECTION OF PARALLEL COMPONENTS WITHIN ARITHMETIC EXPRESSIONS

Given an expression A+B+C+D*E*F+G+H, there are three different algorithms that can be used to compute the arithmetic expression in parallel. Those are introduced in this section(15,21,23).

(1) Hellerman's algorithm

This algorithm(15) assumes that the input string is written in reverse Polish notation and contains only binary operators. The string is scanned from left to right replacing by temporary results each occurrence of adjacent operands immediately followed by an operator. Those temporary results will be considered as operands during the next passes. Temporary results generated during a given pass are said to be at the same level and therefore can be executed in parallel. The compilation of the given expression listed above is shown as follows:

	input string after the i-th pass	temporary results during the i-th pass
0	AB+C+DE*F*+G+H+	
1	R1C+ R2F*+G+H+	R1 = A+B; R2 = D*E
2	R3R4+G+H+	R3 = R1+C; R4 = R2*F
3	R5G+H+	R5 = R3+R4
4	R6H+	R6 = R5+G
5	R7	R7 = R6+H



(the parallel computation using Hellerman's)

The algorithm seems simple and fast. However, it is difficult to implement for it requires the Polish notation for input string, and it is unable to handle operators which are not commutative.

(2) Squire's algorithm

This algorithm(23) begins with the rightmost symbol of the input string to scan, and proceeds from right to left until an operator is found whose priority is lower than that

of the previously scanned operator (a substring is then formed). Then a left to right scan proceeds the same way to search for the lower priority operator thing in the substring. And the temporary result replaces a pair of operands with one operator and leaves the others. The left to right scans are repeated until no further temporary result can be produced, and at that time, the right to left scan is reinitiated.

The goal of the algorithm is to form quintuples of temporary results of the form:

Ri (op1, operator, op2, start-level, end-level)

where start-level = max(end-level op1; end-level op2)

end-level = start-level + 1

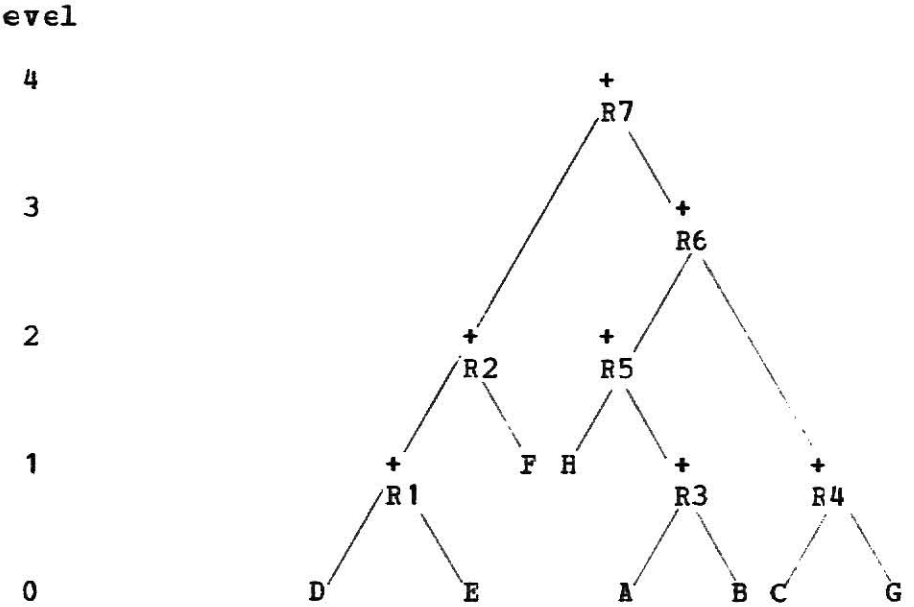
op1 = cperand 1

op2 = operand 2

The results of the process with input string "A+B+C+D*E*F+G+H" are shown as:

right to left scan	left to right scan
D*E*F+G+H	R1*F+G+H
	R2+G+H
A+B+C+R2+G+H	R3+C+F2+G+H
	R4+R3+R2+H
	R4+R5+R2
	R6+R2
	R7

quintuples	op1	operator	op2	start-level	end-level
R1	D	*	E	0	1
R2	F	*	R1	1	2
R3	A	+	B	0	1
R4	C	+	G	0	1
R5	H	+	R3	1	2
R6	R4	+	R5	2	3
R7	R2	+	R6	3	4



(the parallel computation using Sguire's algorithm)

All temporary results which have the same start-level therefore can be computed in parallel. This algorithm can also handle substraction and division with a corresponding increase in complexity. Another feature is that Polish notation plays no part in either the input string or the output quintuples. Because the algorithm requires many scans and comparisons, it becomes more complex as the length of the expression and the diversity of operators within the expression increase.

(3) Ramamoorthy's algorithm

The first step of the algorithm(21) is to rewrite the expression in reverse Polish form and reverse its order. Starting with the rightmost symbol of the string, assign a value to each member based on the following procedure:

Assign to symbol Si, the value Vi = V(i-1) + Ri, i = 1, 2, ..., n

where Ri = 1 - Wi, given that

Wi = 0, if Si is a variable

Wi = 1, if Si is a unary operator

Wi = 2, if Si is a binary operator

and V1 = R1, V0 = 0

Then the procedure follows by (i) Find the first with the highest value of a symbol from rightmost, namely Vm. (ii) Starts from Vm to left, find the first symbol which has a value of 1, departs the string into two substrings. (iii) Consider the rightmost substring, form a new substring consisting of the symbols within the value of Vi = 1 to the right and to the left of Vm. Transpose this substring with the substring to the right of it whose leftmost member has a value of Vi =1. (iv) Repeat the procedure until the initial Vm occupies the position i = 2 in the substring. And apply

to the leftmost substring with the same procedure. As a result, two branches on either side of the root node can be executed in parallel.

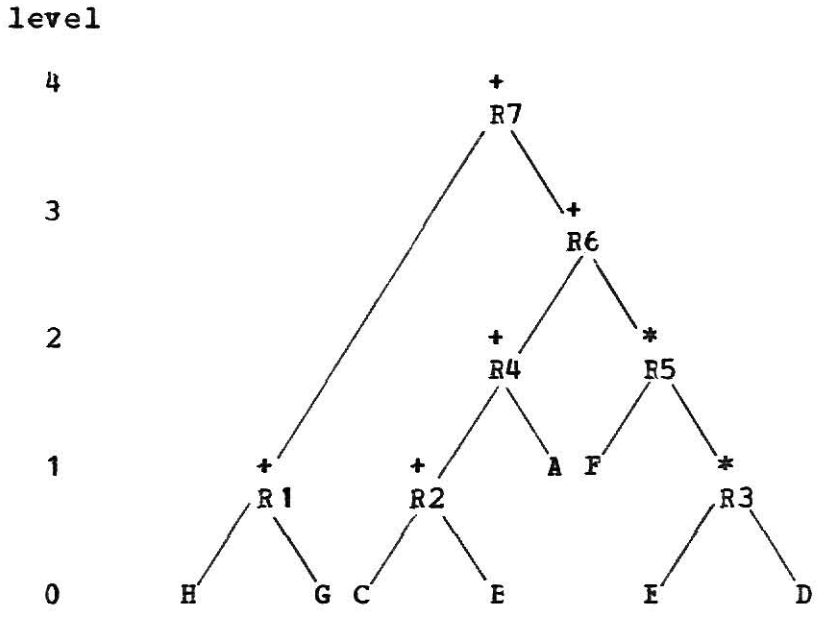
Using this procedure, taking the same example with input string in reverse order of Polish form: "+H+G+*F*ED+C+BA".

the string with value assigned as:

```
    Rcot                                  Vm
i :  15  14  13  12 |  11  10  9   8   7   6   5   4   3   2   1
Si:  +   H   +   G  |  +   *   F   *   E   D   +   C   +   B   A
Vi:  1   2   1   2  |  1   2   3   2   3   2   1   2   1   2   1

initial rightmost   +   *   F   *   E   D   +   C   +   B   A
    substring       1   2   3   2   3   2   1   2   1   2   1
                        ------><-----
                  (transpose this part)

final rightmost     +   +   C   +   B   A   *   F   *   E   D
    substring       1   2   3   2   3   2   1   2   1   2   1
                                            ------><-----
                                      (the transposed part)
```



(the parallel computation using Ramamoorthy's)

In order to implement the techniques mentioned here for components within arithmetic expressions, several features are desirable. Schemes for detecting parallel processable components are oriented primarily needed. Also, string manipulation ability would be highly desiable, and an associative memory could reduce execution time in the implementation of precedence partitions.

CHAPTER III

FACTORING ON FINITELY INDUCTIVE SEQUENCES

3.1 INTRODUCTION

IF a sequence

a0,a1,a2,a3,...

of letters from an alphabet A of K many letters have the property that for some fixed n >= 1, the choice of ai for all i >= n depends only on the choices of

a(i-n), ..., a(i-2), a(i-1).

such a sequence is called a Finitely Inductive sequence, and the least such n is called the inductive base of the sequence.

A function F having the property that:

(A) the domain of F is a set of n-tuples of letters from A, and the range of F is contained in A,

and

(B) for each i=n, n+1, n+2, ..., the n-tuple (a(i-n), ..., a(i-2), a(i-1)) is in the domain of F

and F(a(i-n), ..., a(i-2), a(i-1)) = ai

This function is defined to be an F.I. function for the sequence, and the minimal such function to be the F.I. function for the sequence.

Consider the following example:

A = {a,b,c,d}

n = 2

F is given by the table: ab --> c
(the left hand side bc --> d
of each notation is cd --> b
called implicant.) db --> d
da --> c
ac --> b

starting segment = ab

resulting sequence: abcdbd

The maximum number of symbols within the implicants is called inductive base, and note that the seguence stops. If the sequence become infinite, the pair (consisting of the function and the starting segment) will be called a full F.I. pair; otherwise, it will be called a partial F.I. pair.

Case & Fisher (7) introduce a ruling as a finite sequence of F.I. pairs in which only the last may be full, and all use the same alphabet(but the inductive bases may

differ). A factorization of an F.I. sequence (or corresponding full F.I. pair) is a ruling in which the sequence generated by this ruling is the same as the F.I. sequence.

In this section, four heuristics are discussed on factoring an F.I. sequence (Finitely Inductive sequence). Also, four examples are given to explain the heuristics' functions. Finally, the applications on factoring are introduced.

3.2 HEURISTICS AND EXAMPLES

(1) THE FIRST HEURISTIC

The first heuristic says that it is good to delete (if possible) one occurrence of some of common letters between implicants in a function table.

Example 1. Given sequence (the vertical stroke devides the period of the sequence) :

aababcabcd|aababcabcd

the function table:

```
   d --> a
  da --> a
  aa --> b
 aab --> a *
  ba --> b
 bab --> c *
babc --> a **
  ca --> b
 cab --> c *
cabc --> d **
```

Here, the first occurrence of those having one single star marked implicants, and the second occurrence of "a","b" were chosen to delete for those having double starred implicants.

the factoring shows:

```
L2  a  abc  cd|a  abc  cd
L1  aababcabcd|aababcabcd
```

function table for L2

```
 d --> a
da --> a
aa --> b
 b --> c
bc --> c
cc --> d
```

function table for L1

```
 da --> a
 aa --> b
babc --> a
 ca --> b
```

(a 2-4 factorization)

So, this heuristic worked guite well at making the top level have minimal base, however, it could cause the lower level to remain with a larger base. In this example, the top level leaves the inductive base equals to 2, but the inductive base in the lower level becomes to 4.

(2) THE SECOND HEURISTIC

In any two level factorization of an F.I. Sequence, the second heuristic simply combines the first heuristic with the function table having minimal implicants for the lowest level cf the factorization.

Example 2. Again, given the same F.I. sequence from the example 1.

aababcabcd|aababcabcd

the function table:

```
 d --> a *
da --> a *
aa --> b *
aab --> a
 ba --> b **
bab --> c
babc --> a
 ca --> b *
cab --> c
cabc --> d
```

There are five implications having lengths less than or equal to 2. However, the second "b" in "bab --> c" has already had its reflection removed in implication "aab --> a" by removing the consequent of "aa --> b". Also, the second "b" in "babc --> a" has already had its reflection removed in "cabc --> d" by removing the consequent of "ca --> b". Therefore, we only remove the consequents of the implications followed by a single star (7).

the resulting 2-2 factorization:

```
L2     abca cd|   abca cd
L1  aababcabcd|aababcabcd
```

function table for L2

```
 d --> a
da --> b
 b --> c
bc --> a
ca --> c
ac --> d
```

function table for L1

```
 d --> a
da --> a
aa --> b
ca --> b
```

(a 2-2 factorization)

Basically, the lower level has the minimal implicants, which is the subset of the function table of the original F.I. sequence's. However, it requires an intuitive method to decide the implicants to be deleted.

(3) THE THIRD HEURISTIC

This procedure starts with a trial factorization of the type n-0-0-0 in which the top level is the given F.I. Sequence, and the function tables in the lower levels are empty. The principal idea of the procedure is to "push down" implicants from the top level as far as possible without violating the bounded base length.

Example 3. Given the same F.I. sequence as before:

aababcabcd|aababcabcd

the starting situation is

L3 aababcabcd|aababcabcd
L2 aababcabcd|aababcabcd
L1 aababcabcd|aababcabcd

table for L3	table for L2	table for L1
d --> a * da --> a * aa --> b * aab --> a ba --> b ** bab --> c babc --> a ca --> b * cab --> c cabc --> d	empty	empty

(a 4-0-0 factorization)

For the first step, as in example 2, the implications that are followed by a single star will be pushed down to the closest lower level, then we can get:

```
L3     abca cd|   abca cd
L2  aababcabcd|aababcabcd
L1  aababcabcd|aababcabcd
```

table for L3	table for L2	table for L1
d --> a da --> b b --> c bc --> a ca --> c ac --> d	d --> a da --> a aa --> b ca --> b	empty

(a 2-2-0 factorization)

Since the implicants in L3 are now equal to or less than the inductive base 2, the process is stopped. This procedure is finished by pushing down all the implicants from L2 to L1, and then all of the implicants from L3 to

L2.

table for L3	table for L2	table for L1
empty	d --> a da --> b b --> c bc --> a ca --> c ac --> d	d --> a da --> a aa --> b ca --> b

(a proper 0-2-2 factorization)

By repeating push-down process, the implicants can be moved one level lower each time. So, a multi-level factorization can be achieved by this heuristic, however intuition is still needed.

(4) THE FOURTH HEURISTIC

The fourth heuristic is a procedure for factoring a given Finitely Inductive Sequence intc a multi-level factorization in which each level has inductive base less than or equal to a given number. That is, the heuristic leads a multi-level factorization by keeping pushing down the implicants having inductive base less than or equal to a given number in each residual function table.

Example 4. Given a binary sequence of base 4 which has only one implicant of length less than 4 but still gets a proper factorization by using the fourth heuristic.

the sequence:

(R1) 1111000100110101111000100110101

the function table:

```
1010 --> 1
0101 --> 1
1011 --> 1
0111 --> 1
1111 --> 0
1110 --> 0
1100 --> 0
 000 --> 1 *
0001 --> 0
0010 --> 0
0100 --> 1
1001 --> 1
0011 --> 0
0110 --> 1
1101 --> 0
```

The first step is to push down the starred implicant which is the only one of length less than 4.

the residual after (R1):

(R2) 1111000001101111000001101

the function table:

```
  010 --> 1 *
 0101 --> 1
 1011 --> 1
 0111 --> 1
 1111 --> 0
 1110 --> 0
  100 --> 0 *
 1000 --> 0
10000 --> 0
00000 --> 1
  001 --> 1 *
 0011 --> 0
 0110 --> 1
 1101 --> 0
```

the residual after (R2):

```
(R3)    11100001010|11100001010
```

the function table:

```
  010 --> | *
10101 --> 1
  011 --> 1 *
  111 --> 0 *
  110 --> 0 *
  100 --> 0 *
 1000 --> 0
 0000 --> 1
  001 --> 0 *
00101 --> 0
```

the residual after (R3):

```
(R4)    1010|1010
```

the function table:

```
0 --> 1
1 --> 0
```

the result is a 1-3-3-3 factorization :

```
L4      1      01    0|    1      01    0
L3    11100   001 010| 11100   001 010
L2  1111000 0011010|1111000 0011010
L1  1111000100110 10|1111000100110 10
```

table for L4	table for L3	table for L2	table for L1
0 --> 1	010 --> 1	010 --> 1	000 --> 1
1 --> 0	011 --> 1	100 --> 0	
	111 --> 0	001 --> 1	
	110 --> 0		
	100 --> 0		
	001 --> 0		

The fourth heuristic therefore can not only factor an F.I. sequence into a multi-level factorization, but help the large problems to work better than intuitive methods.

3.3 APPLICATIONS

Case & Fisher(7) gave an example of a piece of music, which is a "Minute in G" by J.S. Bach, to apply the fourth heuristic. They uses only letters and numbers to represent the notes and the length of notes separately.

The sequence was started with inductive base of 65. In factoring, the fourth heuristic was used, at each stage, implicants of length less than or equal to 2 were pushed down. The general numerical results were as follows:

sequence	base	total implicants	number <= 2
original	65	102	59
first residual	28	44	35
second residual	4	11	6
third residual	2	4	4
		total	104

The result is a 2-1-2-2 factorization.

In the next chapter, we will apply the heuristic to computer programs. By factoring programs into multi-level factorization, one can both save storage space as well as make the time dependencies rather short.

CHAPTER IV

PROGRAM FACTORIZATION

4.1 INTRODUCTION

In Chapter II, we have introduced some algorithms that apply parallel execution of program statements. In this chapter, we apply the factoring technique discussed in Chapter III to programs to achieve a parallel computational environment.

Given a program, one can take out the operands from every statement and construct to a sequence. If we factor a sequence of a given program into multiple levels as a finitely inductive sequence can be done (mentioned in Chapter III), a program then can be executed in parallel, where each of the levels represent a different machine. The objective of parallel execution will be especially useful to large scale programs in terms of time-saving and space-saving.

4.2 DIFFICULT PROBLEMS

(1) The first stage

To an operand-formed sequence, we only want to regenerate variables as well as constants, which are assuming necessary roles during execution. At first, it seems that the only statements that we need to be concerned with are the assignment statements and some input statements. Taking a static structured program segment as follows:

```
BEGIN

  READ (radius);
  area := 3.14 * radius ** 2;
  circle := 2 * 3.14 * radius;

END
```

the sequence can be given as follows:

|radius area 3.14 radius 2 circle 2 3.14 radius|

The function table for this sequence is:

```
          radius radius --> area   *
                   area --> 3.14   *
                   3.14 --> radius *
area  3.14 radius --> 2
          radius    2   --> circle *
                 circle --> 2      *
          circle    2   --> 3.14   *
 2     3.14 radius --> radius
```

Using the fourth heuristic to push down those having stared implicants given

```
L2|                              2                    radius|
L1|radius area 3.14 radius 2 circle 2  3.14 radius|
```

table for L2

```
   2   --> radius
radius --> 2
```

table for I1

```
radius radius --> area
        area  --> 3.14
        3.14  --> radius
radius    2   --> circle
       circle --> 2
circle    2   --> 3.14
```

Therefore, a static structured program without the conditional statement can be factored properly to be multi-level, and then be executed in parallel. However, for a dynamic structured program, there is no way to handle both conditional statements and iteration statements by using the same step shown here. We then consider the next stage.

(2) The second stage

We began by keeping a record for the reference of variables by marking out the statement's order. That is,

in order to keep track of every variable occurrence in a dynamic structured program, we use the notation "(variable, value, reference)", where the value specifies the variable's value and the reference is the referenced statement number if there is any(otherwise, just omit it). Consider the following example program:

```
BEGIN

  min := a(1);
  max := min;
  i := 2;
  WHILE i < n DO
    BEGIN
      u := a(i);
      v := a(i+1);
      IF u > v
        THEN BEGIN
          IF u > max THEN max := u;
          IF v < min THEN min := v;
        END
        ELSE BEGIN
          IF V > max THEN max := v;
          IF u < min THEN min := u;
        END;
      i := i + 2;
    END;  (* while *)
    IF i = n THEN
      IF a(n) > max THEN max := a(n)
                    ELSE IF a(n) < min
                         THEN min := a(n);
    WRITELN (max,min);

END.
```

We modify those statements into the following form:

```
  statement number

        1          (  min, a(1) ) <-- (  a(1) , a(1)  )
        2          (  max, a(1) ) <-- ( min , a(1), 1 )
        3          (   i ,   2  ) <-- (   2   ,   2   )
        4          (   u , a(i) ) <-- ( a(i), a(i), 3 )
     or 4          (   u , a(i) ) <-- ( a(i), a(i),10 )
```

```
 5        (   v,a(i+1) ) <-- ( a(i+1),a(i+1) )
 6        (  max, a(i) ) <-- (  u  , a(i), 4 )
 7        ( min,a(i+1) ) <-- (  v, a(i+1), 5 )
 8        ( max,a(i+1) ) <-- (  v, a(i+1), 5 )
 9        (  min, a(i) ) <-- (  u  , a(i), 4 )
10        (   i , i+2  ) <-- ( i,i,3 )( 2, 2 )
11        (  max, a(n) ) <-- ( a(n) ,a(n), ? )
12        (  min, a(n) ) <-- ( a(n), a(n), ? )
```

Since the variable a(i) in the statement 4 can be referenced dynamically from either statement 3 or statement 10, a duplicate line is included which provides both references. Also, in statements 11 and 12, we cannot decide the referenced statement either. With this approach, we still can't get a well-factoring program sequence.

(3) The third stage

By utilizing the data structure of a program, we found that a better approach would be to rewrite a program into several program-groups by repeating its iterations. In this stage, we did solve the problem related to the iteration statements.

Given a program segment as follows:

```
BEGIN

  total := 0;
  FOR i := 1 TO n DO
```

```
total := total + bal(i);
FOR i := 1 TO n DO
BEGIN
  READ (dw, amount);
  IF dw = 'deposit'
  THEN BEGIN
    bal(i) := bal(i) + amount;
    total := total + amount;
    sum := sum + amount;
  END
  ELSE BEGIN
    IF bal(i) > amount
    THEN BEGIN
      bal(i) := bal(i) - amount;
      total := total - amount;
    END
    ELSE BEGIN
      bal(i) := 0;
      total := total - bal(i);
    END;
    sum := sum -amount;
  END;
END;  (* for *)
IF sum >= 0 THEN result := result + sum
            ELSE result := result - sum;

END;
```

If we duplicate the iteration statements, then the new program would be rewrited as follows:

```
BEGIN

  total := 0;  (* P1 *)
  total := total + bal(1);  (* P21 *)
  total := total + bal(2);  (* P22 *)
    :
    :
  total := total + bal(n);  (* P2n *)

  READ (dw, amount);  (* P31*)
  IF dw = 'deposit'
  THEN BEGIN
    bal(1) := bal(1) + amount;
    total := total + amount;
    sum := sum + amount;
  END
  ELSE BEGIN
    IF bal(1) > amount
```

```
  THEN BEGIN
    bal(1) := bal(1) - amount;
    total := total - amount;
  END
  ELSE BEGIN
    bal(1) := 0;
    total := total - bal(1);
  END;
  sum := sum - amount;
END;

READ (dw, amount);   (* P32 *)
IF dw = 'deposit'
THEN BEGIN
  bal(2) := bal(2) + amount;
  total := total + amount;
  sum := sum + amount;
END
ELSE BEGIN
  IF bal(2) > amount
  THEN BEGIN
    bal(2) := bal(2) - amount;
    total := total - amount;
  END
  ELSE BEGIN
    bal(2) := 0;
    total := total - bal(2);
  END;
  sum := sum - amount;
END;

READ (dw, amount);   (* P33 *)
  :                     :
  :                     :

READ (dw, amount);   (* P3n *)
IF dw = 'deposit'
THEN BEGIN
  bal(n) := bal(n) + amount;
  total := total + amount;
  sum := sum + amount;
END
ELSE BEGIN
  IF bal(n) > amount
  THEN BEGIN
    bal(n) := bal(n) - amount;
    total := total - amount;
  END
  ELSE BEGIN
    bal(n) := 0;
    total := total - bal(n);
  END;
```

```
      sum := sum - amount;
    END;

    IF sum >= 0   (* P4 *)
      THEN result := result + sum
      ELSE result := result - sum;

END;
```

If we split the program segment into several groups (distinguished by Px, where x is the group number), then the data structure turns to be a sequential flow: from P1, P21, P22, ..., P2n, P31, P32, ..., P3n, to P4. Obviously, this sequential data flow can then be treated normally using the factoring technique. Each program-group can be treated as multiple levels in the structure to permit the required parallelism. However, after factoring this example program, we found that within the P3 groups, the problem of deciding the conditional branch and its factoring level still exist. We then moved to the fourth stage of consideration.

(4) The fourth stage

We consider attaching a boolean symbol to each conditional statement. In other words, we rewrite the conditional statements in order to signal a proper branch in each level after proper factoring. We attached to

each conditional statement a boolean, by using the notation "Bi (IF expression) THEN BiT (statements) ELSE BiF (statements)" to rewrite the IF-THEN-ELSE conditional statements, where i is the order of the booleans.

Consider again the same program segment as in the previous stage, and set the value *n* to be the decimal number two. After attaching the true-false booleans, we have the program as follows:

```
BEGIN

  total := 0;  (* P1 *)
  total := total + bal(1);  (* P21 *)
  total := total + bal(2);  (* P22 *)

  READ (dw, amount);  (* P31 *)
  B1 IF dw = 'deposit'
     THEN BEGIN
       B1T bal(1) := bal(1) + amount;
       B1T total := total + amount;
       B1T sum := sum + amount;
     END
     ELSE BEGIN
       B2 IF bal(1) > amount
          THEN BEGIN
            B1F B2T bal(1) := bal(1) - amount;
            B1F B2T total := total - amount;
          END
          ELSE BEGIN
            B1F B2F bal(1) := 0;
            B1F B2F total := total - bal(1);
          END;
          B1F sum := sum - amount;
      END;

   READ (dw, amount);  (* P32 *)
   B1 IF dw = 'deposit'
      THEN BEGIN
        B1T bal(2) := bal(2) + amount;
        B1T total := total + amount;
        B1T sum := sum + amount;
      END
      ELSE BEGIN
```

```
        B2 IF bal(2) > amount
           THEN BEGIN
             B1F B2T bal(2) := bal(2) -amount;
             B1F B2T total := total - amount;
           END
           ELSE BEGIN
             B1F B2F bal(2) := 0;
             B1F B2F total := total - bal(2);
           END;
           B1F sum := sum - amount;
      END;

      READ (dw, amount);  (* P33 *)
        :                    :
        :                    :

      B3 IF sum >= 0  (* P4 *)
         THEN B3T result := result + sum
         ELSE B3F result := result - sum;

END;
```

With this example the new sequence becomes:

```
|total 0 total total bal(1)  total total bal(2)
 dw amount B1  dw='deposit'  B1T bal(1)  bal(1)
 amount  B1T  total total amount  B1T  sum  sum
 amount B2  bal(1)>amount B1F B2T bal(1) bal(1)
 amount  B1F  B2T total  total  amount B1F  B2F
 bal(1) 0  B1F  B2F total total bal(1)  B1F sum
 sum  amount  dw  amount  B1  dw='deposit'  B1T
 bal(2)  bal(2)  amount B1T  total total amount
 B1T  sum  sum  amount  B2  bal(2)>amount  B1F
 B2T bal(2)  bal(2)  amount B1F B2T total total
 amount B1F B2F  bal(2)  0  B1F B2F total total
 bal(2)  B1F  sum  sum  amount  B3  sum>=0  B3T
 result  result  sum  B3F  result  result  sum|
```

Using the fourth heuristic of the factoring technique to push down those boolean branches, we then get the multi-level sequence where each level is continued in next block:

```
L4 |
L3 |
L2 |
L1 |total 0 total total bal(1) total total bal(2)  dw


L4:
L3:
L2:                              B1T bal(1) bal(1) amount
L1: amount B1  dw='deposit'      B1T bal(1) bal(1) amount


L4:                               B1T  sum sum  amount
L3: B1T  total total  amount      B1T  sum sum  amount
L2: B1T  total total  aumout      B1T  sum sum  amount
L1: B1T  total total  amount      B1T  sum sum  amount B2


L4:
L3:                                                   B1F
L2:                  B1F  B2T bal(1) bal(1)  amount B1F
L1: bal(1)>amount    B1F  B2T bal(1) bal(1)  amount B1F


L4:                                                B1F B2F
L3: B2T  total total amount B1F B2F bal(1)  0 B1F B2F
L2: B2T  total total amount B1F B2F bal(1)  0 B1F B2F
L1: B2T  total total amount B1F B2F bal(1)  0 B1F B2F


L4: total total  bal(1)
L3: total total  bal(1)
L2: total total  bal(1) B1F sum sum  amount
L1: total total  bal(1) B1F sum sum  amount dw amount


L4:
L3:                                                   B1T
L2:                     B1T  bal(2)  bal(2)  amount B1T
L1: B1  dw='deposit'    B1T  bal(2)  bal(2)  amount B1T


L4:                            B1T  sum  sum  amount
L3: total  total  amount       B1T  sum  sum  amount
L2: total  total  amount       B1T  sum  sum  amount
L1: total  total  amount       B1T  sum  sum  amount   B2


L4:
L3:                                                   B1F
L2:                  B1F B2T bal(2)  bal(2)  amount B1F
L1: bal(2)>amcunt    B1F B2T bal(2)  bal(2)  amount B1F
```

```
L4:                                                    B1F B2F
L3: B2T total total amount B1F B2F bal(2)   0   B1F B2F
L2: B2T total total amount B1F B2F bal(2)   0   B1F B2F
L1: B2T total total amount B1F B2F bal(2)   0   B1F B2F


L4: total total bal(2)
L3: total total bal(2)
L2: total total bal(2) B1F sum sum amount
L1: total total bal(2) B1F sum sum amount   B3  sum>=0


L4:                                                    |
L3:                                                    |
L2: B3T  result  result  sum  B3F  result  result sum  |
L1: B3T  result  result  sum  B3F  result  result sum  |
```

It seems that we can run this multi-level program in parallel with conditional branches, except that we don't need to regenerate the booleans in the sequence. So, it might be another thought to attach the booleans above certain variables to achieve the parallelism of the factorization.

(5) The fifth stage

In order to achieve a well factored multi-level program for execution, we recognize there are two conditions:

(i) the levels should be about equal in the number of implicants.

(ii) the implicants in each level should represent non-sequential operations.

Further, we recognize that the dynamic structure of the program is the representation which we must use since it represents the actural sequence instances of the static program structure.

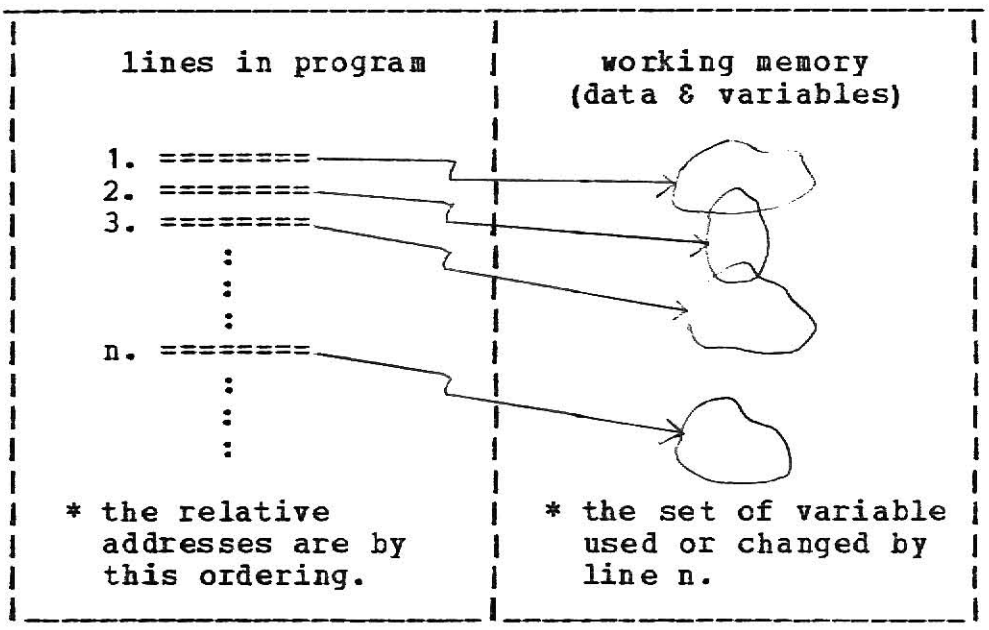
For each statement in program there is an area in the machine memory which represents or contains an equivalent representation of the statement. The actual sequence represents a transition through these areas. However, the factoring must consider all possible sequences which might exist. The method of factoring the program structure is one which now involves describing an alphabet for each of the possible local memory areas. The primitives of the alphabet for factoring programs would look like the follows:

(1) Individuals: letters representing variables, constants, and perhaps letters representing indices in arrays.

(2) Instructions: letters representing instructions -- unary, binary, goto's, and branches are all considered.

(3) Relative Addresses: integers.

The relationship is shown in the following figure:



(figure of the whole memory)

Each instance is as follows:

(A,B,C,D;I;α,β)

where A, B, C, and D are individuals, which are either used or operated upon; I is the instruction; α and β are relative addresses of the next instruction to be executed:

Type (i) -- if the instruction is neither a "branch" nor a "goto" then $\alpha=\beta=1$.

Type (ii) -- if it is a "goto" then $\alpha=\beta$=the relative address of the next line (relative to the present line).

Type (iii) -- is in the case of a trance $\alpha=1$ and β is the relative address of the next

line (a branch).

An implicant would append as

$$(A1,B1,C1,D1;I1;\alpha 1,\beta 1),(A2,B2,C2,D2;I2;\alpha 2,\beta 2),\ldots,$$
$$(An,Bn,Cn,Dn;In;\alpha n,\beta n) \longrightarrow (A,B,C,D;I;\alpha,\beta)$$

It should be noted that in the original program these may not be in order due to "goto's" and "branches". Here the line $(A,B,C,D;I;\alpha,\beta)$ from the original ordering is the line having relative addresses αn or βn relative to line $(An,Bn,Cn,Dn;In;\alpha n,\beta n)$. We trace back for enough to where A, B, C, and D were last changed.

Pushing down an implicant requires changing the relative addresses in the residual, which "includes" or "skips over" the "letter" being pushed down. So when we are consolidating we are consolidating a family of sequences not just one.

There are an infinite number of implicants corresponding to any program with loops. This is not as serious as it may sound. In the factoring method (7), they achieved a "push down" by eliminating a particular occurrence of a single letter. Here suppose one delete a particular occurrence of a "letter", this amounts to pushing down an infinite set of implicants. Pushing down a whole loops then would be pratical.

CHAPTER V

CONCLUSION

For parallel execution(10), to determine the appropriate order of execution, the information can be obtained from two sources : (i) explictly defined by the user via machine instructions, and (ii) directly determined by the computer using a hardware instruction lookahead scheme. We have reviewed several algorithms to detect the parallel components within arithmetic expressions. Also, Benstein(6) has deviced a set of conditions which must be satisfied before two computations can be executed in parallel. Based on those conditions, Gonzalez and Ramamoorthy(12,21) have developed a Fortran parallel paths.

However, consider to the factoring technique, Case & Fisher(7) introduced some heuristics to Finitely Inductive Sequences. And we have mentioned it in Chapter III.

This paper then deals with the exploition of parallelism in a proper-factored program. The focus is on the development of a multi-level program by using the factoring technique, which helps users to speed up the execution of a program in parallel. In other words, the

approach, factoring programs, is derived from the heuristic of F.I. sequences to achieve parallel computation.

The key to speed up the execution of a program is to process as many sequence-levels as possible in parallel. It remains an open question to determine whether there exists algorithms for the speedup ratios with respect to the number of processors available. However, our approach is certainly valuable for an interim period due to the magnitude of factoring technique. It will also give a clear insight into the problems encountered when designing software for the multiprocessor system. To prove its actual worth is much harder to obtain. That program factoring on parallel execution will play one of the vital roles in the future of information processing cannot be denied.

SELECTED BIBLIOGRAPHY

1. Adams, D.A. "A Model For Parallel Computations", Parallel Processor Systems, Technologies, and Applications., pp. 311-333.

2. Anderson, J.P., "Program Structure For Parallel Processing", Communications ACM, Vol. 8, No. 13, Dec. 1965, pp. 786-788.

3. Allam, S.J., and Oldehoeft, A.E. "Loop Decomposition In The Translation Of Sequential Languages To Data Flow Languages", 1980 Parallel Processing, pp. 139-140.

4. Bear, J.L., "A Survey Of Some Theoretical Aspects Of Multiprocessings", ACM Computing Surveys, Vol. 5, No. 1, March 1973, pp. 31-80.

5. Bear, J.L., and Russel, E.C., "Preparation And Evaluation Of Computer Programs For Parallel Processing Systems", Parallel Processor Systems, Technologies, and Applications., pp.375-415.

6. Bernstein, A.J., "Analysis Of Programs For Parallel Processing", IEEE Trans. on EC, Vol. 15, No. 5, Oct. 1966, pp. 757-763.

7. Case, J.H., and Fisher, P.S., "Factoring Finitely Inductive Sequences", 1981, 50 pp.

8. Chamberlin, D.D., "The Single-Assignment Approach To Parallel Processing", 1971 Fall Joint Computer Conf., AFIPS Conf. Proc., Vol. 39, pp. 263-269.

9. Cheung, L., "Dynamic Block Concept", Proceeding of the 1978 International Conf. on Parallel Processing, pp. 52-57.

10. Chin, F.Y., Lam, J., and Chen, I., "Optimal Parallel

Algorithms For The Connected Component Problem", Proceeding of the 1981 International Conf. on Parallel Processing, pp.170-175.

11. Gajski, D.D., Padua, D.A., Kuck, D.J., and Kuhn, R.H., "A Second Opinion On Data Flow Machines And Languages", Computer, Feb. 1982, pp.58-68.

12. Gonzales, M.J., and Ramamoorthy, C.V., "Program Suitable For Parallel Processing", IEEE Trans. Computer, C-20, June 1971, pp. 647-654.

13. Gottlieb, A., and Schwartz, J.T., "Networks And Algorithms For Very-Large-Scale Parallel Computation", Computer, Jan. 1982, pp. 27-36.

14. Gurd, J., and Watson, I., "Data Driven System For High Speed Parallel Computing -- Part 1: Structuring Software For Parallel Execution", Computer Design, June 1980, pp. 91-100.

15. Hellerman, H., "Parallel Processing Of Algebraic Expressions", IEEE Trans on EC, Vol. 15, No.1, Feb. 1966.

16. Knuth, D., The Art Of Computer Programming, Vol.1, Fundamental Algorithms, Addison-Wesley 316.

17. Kuck, D.J., "Parallel Processor Architecture --- A Survey", 1975 Sagamore Conference Proceeding.

18. Kuck, D.J., "A Survey Of Parallel Machine Organization And Programming", ACM Computing Surveys, Vol. 7, No. 1, March 1977, pp. 29-60.

19. Lint, B., and Agerwala, T., "Communication Issues In The Design And Analysis Of Parallel Algorithms", IEEE Trans. on SE, Vol. se-7, No. 2, pp. 174-188, March 1981.

20. Reghbati, E., and Corneil, D.G., "Parallel Computations In Graph Theory", SIMA J. Computing,

Vol. 7, May 1978, pp. 230-237.

21. Ramamoorthy, C.V., and Gonzalez, M.J., "A Survey Of Techniques For Recognizing Parallel Processable Streams In Computer Programs", 1969 Fall Joint Computer Conference, AFIPS Conference Proceeding, Vol. 35, pp. 1-14.

22. Ramamoorthy, C.V., and Leung, M.H., "A Scheme For The Parallel Execution Of Sequential Programs", Proceeding of the 1976 International Conference on Parallel Processing, pp. 312-316.

23. Squire, J.S., "A Translation Algorithm For A Multiprocessor Computer", Proc. 18th ACM National Conference 1963.

24. Wirth, N., "Program Structures For Parallel Processing", Communications ACM, Vol. 9, No. 5, May 1966, pp.320-321.

AN APPROACH OF PARALLEL COMPUTATION ON FACTORING PROGRAMS

by

JIEH-SHWU LIN

B.S., Chinese Cultural College(Taiwan, R.O.C), 1978

AN ABSTRACT OF MASTER'S REPORT

submitted in partial fulfillment of the

requirements of the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1982

ABSTRACT

Whereas the goal of parallel computation is to obtain the higher and possible performance. The factorization of a given sequence would regenerate the same sequence as a result.

This paper first describes several parallel computation algorithms and a technique on factorization of Finitely Inductive Sequences.

Then followed by an approach, which is based on the factoring technigue, to factor programs into multiple levels is been introduced. Examples are given to support this approach to fulfill parallel computations on saving storage space as well as reducing time dependencies.

In the very last, some conclusions and future trends are given.