ASC PRIMER

Department of Mathematics and Computer Science

Abstract

ASC is a high level parallel language developed at the Department of Mathematics and Computer Science, Kent State University for ASsociative Computing. An associative SIMD emulator option has also been developed. The purpose of this paper is to present the basic components of the language and in particular, to show how it works on the emulator. The material for this paper is taken from Professor Potter's publications [5], [6], [7], Hioe's [2] thesis, and Lee's [3] thesis. The handbook consists of six chapters. The first chapter discusses the background of ASC; the second chapter tells the user how to get started; Chapters 3, 4 and 5 discuss the parallel operations of ASC; Chapter 6 presents the additional features of ASC.

Acknowledgement: This document was prepaired through the efforts of many people. In particular, Julia Lee and Dr. Chandra Asthagiri.

CHAPTER 1

The Background of ASC

1.1 Introduction

ASC is a parallel computer language developed at the KSU Department of Mathematics and Computer Science of KSU for SIMD machines, namely the WAVETRACER (WT), and the CONNECTION MACHINE (CM). An emulator has also been developed so that ASC can be executed on any computer with C. The first pass of the compiler is the same for all the machines but the second pass of the compiler is different for each machine. The first pass produces a *.iob* file which contains the intermediate code which is the input for the second pass, and a *.lst* file which shows the relationship between the source code and the triple address intermediate codes. The second pass generates a *.c* file for the Connection Machine and *.c, .wt, .ex & -st.c* file for the WT. The emulator option second pass does not produce an output file. This handbook will provide ASC users with a quick reference guide for executing programs. The bulk of the material is taken from Potter [7], Hioe [2], Michalakes [4] and Lee [3]; the drawings are taken from Lee's master's thesis [3]. It is assumed that the reader is familiar with the basics of computer programming.

1.2 SIMD Machines the Target Architecture

There are several ways to exploit parallelism. One of these is to partition the data and let each block of data be processed by a particular processing element (PE). Here, each PE holds its own data, and all the PEs can execute at the same time to process the data, thus eliminating the *CPU memory bottleneck* that exists in sequential computers. Since there is a one to one relationship between the processors and memories, large amounts of data can be passed between the processors and memory. The same instruction is applied to all the PEs by *broadcasting* the instruction to all the PEs. Moreover, all *active* PEs will execute the same instruction at the same time on their own data, because each PE has its own local memory. This type of computer is called the Single Instruction Multiple Data (SIMD) computer. The SIMD computer consists of a Control Unit, a number of Processing Elements, a number of Memory Modules and an Interconnection Network.

1.3 Associative SIMD Computers

An associative processor (such as the STARAN-E [9]), consists of a conventional sequential computer called the host, a sequential control and an array of processors. Each processor in the array has its own local memory.

There are three basic components in a SIMD computer:

1. Memory
   (a) Sequential Associative Processor Control Memory.
   (b) Parallel Associative Array Memory.
2. A set of Processing Elements (PEs).

3. A Communication Flip Network.

The Control Memory stores the assembled instructions of the program and the array memory stores the data. The array memory, the local memory for each of the processors, provides content addressable and parallel processing capabilities. Each PE is associated with a row of memory. The processing elements consists of three one bit registers, M, X and Y. The M register is used to mask and to select which processors will carry out the broadcast instruction. The X register is used for temporary storage and the Y register for storing the result of a search operation.

1.4 The ASC Emulator

The ASC emulator is written in the C language. At this writing it emulates 300 processors and a parallel array memory of 8000 bits wide. The emulator is constantly evolving and therefore may act slightly different than described here. Questions and suggestions concerning the emulator can be directed to *potter@mcs.kent.edu*

1.5 The Associative Computing Model

ASC is based on the Associative Computing Model of Potter[7]. Data items which are *related* or *associated* are stored as one record and one such record is stored in the memory allocated to one processor in the array. All similar records, one record per PE, may be accessed in parallel. In associative computing, data records are referenced by describing them (their contents) and not by naming them (their address). Since each memory cell has its own processor, there is no need to move data from memory to a CPU. Thus, associative programming promotes parallelism by maximizing the amount of data parallel processing.

The Associative Computing Model consists of a basic cycle of three phases :

`` search, process, retrieve ''

The *search* operation broadcasts the description of the desired object(s) of an association to all the active processors. This allows all processors to search for the desired object(s) in parallel. The processors which locate the desired object in the search phase are called the *active responders.* The *process* phase consists of a sequence of operations which are executed by the active responders. In the *retrieve* phase, values of specific items of the active responders are retrieved and used in subsequent cycles. The basic *search, process, retrieve* cycle can be nested and in any one cycle the process and/or retrieve phase may be omitted [7]. When exiting a nested level, the group of active responders is restored to the previous state.

CHAPTER 2

Getting Started

2.1 How to execute ASC programs on the emulator

In every modern comuting environment, there are networks of different computers. The ASC user must be certain that he is logged on the specific host and in the specific directory on which ASC has been installed. That is, if he wants to execute ASC on the Wavetracer, he must be logged in on the WT host. If she wants to execute on the Connection Machine, she must ber logged on the CM host. If she wants to emulate ASC on a specific machine, the emulator (pass2) must have been compiled for that machine. Moreover, the user must specify the correct option for the particular machine wanted. These are:

```
-e      for the Emulator
-cm     for the CONNECTION MACHINE
-wt     for the WAVETRACER
```

In order to execute ASC, login to the correct machine and directory and specify the ASC command with the proper option and filename. All ASC source code programs must end with an *.asc* extension. An example of an ASC invocation is:

```
%asc -e Myprog.asc
```

or

```
%asc -e Myprog.asc < Myfile.dat
```

If the ¡ *Myfile.dat* redirection is not specified, the program data must be entered interactively. Whether using redirection or entering interactively, a blank line is required to terminate input. If *Myfile.dat* does not end with a blank line, the system will sit waiting for additional input. The user will think that nothing is happening. Kill the job and insert a blank line and restart.

2.2 Program Structure

An ASC program can be identified as a main program or a subroutine (refer to Chapter 6 for subroutines). The first word is MAIN or SUBROUTINE followed by the program name. This statement is then followed by the define statements, the variable declarations and the program body. The last statement is the END statement. All statements end with a semicolon. Note the program syntax is " MAIN name declare body END;" and "SUBROUTINE name declare body END;". The order of the program structure is important, which is as follows:

```
MAIN   program name

       defining constants;
```

```
        equivalencing variables;
        defining variables;
        declaring associations;

        body of program;

END;
```

2.3 Defining Constants

The DEFINE and DEFLOG statements are used to define constants. The DEFINE keyword is used to declare scalar constants, whereas the DEFLOG keyword is used to declare the logical constants. Scalar constants can be decimal, hexadecimal, octal or binary. The letter X indicates hexadecimal, O denotes octal and B is for binary.

Format:

```
DEFINE( identifier, value);
DEFLOG( identifier, value);
```

For example:

```
DEFINE(Maxnum, 200);
DEFINE(MyHex, X'3FA');
DEFINE(MyOct, O'765');
DEFINE(MyBin, B'11001');
DEFLOG(One, 1);
DEFLOG(Zero, 0);
```

2.4 Reserved Words

Reserved words cannot be redefined, so they must not be used as variable names. The reserved words are given in an appendix.

2.5 Declaring Variables

To declare a variable in ASC, the data mode and the data type of the variable must be indicated. The following paragraphs discuss the various modes and types of data.

ASC supports two modes of data items, namely scalar and parallel. Scalar data items are stored in the sequential memory of the computer and the parallel data items are stored in the parallel array memory. Scalar data items are stored in fixed length words (depending of the word size of the host) whereas parallel data items have varying lengths depending on the type of data. The keywords SCALAR and PARALLEL define the mode of the variable. In addition, the character $ indicates parallel mode or parallel operation.

ASC supports eight data types, namely integer(INT), hexadecimal(HEX), octal(OCT), binary(BIN), unsigned(CARD), character(CHAR), logical(LOGICAL) and index(INDEX). Integers may occupy from 2 up to 2^{256} bits, but default to the word size of the host machine (i.e. 32 bits for most mainframes and wrokstations, 16 bits for most PCs). When declaring integers, users must determine the range of the variable since overflow of integer arithmetic operations is not detected by ASC. A logical variable is 1 bit long and are used to store values of either 1 or 0, TRUE or FALSE. An index variable also occupies 1 bit of memory (parallel mode only). Index variables are important in ASC because they link the *search*,

process and retrieve phases [7]. These variables are used for parallel to scalar reduction (or indexing) purposes when using any control structure dealing with parallel variables. When used with a parallel field, the index variable refers to only one specific association entry at a time.

Variable names start with a letter and are at most 32 characters long. Lower and uppercase letters are indistinguisable (i.e. lower case are converted to uppercase). The width of a variable can be specified by using a colon after the variable name followed by the number indicating the field width. This is the general form of declaring variables names :

```
Format:        data-type  data-mode  var1, var2 ;

For example:
          INT SCALAR AA,B;
          INT PARALLEL  Tail[$], Head[$], Weight[$];
          INT FOUND:8[$];              /* 8 bit field */
          CHAR PARALLEL  Node[$];
          HEX PARALLEL Code:4[$];      /* 4 bit field */
          LOGICAL PARALLEL Found[$];
          INDEX PARALLEL XX[$], YY[$];
```

Commas and spaces are used to separate lists of identifiers. In general, either can be used.

2.6 Multi-Dimensional Variables

ASC allows the user to specify multidimensional variables up to three dimensions.

```
                INT PARALLEL AA[$,5];  /* two dimension parallel */
                INT PARALLEL B[$, 5, 5]; /* three dimension parallel */
INT SCALAR C[3,5];  /* two dimensional scalar array */
```

The two dimensional parallel array AA, above,consists of 5 parallel arrays : AA[$,0], AA[$,1], AA[$,2], AA[$,3] and AA[$,4]. Note that array indicies start at zero and go to n-1.

```
Parallel Array AA :

AA[$,0]  AA[$,1]  AA[$,2] AA[$,3] AA[$,4]
-------------------------------------
  2        7        6      9       3
  5        0        1      6       3
  6        8        8      2       4
  3        5        6      3       5
  .        .        .      .       .
  .        .        .      .       .
```

2.7 The DEFVAR Statement

The DEFVAR statement allows the user to define the location of a variable in memory by defining the beginning of the field. This is helpful when the user needs to overlap two or more variables. The DEFVAR statement must come before the variable declaration and the second variable must be declared before the first variable.

Format:

```
DEFVAR(var1, var2);
DEFVAR(var1, absolute address);
```

For Example:

```
DEFVAR (AA, B);          /* define AA in terms of B */
DEFVAR (C,128);           /* define where C begins */
HEX PARALLEL B:24[$];
HEX PARALLEL AA:4[$,6];        /*  AA overlap with B */
INT PARALLEL C[$];    /* declare C at location 128 */
```

An absolute address option is provided to allow interface to assembly language and host machine native languages (i.e. C* on the CM, and MultiC on the WT). In general, it should not be used by the beginning programmer.

2.8 The scalar IF statement

ASC has two kinds of IF statements, scalar and parallel. The scalar IF is similar in function to the IF statements found in conventional languages; it is a branching statement. When the IF statement evaluates to TRUE the body of the IF part is executed, otherwise the body of the THEN part is executed. Note that the ELSE substatement is optional.

Format:

```
IF  scalar logical expression  THEN
          body
< ELSE
          body >
ENDIF;
```

Example:

```
IF  AA .EQ. 5 then
   SUM = 0;
ELSE
   B = SUM;
ENDIF;
```

The parallel IF is used for parallel searching and is discussed is section 4.2.

2.9 Establishing Associations Between Variables

Associations can be established for parallel variables by using a logical parallel variable. Variables that are associated are related and can be referenced as a group using the logical parallel variable. The parallel variables must be declared prior to establishing the association.

```
Format:
        ASSOCIATE var1, var2 WITH logical parallel variable ;
```

```
For example:
                    CHAR PARALLEL Node[$];
                    INT PARALLEL Level[$], Child[$];
                    LOGICAL PARALLEL Tree[$];

                    ASSOCIATE Node[$], Level[$], Child[$] WITH Tree[$];
```

Node[$]	Level[$]	Tree[$]
R	0	1
A	1	1
B	1	1
C	2	1
D	2	1
0	0	0
0	0	0

In the example above, the parallel variables Node and Level are grouped as an association by the logical parallel variable Tree.

2.10 The Use of Operators in ASC

The use of relational operators, logical operators and arithmetic operators is as follows:

2.10.1 Relational Operators

```
Less than :                 .LT. or  <
Greater than :              .GT. or  >
Less than or equal to :     .LE. or  <=
Greater than or equal to :  .GE. or  >=
Equal to :                  .EQ. or  ==
Not equal to :              .NE. or  !=
```

2.10.2 Logical Operators

```
Negation :        .NOT.    or   !
Or :              .OR.     or   ||
And :             .AND.    or   &&
Exclusive Or :    .XOR.    or   ^^
```

2.10.3 Arithmetic operators

Operation	Operator	Operand type	Result type
Multiplication	*	real	real
Multiplication	*	integer	integer
Multiplication	*	real, integer	real
Division	/	real, integer	real
Division	/	integer	integer
Division	/	real, integer	real
Addition	+	real, integer	real
Addition	+	integer	integer
Addition	+	real, integer	real
Subtraction	-	real	real
Subtraction	-	integer	integer
Subtraction	-	real, integer	real

2.11 Parallel Arithmetic Operations

Arithmetic operations can be performed in parallel, which means that the operation applies to all the active responders in one operation. In a sequential computer, this would require a loop. For example, in Pascal, a FOR loop is needed to add a value into an array. But in ASC only one operation is needed to process the whole array.

For Example:

```
AA[$] = B[$] + C[$];
```

Before:

Mask	AA[$]	B[$]	C[$]
1	2	4	5
1	6	8	8
1	3	10	1
0	5	4	3
0	6	7	2
1	5	6	7

After:

Mask	AA[$]	B[$]	C[$]
1	9	4	5
1	16	8	8
1	11	10	1
0	5	4	3
0	6	7	2
1	13	6	7

In this example, mask of 1 indicates active responders.

NOTE: Currently, the EMULATOR does not support real variables.

2.12 The Assignment Statement

The assignment statement is indicated by an equal sign. There are three types of assignment in ASC, namely assigning to a scalar variable, assigning to a parallel variable and assigning to a logical parallel variable. Any arithmetic expression that evaluates into a scalar value can be assigned to a scalar variable; any parallel arithmetic expression can be assigned to a parallel variable; and any logical parallel expression can be assigned to a logical parallel expression. The data types must be the same on the left and the right hand sides.

Format:

scalar variable = scalar expression
parallel variable = scalar expression or parallel expression
logical parallel variable = logical parallel expression

For example :

```
INT SCALAR  K;
INT PARALLEL AA[$], B[$];
LOGICAL PARALLEL USED[$];
INDEX PARALLEL XX[$];

K = AA[XX] + 5;
B[$] = AA[$] + 5;
B[$] = 3 + 5;
USED[$] = AA[$] .EQ. 5;
```

Given:

```
MASK          AA[$]  B[$]    USED[$]      XX[$]
-------------------------------------------------
 1              7     12        0           0
 1              5     10        1           0
 1              3     8         0           0
 1              9     14        0           1     <--
 1              5     10        1           0
 1              5     10        1           0
 0              0      0        0           0
```

```
then:
   K = AA[XX] + 5 ,
is evaluated
   K = 9 + 5 = 14.
```

2.13 Comments, Delimiters and Program Lines

Comments are enclosed by the symbol /* and */ as in the C language. For example the statement /* *this is a comment* */ will not be processed by the compiler, because it indicates a comment. Delimiters that separate language elements are blanks, new lines and comments. Furthermore ASC program lines must not exceed 132 characters or the result will be unpredictable. Comments may be nested. Thus an unballanced number of comment delimeters may cause an end of file error.

2.14 Embedded Assembler Code

This statement is used when the programmer wants to embed assembler code within the ASC source program. This is useful only when the user is working on an actual machine (i.e. CM or WT) instead of the emulator. The embedded statements must be preceded by the keyword ASMCODE and terminated by the keyword ENDASM. As in the example below, any legal *C** (for the CM) or multiC (for the WT) is allowed. The user is responsible for the correctness of those embedded codes.

```
For example:
                ASMCODE
                  printf (''This is an example\n'');
                ENDASM;
```

CHAPTER 3

Parallel Input and Output

3.1 The Parallel READ Statement

Parallel input in ASC is accomplished by the READ statement. It deals with the contents of the parallel array memory; therefore it works only with parallel variables and not with scalar variables. The association of variables must be established before the READ statement is executed because the parallel variables are read as a group.

```
Format:    READ    parvar1, parvar2   IN  logical parallel variable;
```

Example program :

```
MAIN   Try1

CHAR PARALLEL   Tail[$], Head[$];
INT PARALLEL  Weight[$];
LOGICAL PARALLEL  Graph[$];

ASSOCIATE  Tail[$], Head[$], Weight[$] with Graph[$];

READ Tail[$], Head[$], Weight[$] in Graph[$];

END;
```

3.2 The Input File

The input file for the parallel array is organized into columns. The position of the columns corresponds with the position of the parallel variables in the READ statement. In the Try1 program, for example there are 3 parallel variables that are read in. The input file for this program could be as follows:

```
a       b       40
a       c       30
b       a       38
b       c       24
c       a       26
c       b       20
< BLANK LINE >
```

Note that a blank line is the terminator of the input file. The contents of the parallel array memory after the READ statement are as follows:

TAIL	HEAD	WEIGHT	GRAPH[$]
a	b	40	1
a	c	30	1
b	a	38	1
b	c	24	1
c	a	26	1
c	b	20	1
0	0	0	0
0	0	0	0

3.3 The Parallel PRINT Statement

Parallel output of variables in ASC is accomplished by the PRINT statement, which deals with the contents of the array memory only and not with the scalar variables. The PRINT statement displays a group of variables. Thus the parallel variables to be displayed must be associated. The PRINT statement does not output user specified strings (i.e. text), it only outputs the values of parallel variables. Printing strings or text and scalar variables is accomplished by the MSG statement (see Section 3.4).

```
Format:    PRINT    parvar1, parvar2   IN  logical parallel variable;
```

```
Example program :
          MAIN   Try2
          CHAR PARALLEL   Tail[$], Head[$];
          CHAR PARALLEL  Weight[$];
          LOGICAL PARALLEL  Graph[$];
          ASSOCIATE  Tail[$], Head[$], Weight[$] with Graph[$];
          READ Tail[$], Head[$], Weight[$] in Graph[$];
          PRINT Tail[$], Head[$], Weight[$] in Graph[$];
          END;
```

The output file begins with the message *dump of association* and is followed by the contents of the parallel variables printed in columns.

```
For example:

      DUMP OF ASSOCIATION RESULT FOLLOWS:
      TAIL, HEAD, WEIGHT,
                  a       b       40
                  a       c       30
                  b       a       38
                  b       c       24
                  c       a       26
                  c       b       20
```

3.4 The MSG Statement

The MSG statement is used for displaying scalar variables and messages. Variables are displayed on the line following the message.

```
Format :      MSG  " string "  list ;

For example:
              MSG  " The answers are "  AA  B[XX] B[$];
              MSG  " In the while loop the value of B is "  B ;
              MSG  " Sampai bertemu lagi, Goodbye! " ;
```

The MSG statement may be used to dump parallel variables for debugging purposes. When a parallel variable is specified, the contents of the field for the entire array is printed regardless of the status of the active responders or association variables.

CHAPTER 4

Parallel Searching

Parallel searching in ASC can be accomplished by several different statements. These include the SETSCOPE statement, the different IF statements and the ANY statement. Each of them is discussed briefly in the following sections.

4.1 The SETSCOPE Statement

There are several ways to mark the set of active PEs. The simplest way is to use the SETSCOPE statement. Here, masking is used to mark the active PEs (refer to section 1.3 for masking).

```
Format :        SETSCOPE   logical parallel variable ;
                          body
                ENDSETSCOPE;

Example:
                USED[$] = AA[$] .EQ. 5;

                SETSCOPE USED[$]
                         TAIL[$] = 100;
                ENDSETSCOPE;

or

                SETSCOPE AA[$] .EQ. 5;
                         TAIL[$] = 100;
                ENDSETSCOPE;
```

BEFORE:

AA[$]	USED	TAIL
5	1	7
23	0	6
5	1	9
41	0	7

AFTER:

AA[$]	USED	TAIL
5	1	100
23	0	6
5	1	100
41	0	7

All the PEs whose USED bit has a value of 1 will set the TAIL field to 100.

4.2 The Parallel IF-THEN-ELSE Statement

The parallel IF-THEN-ELSE is different from the conventianal IF statement, because it is actually a masking statement and not a branching statement. This IF statement refers to the active responders of the search process and both body parts of the IF statement are

excecuted. The parallel IF statement below illustrates the masking of the active responders as follows [7]:

1. Save the mask bit of processors that are currently active.
2. Broadcast code to the processors to calculate the IF condition.
3. Set the individual cell mask bit of the active processors to TRUE if its local condition is TRUE. Set the mask bit of the active processors to FALSE otherwise.
4. Broadcast code for the TRUE portion of the "IF" statement.
5. Compliment the mask bits that are obtained in step 3.
6. Broadcast code for the FALSE portion of the "IF" statement.

```
Format:          IF   logical parallel expression  THEN
                     body of then
                 < ELSE
                     body of else >
                 ENDIF;
Example:
                 IF (T[$] .EQ. 1) THEN   /* search T .EQ. 1 */
                      T[$] = 0;          /* process           */
                 ELSE                    /* search T .NE. 1 */
                      T[$] = -1;         /* process           */
                 ENDIF;
```

BEFORE:

T	ORIGINAL MASK
1	1
7	1
2	1
1	1

AFTER:

T	THEN MASK	ELSE MASK
0	1	0
-1	0	1
-1	0	1
0	1	0

4.3 The IF-NOT-ANY statement

The IF-NOT-ANY statement is different from the IF-THEN-ELSE statement in that only one body part is executed. The IF-NOT-ANY statement evaluates the conditional expression and if there are one or more active responders, the THEN statement block is executed. On the other hand, if there is not even one active responder, the ELSE-NOT-ANY statement block is executed. This statement is a masking statement, but when executing the ELSENANY part, the mask used is the original mask existing prior to the IF-NOT-ANY statement.

```
Format:
        IF  logical parallel expression THEN
```

```
      body of if
ELSENANY
      body of not any
ENDIF;
```

For Example:

```
IF AA[$] .GE. 2  .AND. AA[$] .LT. 4  THEN  /* set mask */

    IF  B[$] .EQ.  12 THEN     /* search for B .EQ. 12 */
            C[$]  = 1;         /* process                */
    ELSENANY                   /* search for B .NE. 12 */
            C[$]  = 9;         /* process                */
    ENDIF;

ENDIF;
```

BEFORE:

AA	B	C
1	17	0
2	13	0
2	8	0
3	11	0
2	9	0
4	67	0
0	0	0
0	0	0

AFTER:

AA	B	C
1	17	0
2	13	9
2	8	9
3	11	9
2	9	9
4	67	0
0	0	0
0	0	0

4.4 The ANY Statement

The ANY statement is used to search all data items that satisfy the conditional expression. There must be at least one responder for the body statement to be performed. If there are no responders the ANY statement does nothing (unless an ELSENANY is used). The mask that is used to execute the body part is the original mask prior to the ANY statement. Thus, all active responders are affected if the conditional expression of the ANY evaluates to TRUE.

Format:

```
ANY   logical parallel expression
    body
< ELSENANY
    body >
ENDANY;
```

For example:

```
IF AA[$] .GT. 7  THEN      /* set mask  */
```

```
            ANY AA[$] .EQ. 10
                  B[$] = 11;
            ENDANY;

      ENDIF;
```

BEFORE:

MASK	AA	B
0	3	0
0	5	0
1	16	0
1	10	0
1	8	0
0	7	0
0	0	0
0	0	0
0	0	0

AFTER:

MASK	AA	B
0	3	0
0	5	0
1	16	11
1	10	11
1	8	11
0	7	0
0	0	0
0	0	0
0	0	0

In the given example there are three responders to the IF statement in row 3, row 4 and row 5. Therefore, the B column is changed from 0 to 11 at those rows.

The ANY statement can also be used with the ELSENANY clause as in section 4.3. If there are no responders then the ELSENANY body part is executed.

Format:

```
ANY   logical parallel expression
    body
ELSENANY
    body
ENDANY;
```

For example:

```
IF AA[$] .GT. 7  THEN      /* set mask  */

      ANY AA[$] .EQ. 10
            B[$] = 11;
      ELSENANY
            B[$] = 100;
      ENDANY;

IF;
```

BEFORE:

MASK	AA	B
0	3	0
0	5	0

AFTER:

MASK	AA	B
0	3	0
0	5	0

1	16	0	1	16	100
1	11	0	1	11	100
1	8	0	1	8	100
0	7	0	0	7	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

CHAPTER 5

Looping and Retrieving

5.1 The LOOP-UNTIL Statement

ASC supports a variety of control structures. The LOOP statement, used for looping, resembles the REPEAT UNTIL in Pascal, but it is more flexible, since the UNTIL conditional test can appear anywhere in the body of the loop.

Format:

```
FIRST
       initialization
LOOP
       body 1

UNTIL (logical scalar expression)
      (logical parallel expression)
      (NANY logical parallel expression)

       body 2

ENDLOOP;
```

For example :

```
FIRST
       I = 0;
LOOP
   IF AA[$] .EQ. I  THEN
       B[$] = AA[$] + 2;
   ENDIF
   I = I + 1;
UNTIL I .GT. 10
ENDLOOP;
```

In this example, the variable I is initialized to zero and incremented each time it enters the loop. At each iteration of the loop variable I is tested and if the test evaluates to true, then the loop terminates.

The UNTIL expression may be scalar (as above) or paralle. If parallel, UNTIL exits based on a responder test. It may contain an optional NANY keyword. Without NANY, a parallel UNTIL will exit when responders are (first) detected. With NANY, the UNTIL will exit when the no-responders condition is detected. Thus a parallel UNTIL statement may be used to loop until all responders have been processed as in

```
UNTIL NANY C[$]
```

where C[$] represents a logical parallel expression. Both parallel and scalar UNTILs may be used anyplace in the same loop.

5.2 The Parallel FOR-LOOP Statement

The FOR statement is used for looping and retrieving. It evaluates the conditional expression and stores the resulting active responders in an index variable. This index variable can then be used to retrieve a data item of an association. Active responders are processed one after another until all the active responders have been processed. Note that the logical expression must be in the parallel mode. For example:

```
SUM = 0;
FOR XX IN TAIL[$]  .NE. 999
      SUM = SUM + VAL[XX];
ENDFOR XX;
```

```
TAIL     XX    VAL
-----------------
 3        1    10        1ST TIME THRU LOOP: SUM = SUM + 10 = 10
 5        1    20        2ND TIME THRU LOOP: SUM = SUM + 20 = 30
 999      0    30
 6        1    40        3RD TIME THRU LOOP: SUM = SUM + 40 = 70
```

Here, the index variable XX controls the loop. The loop is repeated only where XX contains the digit 1 starting from the top to the bottom. At each iteration XX is used as a subscript to retrieve the parallel variable VAL. The contents of XX is updated at the bottom of the loop (The first one is changed to zero). So,the value of each of the parallel variable VAL is accumulated in the scalar variable SUM for all the active responders.

5.3 The Parallel WHILE Statement

The WIIILE statement is similar to the FOR except that the WIIILE reevaluates the conditional expression before each iteration.

```
Format:
        WHILE   index variable   IN   logical expression
                body
        ENDWHILE  index variable ;
```

For example:

```
SUM = 0;
WHILE XX IN  AA[$] .EQ. 2
     SUM = SUM + B[XX];
     IF C[XX] .EQ. 1 THEN
```

```
                    IF  AA[$] .EQ. 2 THEN
                          AA[$] = 5;
                    ENDIF;
               ELSE
                    AA[XX] = 7;
               ENDIF;
          ENDWHILE XX;
```

BEFORE:

AA	B	C
1	17	0
2	13	0
2	8	1
3	11	1
2	9	0
4	67	0

AFTER:

AA	B	C
1	17	0
7	13	0
5	8	1
3	11	1
5	9	0
4	67	0

As shown, the parallel variable AA is tested at the beginning of each iteration. Inside the loop the parallel variable AA is changed. The iteration is terminated when the conditional expression evaluates to false or no responders.

The programmer must insure that the logical expression will eventually evaluate to flase or no responders or the while loop will not terminate and may hang up the system. It is advisable for undebugged while statements to include a counter to limit the number of iterations. For example,

```
I = 5;
SUM = 0;
WHILE XX IN  AA[$] .EQ. 2 .and. I .GT. 0
     I = I-1;
     SUM = SUM + B[XX];
     IF C[XX] .EQ. 1 THEN
         /* I FORGOT TO UPDATE AA[xx] */
     ELSE
            AA[XX] = 7;
     ENDIF;
ENDWHILE XX;
IF I .LE. 0 THEN
   MSG "BUG IN WHILE STATEMENT";
ENDIF;
```

5.4 The NEXT Statement

The NEXT statement is used to retrieve the "topmost" memory word of the active responders masked by a logical parallel variable. After retrieving, the topmost bit of the logical parallel variable is set to zero. NEXT is almost always used within a looping structure. NEXT may contain an ELSENANY substatement.

```
Format:
        NEXT  parallel index variable IN  logical parallel variable

           body
        < ELSENANY
           body  >
        ENDNEXT parallel index variable;
```

```
For example:
                        INT PARALLEL AA[$], B[$];
                        LOGICAl PARALLEL  USED[$];
                        INDEX PARALLEL XX[$];

                        USED[$] = AA[$] .EQ.  4

                        NEXT  XX in USED[$]
                               B[XX] = -1;
                        ENDNEXT XX;
```

BEFORE:

AA	USED	B
1	0	2
4	1	2
4	1	2
19	0	2
4	1	2
4	1	2
17	0	2
4	1	2

AFTER:

AA	USED	B
1	0	2
4	0	-1
4	1	2
19	0	2
4	1	2
4	1	2
17	0	2
4	1	2

In this example, USED is the logical parallel variable and the topmost bit that contains the digit 1 is in the second row. Therefore, at the second row the contents of variable B are changed to -1 and the digit of the variable USED is reset to zero. Note that

```
NEXT XX IN AA[$] .EQ. 4
  :
```

is not allowed.

5.5 The GET Statement

To access the value of a specific item in the memory of an active processor, the GET statement is used. It evaluates the logical parallel expression and uses the index parallel variable to mark the first of the active responders which meet the specified condition. In the body of the GET statement the index parallel variable is used to access the specific item. If there are no responders the body of the GET statement is not executed. Get may also contain an ELSENANY substatement.

Format:

```
GET  index parallel variable IN
                  logical parallel expression
     body
< ELSENANY
     body >
ENDGET  index parallel variable
```

For example:

```
GET XX IN TAIL[$] .EQ. 1
       VAL[XX] = 0;
ENDGET XX;
```

BEFORE:

TAIL	VAL
10	100
1	90
2	77
1	83

AFTER:

TAIL	VAL
10	100
1	0
2	77
1	83

THE FIRST RESPONDER IS THE SECOND ENTRY, SO VAL[$] IS CHANGED.

Here, the topmost of the active responders is in row 2, so the variable VAL is changed in row 2. The index parallel variable is not updated by GET, so if GET is in a loop, it will select the same responder on every iteration unless if the values of the logical parallel expression are explicitly changed.

CHAPTER 6

Programming At Large

6.1 Modular Programming : The CALL statement

The subroutine protocol in ASC differs from other languages. In most languages the variables in the calling routine are mapped onto variables in the called routine by their position. In ASC, the actual mapping between fields are specified explicitly. The notation is similar to the UNIX/C file redirection. Suppose subroutine X is calling subroutine Y; Variables AA[$], B[$], and C[$] are in X and variables M[$] and N[$] are in Y; and AA[$] and B[$] are to be mapped onto M[$] and N[$] as input data; and Variable O[$] in Y returns data to variable C[$] in X ; then the call would be :

```
subroutine X
    /* declare variables AA, B, and C here */

    CALL  Y  M[$] < AA[$]  N[$] < B[$]       O[$]  > C[$]

end;
```

Any number of variable/fields may be mapped and they may be specified in any order. The formal syntax is as follows:

```
Grammar:

 sub-call := CALL sub-id args;
 args     := in-arg args | out-arg args | e
 in-args  := destination-field  <  source-field-expression
 out-arg  := source-field-expression  >  destination-field

Format:

CALL subroutine called-field1 < calling-field1 ... calling-fieldn > called-fieldn;

Example:  CALL  BITO AA[$]  <  B[$]  .GT. 5;
```

6.2 Using Subroutines

Subroutines are identified by the keyword SUBROUTINE in the program heading and the structure of the subroutine is the same as the main program (see Chapter 2). The emulator requires the user to put the subroutine and the main program in separate files. All the subroutine files must be named with a suffix *.asc* (just as the main program has the suffix *.asc*), and must be compiled before the main program. The -wt and -cm options require that the subroutines be in the same file as the main program (conventional single file C style).

Variables are shared between the main program and the subroutines using the INCLUDE capability (see section 6.3). In this case the main program and the subroutines allocate the same memory fields to the shared variables like COMMON variables in FORTRAN. In ASC, all variables must be declared in the main program. There are no variables local to a subroutine.

```
Format:
        SUBROUTINE subroutine name
                body
        <RETURN>;
        END;

For example:

        PROGRAM MAIN
                        DEFVAR(AA,B);
                        INT PARALLEL B[$];
                        INT PARALLEL AA[$];
                            body
        END;

        SUBROUTINE SUB1         /* in a separate file */
                        DEFVAR(AA,B);
                        INT PARALLEL B[$];
                        INT PARALLEL AA[$];
                            body
        END;
```

Subroutines conclude with the END statement but may have an optional RETURN statement. The END statement automatically generates a RETURN statement.

6.3 The INCLUDE File

The INCLUDE file statement is especially useful when using subroutines. All the DEFVARs and the declaration of common variables are put into a particular file. Then this file can be included in the ASC program and subroutines by using the INCLUDE statement.

```
Format : #INCLUDE filename.h

For example :
                SUBROUTINE SUB1;
                #INCLUDE  MyFile.h

                  body
                END;

MyFile.h  contain the DEFVARs and the variable declaration, in our example,
as follows:
```

```
DEFVAR(AA,B);
INT PARALLEL AA[$];
INT PARALLEL B[$];
```

6.4 The MAXVAL and the MINVAL Functions

The MAXVAL function returns the maximum value of the specified item among active responders, whereas the MINVAL function returns the minimum value. Consider the following example:

```
IF (TAIL[$]  .NE. 1) THEN  /* set mask */
     K = MAXVAL(VALUE[$]);
ENDIF;

TAIL    VALUE                MASK
-------------                ----
 7        90                   1
 1        12                   0
 5       170   <----           1      Value of K is 170
 3        99                   1


IF (TAIL[$]  .NE. 1) THEN  /* set mask */
     K = MINVAL(VALUE[$]);
ENDIF;

TAIL    VALUE                MASK
-------------                ----
 7        90                   1
 1        12                   0
 5        70   <----           1      Value of K is 70
 3        99                   1
```

6.5 The MAXDEX and the MINDEX Function

The MAXDEX function returns the index of an entry where the maximum value of the specified item occurs, among the active responders. This index is also used to retrieve the related fields. If a minimum value is desired instead of a maximum value, then the MINDEX function is used.

For example:

```
IF (TAIL[$] .NE. 1) THEN     /*  set mask */
   HEAD[MINDEX(VAL[$])]  = -1; /* get index and use it */
ENDIF;

BEFORE:                              AFTER:
MASK          TAIL   HEAD   VAL          TAIL   HEAD   VAL
---           ---------------            ---------------
 1             8       4    100           8       4    100
 1             5       3     80           5       3     80
```

```
1          7     5     20  <----      7     -1    20  <----
1          6     9     70             6     9     70
0          1     5     10             1     5     10
0          1     4      1             1     4      1
```

In this example, a minimum value for the VAL parallel variable is found in the third row. So the contents of the HEAD variable in the third row is changed to -1.

6.6 The COUNT Function

The function COUNT() returns the number of active responders (See Section 6.9 for an explanation of them[]).

For example:

```
IF  TAIL[$]  .EQ. 1 THEN     /*  mask */
        K = COUNT(THEM[$]);       /*  get the number */
END;

TAIL      MASK           Value calculated for K is 3

------------
 5         0
 1         1
 3         0
 1         1
 1         1
```

6.7 The Nthval and the Nthdex Function

The Nthval function returns the Nth value (the smallest is the first) of an item and Nthdex returns the index variable of that entry.

```
Example :  NUM = NTHVAL(A[$], 3); /* third smallest */
           ENTRY = B[NTHDEX(A[$],3)];  /* associated entry */
```

6.8 Inter-Process Communication

The current version of ASC supports a one dimensional CPU , and two dimensional memory configuration. So a two dimensional array declaration

```
INT PARALLEL ARR[$, 512];
```

would be mapped onto memory in rows and columns. The inter-column communication is achieved by specifying a variable or constant expression in the second column index.

```
Example: Adding adjacent columns, element by element in parallel

         ARR[$,I] + ARR[$,I+1]
```

Inter-row communication is achieved by specifying a variable or constant expression in the first dimension index ("+" is a shift down, "-" is a shift up).

```
Example: Adding column I to column J with every element
                                        shifted one row

        ARR[$+1,I] + ARR[$,J]
```

For more about Inter-process communication see Chapter 3 of Associative Computing [7].

6.9 ASC Pronouns and Articles

To be closer to natural language ASC supports the use of associative pronouns and articles. The associative pronouns are THEM, THEIR and ITS and the associative articles are A and THE.

THEM refers to the most recent set of responders to a logical parallel expression from a search; THEIR is a possesive form of the $ notation; And ITS is an automatically declared parallel index variable. ITS is updated to the most recent index assignment in a FOR, WHILE, NEXT or GET statement. The generic index variable EACH can be used in non-nested statements.

```
Example:
        IF  AA[$]  .GT. 100 THEN
            K = COUNT(THEM[$]);
        ENDIF;

        IF  NODE[$]  .GT. 100 THEN
            IF THEIR LEFTCHILD[$] .LT. 50 THEN
            .....
            ENDIF;
        ENDIF;

        FOR  XX in NODE[$] .GT. 100
             IF ITS LEFTCHILD  .LT. 50 THEN
             ...
             ENDIF;
        ENDFOR XX;

        FOR  EACH NODE[$] .GT. 100
             IF ITS LEFTCHILD  .LT. 50 THEN
        ...
             ENDIF;
        ENDFOR EACH;
```

THE refers to the last value of a parallel variable which was reduced to a scalar. Suppose AA[XX] is reduced most recently then THE AA may be used as an alternate form of AA[xx].

The indefinite article A refers to the first entry of an associaton and its used is similar to a get. (NOTE, this document was initially prepaired before the indefinite article feature

was included. As a result it used A[] as a variable name. However, A is now a reserved word. An attempt has been made to change all A[] to AA[], but some may have been missed. In all examples, A[], if present, should be read as AA[ ].)

```
Example:  GET XX in NODE[$]
              K = NODE[XX];
          ENDGET XX;

  is the same as :    K = A NODE;  or
                      K = NODE[A];
```

6.10 Dynamic Storage Allocation

ASC provides dynamic storage allocation by the ALLOCATE and the RELEASE statements. When a new association entry is needed, memory is allocated by the ALLOCATE statement. And the release statement returns the cell to idle status. In general, several association entries may be released simultaneously.

```
Format :  ALLOCATE index-variable IN association name;
              statement-block
          ENDALLOCATE index-variable;

          RELEASE  conditional expression FROM association-name;

Example:
          ALLOCATE XX IN T[$]
               AA[XX] = 100;
          ENDALLOCATE XX;

          RELEASE  AA[$]  .EQ. 100 FROM T[$];
```

Note that the scope of the index variable is limited to the statement-block. READs automatically allocate memory. One cell for every row input.

6.11 The ASC Monitor

ASC has a performance monitor that calculates the number of scalar and parallel operations. The performance monitor can be turned on and off anywhere in the program by the following statements:

```
PERFORM = 1  means monitor is on.

PERFORM = 0  means monitor is off.
```

The values of the scalar and parallel variables can be printed anywhere using the MSG statement as follows:

MSG " Perform Scalar, Parallel " SC_PERFORM PA_PERFORM;

The statement enclosed by quotation marks is a message display; the SC_PERFORM is the counter for scalar operations and PA_PERFORM is the counter for parallel operations; the value of the counters will be printed on the line following the message. Note that the monitor is automatically turned off during the execution of I/O operations. If the performance monitor is on when the program ends, the value of PA-PERFORM and SC-PERFORM are automatically printed.

For example:

```
MAIN Trymon
INT SCALAR K;
INT PARALLEL AA[$], B[$];
INT PARALLEL USED[$];
INDEX PARALLEL XX[$];

LOGICAL PARALLEL LG[$];
ASSOCIATE AA[$], B[$] WITH LG[$];

READ AA[$], B[$] IN LG[$];

K = 0;
PERFORM = 1;      /* monitor on */

WHILE XX IN AA[$] .GT. 0 DO


      IF AA[$] .EQ. B[$] THEN
         AA[$] = B[$] - 5;
      ENDIF;

K = K + 1;
ENDWHILE XX;

PERFORM = 0;    /* monitor off */
MSG " MONITORING SCALAR, PARALLEL" SC_PERFORM PA_PERFORM;

PRINT AA[$], B[$] IN LG[$];
END;
```

6.12 ASC Recursion : The STACKWHILE-RECURSE Construct

ASC does not support general recursion in keeping with the concept of natural language. However, the STACKWHILE-RECURSE construct is useful when layers of nesting of logical parallel expression are needed. A recursive while construct allows " nesting " to a level as deep as the data requires and as the internal stack will allow. Moreover, the compactness of the recursive form greatly reduces the amount of repetitious programming effort. For example, in order to implement a depth first tree search the STACKWHILE-RECURSE construct would be useful. For more explanation about how to use the STACKWHILE-RECURSE construct, see chapter 5 of Associative Computing [7].

```
Format:  STACKWHILE   parallel-index  IN   log-par-expr

              body1

         RECURSE   stack-list  THEN bodya  ENDRECURSE;

              body2

 ENDRSTACKWHILE parallel-index;
```

6.13 Complex Searching : The ANDIF and ANDFOR Statements

Complex searching addresses searching techniques needed to accomodate rule based pattern matching. A matching rule can be expressed in an associative parallel form using ASC ANDIF and ANDFOR statements. The nested ANDIF and ANDFOR constructs can be envisioned as tree searches to a fixed depth, with different search and action specification at every level.

```
Format :      ANDIF     logical parallel expression THEN
                    body
              ENDANDIF;

              ANDFOR   logical parallel expression
                    body
              ENDANDFOR;
```

In the body no control statements are allowed. Note that there is also no else substatements. For more explanation about how to use the ANDIF and ANDFOR constructs, see chapter 6 of Associative Computing [7].

6.14 ASC Debugger

A debugger for ASC on the UNIX is being developed. It should be available in 1992.

BIBLIOGRAPHY

[1] Eisenberg, Ann, (1982) *Effective Technical Communication*, Mc.Graw-Hill Inc.

[2] Hioe, K.H., *ASPROL (Associative Programming Language)*, Master's Thesis, Kent State University, August 1986

[3] Lee, J.L., *The Design and Implementation of Parallel SIMD Algorithms for the Travelling Salesperson Problem* , Master's Thesis, Kent State University, December 1989

[4] Michalakes, John, *ASP-VMS Handbook*, Kent State University

[5] Potter, J.L. (1985) *The Massively Parallel Processor*, MIT Press

[6] Potter, J.L. (1987) " An Associative Model of Computation ", Proceedings of the Second International Conference on Supercomputing, Volume III May 4-7, 1987, pp 1-8

[7] Potter, J.L. (1992) " Associative Computing ", Plenum Publishing Inc., New York,

[8] Price, Jonathan (1984) *How to Write a Computer Manual* The Benjamin/Cummins Publishing Company Inc.

[9] STARAN-E Reference Manual, Ger - 16422, Goodyear Aerospace Corp., November 1977

APPENDIX A

```
/* written by John Michalakes for the ASPRO Emulator in 1987 */
/* modified by Julia Lee for the ASC Emulator in 1990 */
/* modified by Chandra Asthagiri to use 'next' correctly */
MAIN SORT
DEFLOG (ONE, 1);
DEFLOG (ZERO, 0);
INT PARALLEL BEFORE[$];   /* numbers before sorting */
INT PARALLEL AFTER[$];   /* numbers after sorting */
LOGICAL PARALLEL DONE[$], INDATA[$], STORE[$];
INDEX PARALLEL XX[$], YY[$];
INT SCALAR ITEM;      /* temporary variable */
ASSOCIATE BEFORE[$], AFTER[$] WITH INDATA[$];

DONE[$] = ZERO;              /* initialize done to zero */
READ BEFORE[$] IN INDATA[$];       /* input numbers */
STORE[$] = INDATA[$];

PERFORM = 1;          /* turn on monitor */
WHILE YY IN (.NOT. DONE[$]  .and. INDATA[$])
         IF (.NOT. DONE[$] ) THEN
              ITEM = MAXVAL(BEFORE[$] );      /* find largest */
              DONE[ MAXDEX( BEFORE[$] ) ] = ONE; /* mark done */
         ENDIF;

         NEXT XX IN STORE[$]
              AFTER[XX] = ITEM;  /* put largest in next  word */
         ENDNEXT XX;
ENDWHILE YY;

PERFORM = 0;         /* turn off monitor */
PRINT BEFORE[$], AFTER[$] IN INDATA[$]; /* display sorted numbers */
MSG " SCALAR AND PARALLEL OPERATIONS " SC_PERFORM PA_PERFORM;
END;
```

APPENDIX B

```
/* The Parallel Minimum Spanning Tree */
main mst

deflog  (TRUE, 1);
deflog  (FALSE, 0);

char parallel tail[$], head[$];
int  parallel weight[$], state[$];
char scalar node;
index parallel xx[$];
logical parallel nxtnod[$],graph[$],result[$];


associate head[$],tail[$],weight[$],state[$] with graph[$];

read tail[$] head[$] weight[$] in graph[$];

setscope graph[$]
node = tail[mindex(weight[$])];
endsetscope;

if(node .eq. tail[$]) then state[$] = 2; else state[$] =3; endif;
while xx in (state[$] .eq. 2)
asmcode
   printf("node = %c\n",NODE);
endasm;
if(state[$] .eq. 2)then nxtnod[$] = mindex(weight[$]); endif;
node = head[nxtnod[$]];
state[nxtnod[$]]=1;
if(head[$] .eq. node .and. state[$] .ne. 1) then
  state[$] = 0;
endif;
if(state[$] .eq. 3 .and. node .eq. tail[$]) then
  state[$] = 2;
endif;
nxtnod[$] = FALSE; /* must clear when done for next iteration */
endwhile xx;

if (state[$] .eq. 1) then result[$]= TRUE ; endif;
print tail[$] head[$] weight[$] in result[$];


end;
```

APPENDIX C

```
/* PARALLEL NEAREST NEIGHBOR TO SOLVE THE TSP PROBLEM   */
/* WRITTEN BY JULIA LEE 1989                            */
-------------------------------------------------------

/*  ALGORITHM  PARALLEL NEAREST NEIGHBOR :
FOR ALL ENTRY IN THE ATTRIBUTE TABLE
            START-NODE <- TAIL (MINIMUM EDGE )
            CURRENT-NODE <- HEAD (MINIMUM EDGE )
        ADD THE MINIMUM EDGE TO THE PATH
        REPEAT
            FOR ALL TAIL[$] EQUAL CURRENT-NODE
                  CURRENT-NODE <- HEAD (MINIMUM EDGE)
            ADD MINIMUM EDGE TO THE PATH
        UNTIL ALL NODES ARE IN THE PATH
        CONNECT CURRENT-NODE TO START-NODE

*/

main nn

int parallel tail[$],     /* ATTRIBUTE TABLE WITH  FIELDS */
             head[$],           /* TAIL, HEAD AND WEIGHT */
             weight[$],
             tour[$],      /* PARALLEL ARRAY TO MARK TOUR */
             out[$];         /* TO MARK UNNECESSARY EDGES */

int scalar start_node,                  /* STARTING NODE */
           current_node,                 /* CURRENT_NODE */
           num_node,                   /* NUMBER OF NODES */
           prev,                         /* PREVIOUS NODE */
           i;

logical parallel graph[$];   /* LOGICAL  FOR INPUT/OUTPUT */
associate tail[$],       /* ASSOCIATIONS FOR INPUT/OUTPUT */
          head[$],
          weight[$],
          tour[$],
          out[$] with graph[$];


/********************* INPUT ATTRIBUTE TABLE *************/

read tail[$] head[$] weight[$] in graph[$];
```

```
/********************* FIND SMALLEST EDGE *****************/

if (graph[$]) then
   start_node = tail[mindex(weight[$])]; /* SAVE START    */
   current_node = head[mindex(weight[$])];
   tour[mindex(weight[$])] = 1;       /* ADD EDGE TO PATH */
   out[mindex(weight[$])] = 1;
   num_node = maxval(tail[$]);    /* GET  NUMBER OF NODES */
endif;

if (head[$] .eq. start_node) .and.
     (tail[$] .eq. current_node) then
   out[$] = 1;            /* DELETE REVERSE */
endif;

if (tail[$] .eq. start_node) .or. (head[$] .eq. start_node) then
   out[$] = 1;                       /* DELETE UNNEC EDGES */
endif;

if (head[$] .eq. current_node) then
   out[$] = 1;                       /* DELETE UNNEC EDGES */
endif;

/********* REPEAT UNTIL ALL NODES ARE IN THE PATH **********/

first

i = 1;

loop
prev = current_node;

/***************** BRANCH TO NEAREST NEIGHBOR *************/

if (tail[$] .eq. current_node) .and. (out[$] .ne. 1) then

   current_node = head[mindex(weight[$])];
   tour[mindex(weight[$])] = 1;  /* ADD  EDGE TO THE PATH */
   out[mindex(weight[$])] = 1;

endif;

if (head[$] .eq. prev) .and.

     (tail[$] .eq. current_node) then
     out[$] = 1;    /* DELETE REVERSE */

endif;
                          /* ELIMINATE UNNECASSARY EDGES */
if  (tail[$] .eq. prev)  then
    out[$] = 1;
```

```
endif;

if   (head[$] .eq. current_node) then
    out[$] = 1;
endif;

i = i + 1;

until i  .eq. num_node                    /* END REPEAT UNTIL */
endloop;

/************ LAST NODE ADDED TO START NODE ***************/

if (tail[$] .eq. current_node) .and. (head[$] .eq. start_node) then
  tour[$] = 1;              /* get the last leg */
endif;

print tail[$] head[$] weight[$] tour[$] in graph[$];
end;
```

APPENDIX D
ASC Tables

ASC Program Format:

```
main program_name
define constants
define variables
variable declarations
association declarations
body
end;
```

Reserved Words:

a/an	endallocate	logical	return
allocate	endasm	loop	scalar
any	endfor	mindex	scin
andbody	endget	maxdex	sc_perform
andif	endif	minval	scot
andfor	endifany	maxval	setscope
andthen	endloop	msg	stack
asmcode	endnext	next	stop
associate	endrecursewhile	nany	subroutine
call	endsetscope	nthdex	the
count	endstack	nthval	their
define	for	nxtcd	them
deflog	fstcd	parallel	then
defvar	get	pa_perform	trncd
during	if	perform	trnacd
else	ifany	prevcd	until
elsenany	in	print	while
end	include	procedure	with
endandbody	index	real	#include
andandfor	int	recursewhile	
endandif	its	release	
endany			

```
Operators:

 Relational Operators:
   Less than                .lt.   <
   Greater than             .gt.   >
   Less than or equal       .le.   <=
   Greater than or equal    .ge.   >=
   Equal                    .eq.   =
   Not equal                .ne.   !=

 Logical Operators:
   Not                      .not.  !
   Or                       .or.   ||
   And                      .and.  &&
   Exclusive Or             .xor.  ^^

 Arithmetic Operators:
   Negation                  -
   Addition                  +
   Subtraction               -
   Multiplication            *
   Division                  /

Functions:

  fstcd(code)            - first structure code (left most child)
  nxtcd(code)            - next structure code (right sibling)
  prvcd(code)            - previous structure code (left sibling)
  trncd(code)            - truncate structure code (parent)
  trnacd(code)           - truncate all structure code (root)
  maxdex(field)          - index to maximum value of a field
  mindex(field)          - index to minimum value of a field
  maxval(field)          - maximum value of a field
  minval(field)          - minimum value of a field
  nthval(field,n)        - nth value of a field
  nthdex(field,n)        - nth index of a field
  scin(atom,sc,bi)       - structure code input (4 bit)
  scin8(atom,sc,bi)      - structure code input (8 bit)
  scinl(atom,sc,bi)      - structure code input (list)
  scinp(atom,sc,bi,pred,head_flag,tid,pred_flag)
                         - structure code input (Prolog)
  scot(atom,sc,bi)       - structure code output (4 bit)
  scot8(atom,sc,bi)      - structure code output (8 bit)
  scotl(atom,sc,bi)      - structure code output (list)
  scotp(atom,sc,bi)      - structure code output (Prolog)
  scst(atom,sc,bi,list)  - structure code constant (4 bit)
  scst8(atom,sc,bi,list) - structure code constant (8 bit)
  scstl(atom,sc,bi,list) - structure code constant (list)
  scstp(atom,sc,bi,pred,head_flag,tid,pred_flag,list)
                         - structure code constant (Prolog)
```

ASC Statements:

```
 int
 char
 logical
 real     scalar   variable(:width)([,dimension,...]), .. ]
 card     parallel variable(:width)[$(,dimension,...)], ..];
 bin
 hex
 oct


allocate index_variable in association_name;
  statement_block
endallocate index_variable;

allocate n index_variable contiguously in association[$];

any conditional_expression
  statement_block1
(elsenany
  statement_block2)
endany;

associate   item1,item2,... with association_name;

call subroutine called_field1<calling_field1 ...
                      calling_fieldn>called_fieldn;

deflog "("identifier, boolean_constant")";


defvar "("identifier,  decimal_constant               ")";
                       variable(  + decimal_constant)
                                  -


define "("identifier, constant")";

for index_variable in parallel_conditional_expression
  statement_block1
(elsenany
  statement_block2)
endfor index_variable;
```

```
get index_variable in parallel_conditional_expression
  statement_block1
(elsenany
  statement_block2)
endget index_variable;
if conditional_expression then
  statement_block1 ( else      statement_block2)
                     elsenany
endif;

(first
  statement_block)
loop
  until scalar_expression
  until nany parallel_expression
endloop;

msg (handle) delimitertextdelimiter variable_list;

next index_variable in parallel_variable
  statement_block1
(elsenany
  statement_block2)
endnext index_variable;

print  (handle) itema[$]  ...  itemn[$] in
                         logical_parallel_expression;

read   (handle) itema[$]  ...  itemn[$] in association[$];

read   (handle) item[$] ... contiguously in association[$];

readnl (handle) itema[$]  ...  itemn[$] in association[$];

release conditional_expression from association_name;

reread (handle) itema[$]  ...  itemn[$] in association[$];

setscope logical_parallel_expression
  statement_block
endsetscope;

while index_variable in parallel_conditional_expression
  statement_block
endwhile index_variable;
```