Burroughs

BSP

IMPLEMENTATION OF FORTRAN

BSP

BURROUGHS SCIENTIFIC PROCESSOR

IMPLEMENTATION OF FORTRAN

2. BSP INSTRUCTIONS

In order to process FORTRAN programs efficiently, the basic machine instructions produced by the BSP FORTRAN compiler closely resemble FORTRAN assignment statements whose operands are arrays of values. For example, the BSP FORTRAN compiler generates one array operation to execute the following FORTRAN statements:

```
      DO 10 I = 1, 100
         DO 10 J = 2, 100, 2
10          A (I, J) = ((B (I, J) + B (I, J-1)) / 2
```

The instruction that executes these FORTRAN statements is specified by the following:

1. The number of operands, in this case, two distinct subarrays of the array B and the single value 2,

2. The operators involved, in this case, addition and division,

3. A description of each of the operands and a description of the result array A.

The parallel processor that executes the instruction just described consists of parallel memory and 16 arithmetic elements that operate in lock step, i.e., all executing the same arithmetic operation at the same time on different data. The hardware of the parallel processor control unit will break up the required number of computations into groups of 16 and cause the parallel processor to continue processing until all of the specified computations have been completed.

3. TERMINOLOGY

In order to describe the BSP FORTRAN, it is necessary to define a few terms.

1. An ARRAY is a collection of values. A SCALAR is a single value.

2. The DIMENSIONALITY of an array is the number of its dimensions. This property is always determined during compilation from a type statement, DIMENSION statement, COMMON statement, ARRAY statement, or by special subscript forms used in an expression. A BSP FORTRAN array may have up to 16 dimensions.

3. A VECTOR is an array with one dimension. A MATRIX is an array with two dimensions.

4. For each dimension of an array, there is an EXTENT, which is the number of elements along that dimension. An extent may change during execution of a program. The EXTENT LIST is the sequence of the array's extents.

5. The SIZE of a real array is the product of its extents, which is the number of elements in the array. The size of a double precision or complex array is twice the number of elements in the array. The maximum size of an array in a BSP program is 8,388,607 = 2 ** 23 - 1.

   Example:

   ```
   REAL A (2, 0:4) -3:3)
   ```

   The array A has dimensionality 3. Its extent list is (2, 5, 7) and its size is 70.

   Example:

   ```
   SUBROUTINE S (B, N)
   REAL B (N, 0:N)
   ```

   The array B has dimensionality 2. During execution of subroutine S invoked by the statement:

   ```
   CALL S (X, 2)
   ```

it has extent list (2, 3) and size 6. During execution of subroutine S invoked by the statement:

```
CALL S (Y, 3)
```

it has extent list (3, 4) and size 12.

6. Two arrays are CONGRUENT if they have identical extent lists (which means they have the same dimensionality). In BSP FORTRAN, there are many contexts requiring congruence of arrays. For example, if A, B, and C are arrays, the statement:

```
A = B + C
```

is allowed only if A, B, and C are all congruent.

In the following dicussions, each scalar is considered to be congruent to any array, so that if A is any array, the statements:

```
A = A + 1
A = 2 * A
```

are permitted.

7. In BSP FORTRAN there are three kinds of arrays: the actual arrays and dummy arrays of standard FORTRAN and IDENTIFIED arrays. Some arrays are identified EXPLICITLY using the ARRAY statement and others are identified IMPLICITLY in expressions. Each of these kinds of arrays appears in examples given later.

4. THE BSP FORTRAN LANGUAGE PROCESSOR

The FORTRAN language system on the BSP incorporates two very special features. One is a set of language extensions specially designed for array processing. The other is a component of the compiler called a "vectorizer", whose purpose is to recognize constructs in standard serial FORTRAN programs that can be executed in parallel. The language extensions will be discussed first followed by a description of some of the features of the vectorizer.

LANGUAGE EXTENSIONS

The BSP is designed to execute large scientific programs characterized by the processing of arrays. FORTRAN is the most widely used language for such programming applications; however, standard FORTRAN does not have facilities for expressing operations that treat arrays as entities. Several features designed for array processing have been added to BSP FORTRAN for two reasons:

1. In the expression of algorithms, it is convenient to be able to treat arrays and components of an array, such as rows, columns, and diagonals, as a single entity.

2. In some circumstances, no compiler could determine that an operation could be executed in parallel, but the programmer has this knowledge and needs to express the knowledge in the program.

The BSP FORTRAN array processing extensions provide:

1. A notation for naming and referencing arrays and sections of arrays,

2. Intrinsic and user-defined functions for manipulating arrays,

3. Logical masks to extract or selectively operate on irregular sections of arrays,

4. Facilities for dynamic array expansion, compression, and merging.

The ARRAY Statement

The ARRAY statement is a BSP FORTRAN extension that allows either a sparse or a dense section of an actual or dummy array to be identified explicitly by giving it a name.

```
      REAL A (100, 100)
      ARRAY ROW 2 (J = 1:100) = A ( 2, J)
     +,     DIAG (I = 1:100) = A (I, I)
     +,     TRANS (I = 1:100, J = 1:100) = A, (J, I)
```

The REAL statement declares A to be a matrix with 10,000 elements. The ARRAY statement explicitly identifies three arrays:

1. ROW 2, a vector consisting of the second row of A,

2. DIAG, a vector consisting of the main diagonal of A,

3. TRANS, a matrix that is the transpose of A.

No storage is allocated for any identified array. Each identified array is simply a name for a collection of elements belonging to an actual or dummy array.

Although the dimensionality of an identified array is determined at compile time, its extent may vary dynamically.

Example:

```
      REAL A (100, 100)
      ARRAY ROW N (J = N:100) = A (N, J)
```

When N = 1, ROW N is the entire first row of A with extent 100. When N = 2, ROW N consists of elements 2 to 100 of row 2 with extent 99. When N = 100, ROW N consists of the single element A (100, 100). The statements:

```
          DO 28 N = 1, 100
     28      READ /, ROW N
```

would read the upper right triangular portion of A, including the diagonal.

In the example array declarations just given, I and J are dummy variables, but N refers to the value of the program variable N during execution. Changing the extent of the vector ROW N can be achieved by varying the value of N.

Identified arrays must not be used in type, DIMENSION, COMMON, EQUIVALENCE, or DATA statements, consistent with the design goal of isolating BSP extensions in new statements as much as possible. This allows parts of programs that use these features to be identified readily and simplifies their implementation.

Implicitly Identified Arrays

An array may be identified implicitly by the appearance of an array name followed by special subscript forms. The named array may be an actual array, a dummy array, or an array explicitly identified in an ARRAY statement. The special subscript expressions are:

1. * or -* indicating all of the elements in a particular dimension. The expression -* indicates that the elements are to be in the reverse of the usual order.

2. e1:e2 or e1:e2:e3, where e1, e2, and e3 are expressions, and e1 indicates the first element, e2 indicates the last element, and e3 is a skip distance. If e3 is omitted, the skip distance is one, indicating all of the elements from e1 to e2, inclusive.

3. A vector indicating an arbitrary collection of subscripts in one dimension.

Example:

```
REAL A (3, 3), U (2), V (4)
DATA U /3, 1/, V /2, 4, 1, 4/
```

1. A (*, 3) is the third column of A.

2. A (-*, *) is the reflection of A about the second row, i.e., is the matrix:

```
A (3, 1)   A (3, 2)   A (3, 3)
A (2, 1)   A (2, 2)   A (2, 3)
A (1, 1)   A (1, 2)   A (1, 3)
```

3. A (2, 1:3) is the second row of A.

4. A (*, 1:3:2) is the matrix consisting of the first and third columns of A.

5. A (1:N, 1:N) is the upper left hand square submatrix of A of size N X N.

6. V (U) is the vector:

   V (3) V (1) with values 1 2

7. A (U, V) is the matrix:

```
A (3, 2)   A (3, 4)   A (3, 1)   A (3, 4)
A (1, 2)   A (1, 4)   A (1, 1)   A (1, 4)
```

Array Operations

The arithmetic operators +, -, *, /, and **, the relational operators .GT., .GE., .EQ., .NE., .LE., and .LT., and the logical operators .NOT., .OR., and .AND. may be applied to operands that are arrays. For any binary operator, the two operands must be congruent, that is, have identical extent lists.

These operators are applied element by element to the operands.

Example:

```
    I 0  2 I       I 1  3 I
A = I       I  B = I       I
    I 4  6 I       I 2  4 I

        I 1  5 I
A + B = I       I
        I 6 10 I

           I .TRUE.     .TRUE.  I
A .LT. B = I                    I
           I .FALSE.    .FALSE. I
```

Each array operation executed by the parallel processor may have associated with it a LOGICAL ARRAY that serves as a mask, determining which positions are to be affected by the computation.

Example: The FORTRAN statements:

```
      DO 10 I = 1, 500
10    IF (A (I) .GE. 0) B (I) = B (I) + A (I)
```

cause a logical array with value true in the positions where A (I) .GE. 0 and false elsewhere to be constructed. The array assignment:

```
      B (1:500) = B (1:500) + A (1:500)
```

will be executed under control of the logical array, which causes B (I) to be changed only for those values of I where the logical array is true.

WHERE Constructs

The BSP FORTRAN WHERE, WHERE-DO, OTHERWISE, and END WHERE statements allow the programmer to indicate assignment statements that may be executed under control of a logical array.

Example:

```
    WHERE (A .GE. 0) B = B + A
```

is equivalent to the DO loop given previously provided A and B are vectors with subscripts ranging from 1 to 500.

Example:

```
REAL A (0:100, 0:100), B (0:100, 0:100)
WHERE (A 91:100, *) .GE. B (0:99, *) - 1) DO
   B (0:99, *) = A (1:100, *)
OTHERWISE
   B (0:99, *) = B (0:99, *) - 1
END WHERE
```

This example illustrates further use of implicitly identified arrays. The logical array generated by the WHERE statement must be congruent to each of the arrays in the assignment statements in the WHERE construct. In this case each has extent list (100, 101).

Only assignment statements and other WHERE constructs may appear within a WHERE construct. The assignment statements within a WHERE construct are evaluated according to the following rules:

1. If the right side contains no functions other than elemental intrinsic functions (described later), it will be evaluated only for those positions in the array indicated by a true in the logical array: otherwise, it is evaluated for all positions.

2. The array of results is stored using the logical array as a mask so that only positions corresponding to true are affected.

3. If the assignment statement is part of the OTHERWISE portion of a WHERE construct, the complement of the specified logical array is used.

Example:

```
WHERE (C .NE. 0) A = B / C
```

According to rule 2 above, the division will be performed only for those values of C which are not zero.

PACK AND UNPACK STATEMENTS

The PACK and UNPACK statements provide facilities for expanding, compressing, and merging arrays.

Example:

Assume A, B, C, and D are 3 X 3 arrays, V1 and V2 are vectors, and A .GE. B is the logical array:

```
F  F  F
F  T  F
T  T  T
```

Then,

```
PACK WHERE (A .GE. B) V1 = C, OTHERWISE V2 = D
```

causes elements of C in positions corresponding to true in the logical array A .GE. B to be assigned to V1 in the order in which they are stored using the usual FORTRAN storage mechanism.

```
V1 (1) = C (3, 1)        C (1, 1)  C (1, 2)  C (1, 3)
V1 (2) = C (2, 2)        C (2, 1)  C (2, 2)  C (2, 3)
V1 (3) = C (3, 2)        C (3, 1)  C (3, 2)  C (3, 3)
V1 (4) = C (3, 3)
```

The elements of D in the remaining positions are stored in V2.

```
V2 (1) = C (1, 1)        C (1, 1)  C (1, 2)  C (1, 3)
V2 (2) = C (2, 1)        C (2, 1)  C (2, 2)  C (2, 3)
V2 (3) = C (1, 2)        C (3, 1)  C (3, 2)  C (3, 3)
V2 (4) = C (1, 3)
V2 (5) - C (2, 3)
```

The UNPACK statement performs the inverse operation of the PACK statement.

Example:

```
UNPACK WHERE (A .EQ. 0) A = V
```

inserts the elements of V into the zero positions of the array A.

CONTROL STATEMENTS

To allow the programmer to express algorithms in FORTRAN in a more natural manner, some extensions to the usual control structures have been implemented in BSP FORTRAN.

The IF-THEN-ELSE construct of the proposed new standard FORTRAN is in BSP FORTRAN.

Example:

```
IF  (K .EQ. 0) THEN
    A = SUM (B)
ELSE IF (K .EQ. 1) THEN
    A = 0
ELSE
    Y = Q + T
END IF
```

A new form of the DO statement that does not require a label has been implemented. Also, by the use of a compiler option, the programmer may have loops interpreted as they are in current Burroughs FORTRAN or according to the interpretation of DO loops in the proposed new FORTRAN standard.

Example:

```
DO K = 1, N
   A (K) = 0
   B (K) = 0
END DO
```

THE EXCHANGE ASSIGNMENT

BSP FORTRAN contains an exchange statement of the form:

```
V1 == V2
```

where V1 and V2 are both scalars or are congruent arrays. This allows the programmer to access the hardware feature that exchanges values without using temporary storage.

ASYNCHRONOUS INPUT AND OUTPUT

In scientific applications, it is often important that the programmer have control of the input and output operations. In BSP FORTRAN, it is possible to read and write values to and from an array without any buffering by naming the array in a DIRECT statement. Instead of waiting for the I/O operation to complete, processing will continue with the next executable statement of the program, so that program execution will occur concurrently with the input or output operation.

Processing will wait just prior to the next statement that requires completion of the I/O operation, so the programmer does not have to be concerned with coordination of I/O operations and computations.

Example:

```
      REAL A (1000), Z (100, 1000)
      DIRECT A
      READ (7) A
      Z = 0
C     EXECUTION WAITS HERE UNTIL READ IS COMPLETE
      Z (1, *) - A + 1
```

CHECKPOINT AND RESTART

The CHECKPOINT statement directs the Master Control Program to save the state of a running program at that point in time so that it can be restarted at a later time. Such a facility is essential for programs that require hours of computation time. The programmer may specify the system manager storage device on which the program values are placed.

INTRINSIC FUNCTIONS

In addition to the usual collection of intrinsic functions for scalar computations, there are two classes of intrinsic functions for arrays.

1. The ELEMENTAL INTRINSIC FUNCTIONS apply a computation to each element of an array. Many of the scalar intrinsic functions have been extended to elemental intrinsic functions for arrays.

   Example:

   ```
   A (1: 100) = SIN (B (1:100))
   ```

   is equivalent to:

   ```
   DO I = 1, 100
      A (I) = SIN (B (II)
   END DO
   ```

2. In addition, there are array intrinsic functions whose values are not determined on an element-by-element basis. Some examples are:

   MMULT (A, B) that forms the product of matrices A and B,

   DOTPRD (A, B) that forms the dot product of vectors A and B,

   MAXVAL (A) that finds the largest value of the array A,

   PROD (A) that computes the product of all elements of the array A,

   IDENT (N) whose value is an N X N identity matrix,

   SIZE (A) whose value is the number of elements of the array A.

SOURCE REGENERATION

There is frequently a requirement to maintain a standard FORTRAN version of a program in order to be able to transport it to other systems. Upon request, the BSP FORTRAN compiler will produce equivalent standard FORTRAN for all array extensions used in a program. This allows the programmer to produce the most efficient program possible for the BSP and still have an equivalent standard FORTRAN version of the program.

SOLVING A LINEAR SYSTEM

Many of the features that have been discussed are illustrated by two versions of a subroutine that solves the linear system

$$A X = B$$

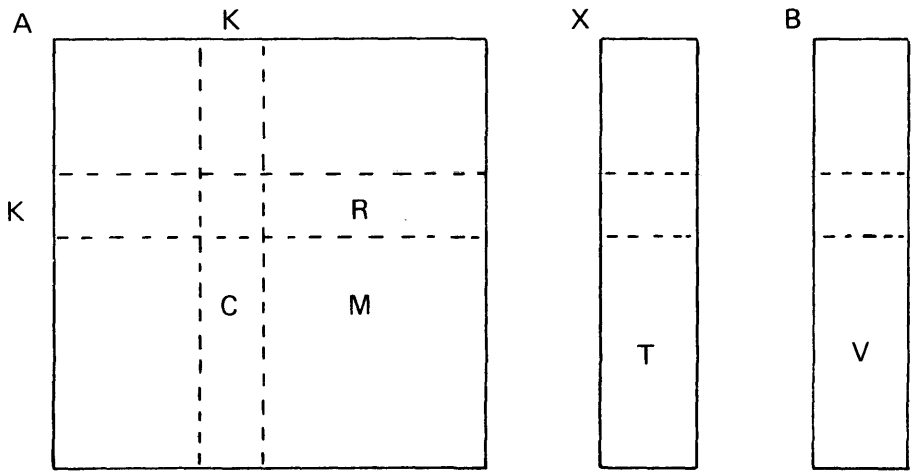
using Gauss elimination.

This computation is divided in two major steps, Gauss reduction producing A' and B' such that A' has zeroes below the main diagonal and

A' X = B'

and a back substitution step.

One scheme is to name certain subarrays of A and B as shown in the following diagram.



Implementation of FORTRAN on the BSP

so that:

```
R = A (K, K+1:N)
C = A (K+1:N, K)
M = A (K+1:N, K+1:N)
T = X (K+1:N)
V = B (K+1:N)

SUBROUTINE GAUSS (A, B, X, N)
REAL A (N, N), B (N), X (N)
ARRAY C (I = 1:N-K, J = 1) = A (I+K, K)
+,     R (J = 1, I = 1:N-K) = A (K, I+K)
+,     M (I = 1:N-K, J = 1:N-K) = A (I+K, J+K)
+,     V (I = 1:N-K, J = 1) = B (I+K)
+,     T (I = 1:N-K)) = X (I+K)
+,     RR (I = 1:N-K) = R (1, I)
 DO K - 1, N-1
    M = M - MMULT (C, R) / A (K, K)
    V = V = C * B (K) / A (K, K)
 END DO
 X (N) = B (N) / A (N, N)
 DO K = N-1, 1, -1
    X (K) - (B (K) - DOT PRD (RR, T)) / A (K, K)
 END DO
 END
```

A second method is to identify all subarrays implicitly in expressions as shown in the following equivalent program.

```
      SUBROUTINE GAUSS (A, B, X, N)
      REAL A (N, N), B (N), X (N)
      DO K = 1, N-1
         A (K+1:N, K+1:N) = A (K+1:N, K+1:N)
     +      - MMULT (A (K+1:N, K:K), A (K:K, K+1:N)) / A (K, K)
         B (K+1:N) - B (K+1:N) - A (K+1:N, K:K) * B (K) / A (K, K)
      END DO
      X (N) = B (N) / A (N, N)
      DO K = N-1, 1, -1
         X (K) = (B (K) - DOT PRD (A (K, K+1:N), X (K+1:N))) / A (K, K)
      END DO
      END
```

OPTIMIZATION

The BSP FORTRAN compiler has two distinct phases of optimization. The first phase is a machine-independent optimization that does such things as eliminate unreachable code, perform algebraic simplifications, and propagate constants. These optimizations are based primarily on a program graph representing the flow of control. This phase may or may not be used during compilation at the option of the programmer.

The second phase of optimization is more machine-dependent, performing such functions as scheduling vector and scalar operations to execute concurrently, allocating registers efficiently, eliminating common subexpressions, performing loop fusion, and performing operator strength reduction.

THE VECTORIZER

It is possible to access most of the power of the parallel processor without using any of the language extensions by using the vectorizer of the BSP FORTRAN compiler. Thus, existing FORTRAN programs can be run efficiently without the necessity to make major changes.

The vectorizer is a component of the BSP FORTRAN compiler whose function is to recognize FORTRAN constructs that can be executed in parallel. Like the first phase of optimization, it can be invoked selectively during compilation at the discretion of the programmer.

The BSP FORTRAN vectorizer advances the state-of-the-art in this area. It recognizes parallelism in more constructs than any other vectorizing compiler available. The vectorizer does not implement ad hoc pattern matching techniques to recognize parallelism. It does a sophisticated analysis of data dependencies and uses an algorithm to determine if vectorization is possible. Some of the following examples illustrate the concept of data dependence.

Two important classes of constructs the BSP vectorizer will handle that others will not are those containing IF statements and those containing recurrences. A typical recurrence is:

```
      A (0) = X
      DD I = 1, N
         A (I) = A (I-1) * B (I) + C (I)
      END DO
```

This construct appears to require serial execution because each value A (I) depends upon the value A (I-1) computed during the previous execution of the loop. However, there is a way to take advantage of the repetitive nature of the arithmetic operations involved to perform calculations concurrently and achieve execution speeds approximately five times that of a serial computation. The BSP vectorizer will recognize recurrences such as these and produce code that will achieve a five-fold increase in execution speed.

The following examples illustrate the kinds of constructs recognizable by the vectorizer and show the equivalent computations performed utilizing the array constructs discussed previously.

```
  DO 1 I = 1, N
1    A (I) = B (I) + C (I)
           II
           II
           \/
  A (1:N) = B (1:N) + C (1:N)

  DO 2 I = 1, N-1
2    IF (M .GT. 3) A (I) = A (I+1) + C (I)
           II
           II
           \/
  IF (M .GT. 3) A (1:N-1) = A (2:N) + C (1:N-1)

  DO 3 I = 1, N
     IF (A (I) .GE. B (I)) THEN
        C (I) = E (I) + 2
        A (I) = 0
     ELSE
        C (I) = F (I)
3 CONTINUE
           II
           II
           \/
  LOGICAL L (N)
  L = A (1:N) .GE. B (1:N)
  WHERE (L) DO
     C (1:N) = E (1:N) + 2
     A (1:N) = 0
  OTHERWISE
     C (1:N) = F (1:N)
  ENDWHERE

  DO 4 I = 2, 100
     A (I) = B (I) + 1
     C (I) = D (I-2) + E (I)
     D (I) = C (I-1) + A (I)
4    F (I) = D (I) + 2
           II
           II
           \/
```

```
      A (2:1:00) = B (2:100) + 1
      DO I = 2, 100
         C (I) = D (I-2) + E (I)
         D (I) = C (I-1) + A (I)
      END DO
      F (2:100) = D (2:100) + 2
```

In this last example, the values of D depend on values of A and cannot be computed before the first statement in the loop. Similarly, the values of F depend on the values of D and must be computed after the values of D are computed. Values of C depend on values of D and values of D depend on values of C in such a way that the computations for these two arrays are performed serially.

```
      DO 5 I = 1, N
         A (I) = B (I) + C (I)
         C (I) = B (I-1)
5        B (I) = 2 * A (I+1)
               II
               II
               \/
      REAL TEMP (N)
      TEMP = A (2:N+1)
      A (1:N) = B (1:N) + C (1:N)
      B (1:N) = 2 * TEMP
      C (1:N) = B (0:N-1)
```

In this case, old values of A are needed to compute B, so all new values of A cannot be computed without saving the old ones. Also, new values of B must be used to compute values of C. This works correctly if all values of B are computed prior to computing any values of C. The vectorizer will save the old values of A in TEMP and exchange the last two statements in order to do parallel computations.

```
      DO 6 I = 1, M
      DO 6 J = 1, N
      DO 6 K = 1, L
         A (I, J+1, K) = B (I, J-1, K)
6        B (I, J, K) = A (I, J, K)
               II
               II
               \/
      DO J = 1, N
         A (1:M, J+1, 1:L) = B (1:M, J-1, 1:L)
         B (1:M, J, 1:L) = A (1:M, J, 1:L)
      END DO
```

Note that the DO J loop has become the outer loop.

When the vectorizer recognizes a construct that can be executed in parallel, the programmer is given this information. This provides a learning tool in that the programmer can examine the output of the vectorizer and gain insight into the ways that algorithms can be expressed using parallel constructs.

5. SUMMARY

The Burroughs Scientific Processor FORTRAN language processing system is designed to provide a means by which the programmer can access the high speed of the BSP and express the algorithms to be executed in FORTRAN, the language most commonly used for scientific applications.

The language extensions, particularly those provided for array processing, allow algorithms to be written in a notation that is more natural for the expression of the computations to be performed and allow parallel operations to be expressed directly.

On the other hand, if it is desirable to have programs that are expressed in standard FORTRAN, the vectorizer of the FORTRAN compiler will detect many constructs that can be executed in parallel and generate the appropriate operators for the parallel processor.