An Empirical Study of Fortran Programs for Parallelizing Compilers

Zhiyu Shen
Zhiyuan Li
Pen-Chung Yew

Abstract

In this paper, we report some results from an empirical study on program characteristics that are important to parallelizing compiler writers, especially in the area of data dependence analysis and program transformations. The state of the art in data dependence analysis and some parallel execution techniques are also examined.

The major findings include: (1) Many subscripts contain symbolic terms with unknown values. A few methods to determine their values at compile time are evaluated. (2) Array references with *coupled subscripts* appear quite frequently. These subscripts must be handled simultaneously in a dependence test, rather than being handled separately as in current test algorithms. (3) Nonzero coefficients of loop indices in most subscripts are found to be simple: they are either 1 or -1. It allows an exact real-valued test to be as accurate as an exact integer-valued test for 1-dimensional or 2-dimensional arrays. (4) Dependences with uncertain distance are found to be rather common. One of the main reasons is the frequent appearence of symbolic terms with unknown values. This might have a significant impact on current techniques of data synchronization, loop scheduling and partitioning.

Index Terms --- Array subscripts, data dependences, Fortran packages, vectorization, parallelization, program statistics

AN EMPIRICAL STUDY OF FORTRAN PROGRAMS FOR PARALLELIZING COMPILERS

1. Introduction

The key to the success of a parallelizing compiler is to have accurate data dependence information on all of the statements in a program. We would like to identify all of the independent variable references and statements in a program, so they can be executed independently (i.e. in parallel). Several algorithms have been proposed and used quite successfully in many parallelizing compilers [1], [2], [3], [4], [28], [11]. Nonetheless, their ability is still limited to relatively simple subscripts. This paper identifies three factors that could potentially weaken the results of current algorithms: (1) symbolic terms with unknown values; (2) coupled subscripts; (3) nonzero and non-unity coefficients of loop indices. We discuss the effects of these factors and present some measured results on real programs. We also report some characteristics of data dependences found in real programs. The state of the art in data dependence analysis and various parallel execution techniques can be examined in light of such information. The information can also help to indicate the direction of further improvement in those areas. We begin with a brief review of some basic concepts in data dependence analysis and their effects on parallel execution of programs.

2. Data Dependences

There are three types of data dependences [16]. If a statement, S1, uses the result of another statement, S2, then S1 is *flow dependent* on S2. If S1 can store its result only after S2 fetches the old data stored in that location, then S1 is *antidependent* on S2. If S1 overwrites the result of S2, then S1 is *output dependent* on S2. Data dependences dictate execution precedence among statements. The following DO loop is an example.

Example 2.1

```
DO  I = 3, N
  . = A(I-1)          (S1)
    .
  A(I) = .            (S2)
    .
  A(I-2) = .          (S3)
  ...
END
```

In this loop, S1 is flow dependent on S2 because it reads the result of S2 (from the previous iteration). Due to the dependence, the execution of S1 in iteration *I* must follow the execution of S2 in iteration *I-1*. S3 is antidependent on S1 and the execution of S3 in iteration *I* must follow the execution of S1 in iteration *I-1*. S3 is output dependent on S2 and the execution of S3 in iteration *I* must follow the execution of S2

in iteration *I-2*. Execution precedence may also be affected by *control dependence*. For example, an IF statement decides which branch to take. Hence, the statements in the branches cannot be executed before the decision is made. Control dependence is not studied in this paper.

In order to speed up program execution on a parallel machine, a parallelizing compiler can be used to discover independent statements which can be executed in parallel. DO loops are usually the most important source of such parallelism because they usually contain most of the computation in a program. If there are no dependences among the statements in a DO loop, or the dependences are restricted within the iteration boundaries, different iterations of the loop can be executed concurrently.

Example 2.2

```
DO  I = 1, N
  A(I) = B(I) * (C(I) + C(I-1))          (S1)
  D(I) = A(I) / F(I)                     (S2)
END
```

In the above example, although S2 is flow dependent on S1, the dependence is restricted within each iteration (i.e. there is no cross-iteration dependences). Therefore, all of the iterations in the loop can be executed in parallel. This example shows that a parallelizing compiler not only needs to determine whether data dependence exists, but also needs to analyze whether such dependence prohibits loop parallelization. Many transformation techniques (e.g. *loop interchange* [28] and the detection of *Doacross loops* [9]) require even more information about dependences, such as *dependence distances* and *dependence direction vectors*.

If a data dependence occurs across several iterations of a loop, the distance is called its *dependence distance* (with respect to that loop). All of the data dependences in Example 2.1 have constant distances. For instance, the output dependence between S2 and S3 has a distance of 2. If a dependence occurs within the same iteration, the dependence distance is 0. Note that statements may be nested in a number of loops. Their dependences may have different distances with respect to different loops.

Example 2.3

```
DO  I = 2, N
  DO  J = 2, N
    A(I,J) = .                  (S1)
       .
    . = A(I-1,J+1)              (S2)
       .
  END
END
```

In the example above, the flow dependence between S1 and S2 has a distance of 1 with respect to the *I* loop and a distance of -1 with respect to the *J* loop. A data dependence distance may not always be constant. Consider the following example.

Example 2.4

```
DO  I = 1, N
  A(I) = .                      (S1)
       .
  DO  K = 1, I
       .
  . = A(K)                      (S2)
       .
  END
END
```

The dependence between S1 and S2 has a variable distance. If we use S1<i> to denote the instance of S1 in iteration *i* of the *I* loop and use S2<i,k> to denote the instance of S2 in iteration *i* of the *I* loop and iteration *k* of the *K* loop, then S1<1> should be executed before S2<1,1>, S2<2,1>, ..., S2<N,1>; S1<2> should be executed before S2<2,2>, S2<3,2>, ..., S2<N,2>, and so on. We shall give more examples on variable dependence distances in Section 4.2.

A dependence direction vector [28] contains several elements, each corresponding to one of the enclosing loops. Each element of a dependence direction vector is called a *dependence direction*. To simplify the discussion, we take as an example a nest of two loops, where the outer loop has index variable *I* and the inner loop has index variable *J*. Suppose as the result of a data dependence between statements S1 and S2, the execution of S1<i_1,j_1> must precede that of S2<i_2, j_2>. The dependence direction for the *J* loop is "<", "=", or ">" depending on if we have $j_1 < j_2$, $j_1 = j_2$, or $j_1 > j_2$ respectively. The dependence direction for the *I* loop is determined similarly. Note, however, that since the *I* loop is the outmost loop, its dependence direction cannot be ">". In Example 2.3, the flow dependence between S1 and S2 has a dependence direction vector (<, >). In Example 2.4, the flow dependence between S1 and S2 has a dependence direction vector (<) and a dependence direction vector (=). The two vectors can sometimes be combined, written as (<=).

Obviously, dependence direction vectors can be used to describe general data dependences, although they are not as precise as the latter. For many important loop parallelization and transformation techniques, dependence direction vectors usually provide sufficient information. Nonetheless, dependence distances are important to techniques such as data synchronization [25], [29], *loop partitioning* [21], [22], [24], *processor allocation* [9], [23], and *processor self-scheduling* [12], [26]. As a matter of fact, most data synchronization and loop partitioning schemes assume subscripts to have a simple form of i+c where i is a loop index and c is a constant. Further, data dependences are assumed to have *constant distances*. If dependences do not have constant distances, existing schemes either fail or suffer from loss of run-time efficiency.

2.2 The Experiment

This empirical study evaluates the complexity of array subscripts and data dependences in real programs. Our measurements are done on a dozen Fortran numerical packages (Table 1) which have a total of more than a thousand routines and over a hundred thousand lines of code. This sampling is a mix of library packages (Linpack, Eispack, Itpack, MSL, Fishpak) and working programs (SPICE, SMPL, etc.). Library packages are important because their routines are called very frequently in user's programs for scientific and engineering computing. On the other hand, the working programs may better reflect the array reference behavior in user-written programs. Further study may be needed to distinguish array referencing behavior in library routines and in working programs. Our code for measurement is embedded in Parafrase [13], which is a restructuring compiler developed at the Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign.

Our study differs from previous related works in that we directly measure the variable references and data dependences, whereas previous works (e.g. [22], [25], [15]) focused on counting the number of statements that can be executed in parallel. Our work is a preliminary attempt to examine closely the effects of array subscript patterns on some important techniques used at compile time and run time for efficient parallel execution. We have yet to relate our results to previous results which were mostly at higher levels, e.g. the statement level. The relationship between those different levels is certainly an important subject for further study.

Our analysis is presented as follows. In Section 3, we examine the form of array subscripts. We cover three factors that can affect data dependence analysis: linearity, coupled subscripts, and coefficients of loop indices. In Section 4, we show the effectiveness of several well-known data dependence test algorithms. To get some idea of how often a pair of array references are detected to be independent by these algorithms, we recorded the number of independent array reference pairs detected by each algorithm. We also report statistics on data dependence distances. Finally, we make some concluding remarks in Section 5.

3. Subscripts in Array References

3.1 Linearity of array subscripts

Consider an m-dimensional array reference in a loop nesting with n loops[1] indexed by I_1, I_2, ..., I_n. Normally the reference has the following form:

$$\text{Arrayname}(Exp_1, Exp_2, \ldots, Exp_m)$$

where: Exp_i is a subscript expression, $1 \le i \le m$.

A subscript expression has the following form:

$$a_1 \times I_1 + a_2 \times I_2 + \ldots + a_n \times I_n + b$$

where: I_j is an index variable, $1 \le j \le n$, a_j is the coefficient of I_j, $1 \le j \le n$. b is the remaining part of the subscript expression that does not contain any index variables. Note that, following the convention in mathematics, b may be called the constant term. However, in a nest of loops, it is possible that b may contain some unknown variables that are updated within the loop. For convenience, we call b the *zeroth* term.

The above subscript expression is *linear* if all of its coefficients and its zeroth term are integer *constants*. Otherwise, it is *nonlinear* because the subscript expressions may behave like a nonlinear function with respect to the loop indices. If all of the subscript expressions in a reference are linear, we say that the reference is linear. If some of the subscript expressions are nonlinear, the reference is partially linear. Finally, if none of the expressions is linear, the reference is nonlinear.

Virtually all current algorithms for data dependence tests operate on linear subscript expressions. Recently, some symbolic manipulation schemes are proposed for partially linear and nonlinear cases [18]. Several restructuring techniques can also be used to transform nonlinear subscripts into linear ones. The most notable ones are *expression forward substitution*, *induction variable substitution*, and *constant folding* (e.g., see [2], [14]). Table 2 gives an overview of the linearity of array subscripts (in the 12 numerical packages) after transformation by those techniques. We count only the references in loops.

From Table 2, we can see that only 8% of the array references have more than two dimensions (in column 2) and also that 53% of the references are linear (in column 3), 13% are partially linear (in column 4), and 34% are nonlinear (column 5).

The major reason for a subscript expression to be nonlinear is that it contains an unknown variable (i.e., a non-index variable with an unknown value) or an array element. Between the two, Table 3 shows that unknown variables are the major cause.

We found that quite a number of unknown variables are dummy parameters of subroutines or are related to dummy parameters. Some of them can assume a fixed value at run time. The value is normally set by a user before the program is run and is transferred from the main program to the subroutines. These variables usually specify the size of a matrix, the number of diagonals, and the number of vectors to be transformed. It has been a common practice to include user assertions to make the value of such parameters known to the compiler [8], [15]. As a matter of fact, the value of such parameters usually does not affect the outcome of a data dependence test. Often, providing the value mainly helps the data dependence test algorithms to eliminate the same symbolic terms in the subscripts and the loop bounds. Unknown symbolic terms may also be eliminated through interprocedural constant propagation [8]. Further, if data dependence tests could be extended to allow symbolic terms,

fixing the value would be unnecessary. For example, [1] presented suggestions about how to handle symbolic terms. We did not use interprocedural constant propagation because we analyzed procedures separately and did not write driver programs to call subroutines in the packages. We only examined the effect of user assertions on the linearity of subscripts. Nonetheless, since user assertions often provide the same results as interprocedural constant propagation, our result partly reflects the effect of interprocedural constant propagation. It is too time consuming to provide user assertions to more than one thousand routines. Instead, six packages were chosen and analyzed with user assertions. These packages are: Linpack, Eispack, Nasa, Baro, Itpack, and Old. Linpack and Eispack were chosen because user assertions were available from a previous experiment [15]. The rest of the packages were chosen randomly. Table 4 shows some details of the study. Without the help of user assertions, 47% of the one-dimensional array references and 45% of the two-dimensional array references were nonlinear. Using user assertions, only 28% of the one-dimensional array references and 15% of the two-dimensional array references remained nonlinear. Without user assertions, 27% of the two-dimensional array references were partially linear. Using user assertions, 25% of the two-dimensional array references were partially linear. Table 4 also shows the number of unknown variables found, including those in partially linear references.

For the remaining unknown symbolic terms, we examined the causes. One is that many nonlinear subscripts showed up in loops with subroutine calls or external function statements. These nonlinear subscripts cannot be transformed into linear subscripts using simple forward substitution techniques, unless the call effects of these statements are determined by interprocedural analysis. We studied 35 real-valued subroutines in the Linpack (there are another 35 complex-valued subroutines in Linpack which are almost identical [10]) using summary USE and MOD information to expose call effects [6]. The results on Linpack which are given in Table 5 suggest that nonlinear subscripts can be reduced considerably by using interprocedural analysis. Note that the number of unknown variables include those in partially linear references.

Besides unknown symbolic terms, the next most common reason for nonlinear subscripts is the presence of an array index which is indirectly an element of an array. The following example is from Linpack, where IPVT(*) is an integer vector of pivot indices.

```
DO  K = 1, N
  L = IPVT(K)
  T = Z(L)
  Z(L) = Z(K)
  Z(K) = T
  ...
END
```

There are other various minor reasons for nonlinear subscripts which are omitted here due to limited space.

3.2 Coupled subscripts

In this section, we study a phenomenon called *coupled subscripts* which demonstrates a weakness of current data dependence tests.

To test data dependence between a pair of array references, ideally all array dimensions should be considered simultaneously. However, most current algorithms test each dimension separately, because a single-dimension test is by far easier. Fortunately, this often suffices for discovering data independence. However, in cases where data *independence* could not be proven by testing each dimension separately, a data dependence has to be assumed. The main reason for those cases is *coupled subscripts* in which a loop index appears in more than one dimension. The following simple example is from Eispack:

```
DO  I = 2, UK
  MP = I - 1
  RM1(I,I+2) =
  RM1(MP,MP) =
END
```

In this example, there is no data dependence between the two references to RM1. But this could be detected only when both dimensions are considered simultaneously.

Of course, to measure how often coupled subscripts actually hurt a single dimension data dependence test requires testing all dimensions simultaneously to find genuine data dependences. A few methods are known to be very time consuming. Recently [19] proposed a new algorithm which is quite efficient. Using the new test on Eispack routines, data independence detection has improved by 10% over using single dimension tests. [7] discussed a different approach called, *linearization*, to dealing with coupled subscripts.

Here we only measure how often coupled subscripts occur in programs. As we shall discuss later, coupled subscripts are also a common cause for a dependence to have a non-constant distance. We examined all pairs of multidimensional array references (in the 12 packages) that need to be tested for data dependence. Aliasing effects were ignored, so each reference pair is to one array. We did not find four- or five-dimensional array reference pairs to have coupled subscripts. Table 6 shows the number of two- and three- dimensional array reference pairs which have linear or partially linear subscripts. Table 7 shows that in 9257 pairs of two-dimensional array references that are linear or partially linear, 4105 (44%) of them have coupled subscripts.

3.3 Coefficients of loop indices

A data dependence exists only when there are *integer* solutions which satisfy loop bounds and other constraints. However, it is very time consuming to obtain integer solutions in general. Existing algorithms either check integer solutions without considering loop bounds or only check real-valued solutions one dimension at a time (e. g., see [1], [3], [28]). By doing so, the test can be more efficient, although less effective. Here we give a brief account of two tests that represent the two approaches.

The GCD test

The GCD test is an integer test that ignores loop bounds. It is based on a well-known fact that if a Diophantine equation has solutions, then the *greatest common divisor* (GCD) of its coefficients must divide its constant term.

Example 3.1

```
DO i = .
  DO j = .

    A(2*i+2*j+101) = .                (S1)

    . = A(2*i-2*j)                    (S2)
  END
END
```

For data dependences to exist between S1 and S2 due to the two references to *A*, the subscript of A referenced in S1 (for some values of the index variables) should be equal to that in S2 (for some other values of the index variables). Hence we can derive the following Diophantine equation from the subscripts.

$$2i_1 + 2j_1 - 2i_2 + 2j_2 + 101 = 0$$

The GCD of the coefficients of the variable terms is 2, which does not divide the constant term, 101. Therefore the equation does not have solutions and there is no data dependence between S1 and S2 due to the *A* references.

Banerjee-Wolfe test

Same as the GCD test, the Banerjee-Wolfe test first establishes a Diophantine equation to equate subscripts in two tested array references. However, the test treats the Diophantine equation as a real valued equation whose domain is a convex set defined by constant loop bounds and dependence directions. According to the well-known *intermediate value theorem* in real analysis, the real valued equation has solutions over the given domain if and only if the minimum of the left-hand side is no greater than zero and the maximum no smaller than zero.

Example 3.2

```
DO i = 1 to 30
  DO j = 1 to 30

    A(200-i) = .                      (S1)

    . = A(i+j)                        (S2)
  END
END
```

The Diophantine equation for the above example is as follows.

$$i_1 + i_2 + j_1 - 200 = 0$$

Treated as a real function on the domain of $1 \le i_1, i_2, j_1 \le 30$, the left-hand side of the equation has a maximum of -110. Therefore the equation has no solutions and there is no data dependence between S1 and S2 due to the *A* references.

Banerjee-Wolfe test was first presented in [3]. [28] produced a new version which includes dependence directions in the function's domain. [1] used another version which determines *dependence levels* instead of dependence directions.

A test based only on real values is not an exact test in general. Nonetheless, [3] showed that, in a pair of single-dimensional arrays, if all of the nonzero coefficients of loop indices are either 1 or -1, then a data dependence exists if and only if there are real solutions to the system derived from their subscript expressions. Obviously, for multidimensional array reference pairs which do not have coupled subscripts (cf. Section 3.2), each dimension is independent of each other, so the conclusion will apply. [19] showed that the conclusion could also apply to two-dimensional array references with coupled subscripts. For array references with more than two dimensions, although the conclusion no longer applies in general, small coefficients do make the test for integer solutions much easier. For these reasons, we are interested in the magnitude of coefficients in array references.

The data in Table 8 are from array references (in the 12 packages) which are linear or partially linear. Notice that the percentage shown there is on a dimension-by-dimension basis. The first column shows the percentage of references which have constant subscripts. The second column shows the percentage of references that have nonzero coefficients, but they are either 1 or -1. The third column shows the percentage of references in which some coefficients are greater than 1. The percentage of the third case is very small. This result suggests that for single-dimensional references which are linear or partially linear, real-valued solutions suffice in most cases.

We also checked the coefficients for array reference pairs with coupled subscripts. As expected, we found that among 4,105 pairs of two-dimensional array references with coupled subscripts, most (3,997 pairs, i.e. 97%) have all of their coefficients being 1 or -1. For three-dimensional array reference pairs with coupled subscripts, 60 pairs (100%) have coefficients of 1 or -1. (Note that, in the single dimension cases, we did make the same examination on subscript pairs. Hence, we could not obtain direct results on how often real-valued solutions suffice in single dimension tests)

4. Data Dependences and Data Dependence Test Algorithms

We also did some measurements on the frequency of different data dependence tests being used in Parafrase, the number of array reference pairs found to be independent by each method, and statistics of dependence distances.

3.1 The usage frequency of dependence test methods

As mentioned earlier, different test algorithms have different complexity and capability. In general, more powerful algorithms can handle more general cases with

more accuracy, but they usually require more execution time. Hence, these test algorithms are applied hierarchically in Parafrase. Parafrase includes most of the existing single-dimension test algorithms. Simpler and faster tests are applied first. If they can prove neither data independence nor data dependence, other tests are then applied. It is conceivable that the chance for a test to be used can be affected by the arrangement of the test sequence, because the testing task may be accomplished before the test is used. It was unclear to us what test sequence would achieve the best compiler efficiency. We did not alter the one chosen in Parafrase. That sequence is described in the following to facilitate understanding of our statistics.

First we explain the input and output of the tests. Parafrase usually retests data dependences for a pass that needs the dependence information. As a result, the same pair of references may be tested in different passes, undergoing the same test sequence described below. However, different passes may require to check different dependence direction vectors. The input to the tests is a pair of array references, a loop nesting that encloses either of the references, and a dependence direction vector relevant to the current pass. The output is an answer to whether data dependence exists under the constraint of the dependence direction vector. If the answer is uncertain, data dependence is assumed.

The test sequence:

(1) If both subscripts in a reference pair are constants, then the Constant Test is performed, which simply compares the two constants. If they are not equal, there is no dependence. Otherwise, dependence is assumed for this dimension and the test proceeds to the next dimension.

(2) If (1) does not apply, then the Root Test is performed. The Root Test is a Banerjee-Wolfe test disregarding the constraint of the given dependence direction vector. If it reports data independence, the test terminates. Otherwise, it proceeds to either test (3), test (4), or test (5), depending on the loop nesting.

(3) If test (2) did not prove data independence and both references are in the same singly nested loop, the Exact Test [4], [28] is performed. In this case, the Diophantine equation derived from the subscripts has at most two unknowns; hence it can be determined exactly whether the dependence exists. If dependence does not exist, the test terminates. Otherwise, it proceeds to test (5).

(4) If test (2) did not succeed and test (3) does not apply, but each subscript contains at most one loop index with a nonzero coefficient, then the GCD test [4] is performed. If independence is proven, the test terminates; otherwise, it proceeds to test (5). We have three remarks here.
(Remark 1) In test (3), the Exact Test could be extended to determine exactly what dependence directions are possible. However, Parafrase chooses to disregard dependence directions in the Exact Test. Instead, it uses the Theta Test in (5) to examine the given dependence directions.
(Remark 2) The Exact Test could be extended and to be used in test (4) where both indices may not be the same. But we did not measure the result of this extension.

(Remark 3) The GCD test could be applied to any subscript. However, if there are more than two unknowns in the equation, it is very likely that the common divisor of their coefficients would equal 1, which is not useful for the test (because 1 can divide any number).

(5) If none of (1), (3) and (4) applies, or (3) did not prove independence, then the Theta Test is performed. It is a Banerjee-Wolfe test that uses the given dependence direction vector as a further constraint. Thus it is more accurate than the Root Test in (2). It would either show that the given dependence directions are impossible (in this case, the test terminates), or conclude that some of the directions are possible. In the latter case, if "=" direction is the only remaining possible direction for *every* loop, the All Equal Test in (6) is performed. Otherwise, the test proceeds to the next dimension.

(6) Test enters here from (5). At this point, "=" is the only remaining possible direction for *every loop*. This corresponds to a dependence which crosses no loop iteration. The All Equal Test is performed to see if such a direction vector contradicts the program's control flow. If all possible execution paths from the reference r1 to r2 need to cross an iteration of any loop, then dependence could not exist from r1 to r2 with an "all equal" direction. In other words, independence is proven. Otherwise, dependence is assumed. Test proceeds to the next dimension.

Following the above test steps, we measured the usage frequency and the *independence detection rate* of the single dimension tests in Parafrase. Table 9 gives the measured results. These data are obtained by running each program through Parafrase for detecting Doall loops (i.e. the loops without cross-iteration data dependences). The independence detection rate is the rate a particular test method detects independence between a reference pair, under the constraints of given dependence direction vectors. If a method detects data independence before others have been used, success is counted only for this method, even though other methods used next could potentially detect independence as well. From Table 9, some useful observations can be made. Overall, the above test sequence was applied 119,755 times, which is the sum of the using frequencies of the Constant Test and the Real Root Test. Summing the independences proven by each method together, we have 50,625 independences in total. This represents an overall independence detection rate of 44%. One can also compute the percentage of the independences detected by each test method over the total independences to get an idea of the contribution by each method (in this particular test sequence). It is very important to note that the test sequence was only applied to linear subscripts or partially linear subscripts.

We mentioned earlier that the same pair of references may be tested repeatedly under different constraints of dependence direction vectors and in different passes. All uses of each test method are counted cumulatively, so are its independence detection successes. They are counted cumulatively because all the uses are required and each success contributes to a certain parallelization technique. Recall, for instance, that even if data dependence exists between two statements, absence of cross-iteration dependence directions (i.e. "<" and ">") would allow parallel execution of different loop iterations.

We point out that the All Equal Test benefits from the Theta Test whenever the latter reduces the original dependence direction vector to one of "all equal" directions. The All Equal Test is a good example of using information of control flow within a loop body to sharpen data dependence analysis. It would be interesting to measure how often independences are detected by the Theta Test and the All Equal Test jointly but not by either one alone.

Our result can certainly be refined by further study. For example, the capability of each test method can be evaluated more precisely by applying it first in the test sequence.

3.2 Data dependence distance

As mentioned earlier, many data synchronization schemes and loop partitioning techniques assume *constant dependence distance*. Constant dependence distance also makes loop scheduling on Doacross loops more effective. Complicated dependence patterns are very difficult to handle efficiently. Moreover, dependence distances are difficult to determine if subscript patterns are complicated. As a matter of fact, many parallelization techniques (e.g., [22], [24], [27]) which require constant dependence distances assumed the following three conditions for array subscripts and loop nesting: (1) Each reference has subscripts of the form a*i+c where i is a loop index, and c and a are constants. Note that if more than one loop index appears in a subscript expression, the dependence at the *outer* loop level is likely to have varied distances. (2) There are no coupled subscripts (cf. Section 3.2). (3) Nonzero coefficients are the same in the same dimension.

(1) can be explained by the following example.

Example 4.1

```
DO  I = 3, N
  DO J = 3, N
  . = A(J)              (S1)
    .
  A(I+J) = .            (S2)
  END
END
```

For S1 and S2 in the above example, the dependence distance with respect to the *I* loop is variable. The dependence distance with respect to the *J* loop is *I* which of course remains fixed within an outer loop iteration, but changes when *I* increments.

Condition (2) can be explained by another example.

Example 4.2

```
DO  I = 3, N
  DO J = 3, N
  . = A(I, J)                (S1)
     .
  A(J,I+1)) = .              (S2)
  END
END
```

For S1 and S2 in the above example, the dependence distance with respect to the *I* loop is variable. The dependence distance with respect to the *J* loop is 1. But to find out this, both dimensions need to be considered simultaneously.

Example 2.4 in Section 2 explained condition (3).

Althouth exceptions can be found for each of the three conditions, our tests looked for such simple forms in real programs. If any condition is not satisfied, we did not pursue more sophisticated algorithms to determine whether the distances are constant but leave the distances as uncertain instead. The only exception is when we could determine that "=" is the only possible dependence direction for a loop, in which case the corresponding dependence distance is zero.

We first determine the common nest loops for an array reference pair. Then we measure the dependence distance for each common nest loop. We divide the dependence distances into four classes:

Zero: The dependence does not cross loop iterations. It occurs within an iteration.

Unity: The dependence crosses one iteration (either forward or backward).

Constant: The dependence crosses a constant number (> 1) of iterations (either forward or backward).

Uncertain: The dependence distance is not constant or cannot be decided in our experiment.

Our measurement shows that 73% of the array references with linear or partially linear subscripts have the i+c form. However, not many dependence distances are constant. The results of dependence distance measurement are presented in Table 10. Note that distance is measured for *every* loop common to a pair of dependent references, because of its definition and its usage. The main reasons for uncertain distance are (1) loops not common to both references, (2) coupled subscripts, and (3) nonlinear subscripts. Note also that a good symbolic dependence test could help to reduce the number of nonlinear subscripts and hence could reduce the cases of uncertain dependence distance. A study can be pursued further by counting the cases for different reasons.

5. Conclusion

We presented some measurements critical to data dependence analysis and DO-loop parallel execution. We found that quite a few array references are not amenable to current data dependence test methods. Although we do not have the data to show

how many of those failed tests really have data dependences, more efficient and more accurate tests are certainly very desirable.

We discovered that a lot of subscripts become nonlinear because of unknown symbolic terms. User assertions and interprocedural analysis can be used effectively to reduce unknown symbolic terms (see Tables 3, 4, 5). A more sophisticated and yet efficient symbolic manipulation scheme could be very useful, since a significant number of nonlinear subscripts still remain (Table 5).

We also discovered that a significant number of reference pairs have coupled subscripts (Table 7), which could cause the inaccuracy in the current dependence test algorithms. Efficient algorithms are needed to handle such subscripts.

A welcome result is that an overwhelming majority of nonzero coefficients are either 1 or -1 (Table 8), which allows more efficient real-valued tests to be as accurate as integer-valued tests [5], [19]. It also makes the test on array references with higher dimensions much easier.

We reported the measurements on the usage frequencies and independence detection rates of several well-known data dependence test methods (Table 9). Those measurements followed the testing sequence in Parafrase. Data dependence distance is also measured between each dependent reference pair (Table 10). The large percentage of uncertain dependence distances (over 86%) suggests that more sophisticated algorithms are needed for distance calculation. It also calls for more effective schemes for data synchronization and DO-loop scheduling. However, in our measurements, we did not separate numerical libraries from user programs. It is conceivable that, because of the generality in library routines, there might be more unknown symbolic terms than in user programs. In our study, we have more numerical packages than user programs. Hence the statistics might be more biased toward library routines than toward user programs. In the future studies, it would be interesting to see if there are differences between these two groups of programs.

Acknowledgement

We thank the referees for their careful reading and insightful comments, which have helped us to improve the paper significantly.

REFERENCES

[1] J. R. Allen, "Dependence analysis for subscripted variables and its application to program transformations," Ph.D. dissertation, Department of Mathematical Science, Rice University, Houston, TX, April 1983.

[2] J. R. Allen and K. Kennedy, "Automatic translation of Fortran programs to vector form," Dept. of Computer Science, Rice University, Houston, TX, Rice Comp TR84-9, July, 1984.

[3] U. Banerjee, "Data dependence in ordinary programs," Department of Computer Sciences, University of Illinois at Urbana-Champaign, Rpt. No. 76-837, Nov. 1976.

[4] U. Banerjee, "Speedup of ordinary programs", Ph.D. dissertation, University of Illinois at Urbana-Champaign, DCS Rpt. No. UIUCDCS-R-79-989, 1979.

[5] U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, Norwell, Mass., 1988.

[6] J. Banning, "A method for determining the side effects of procedure calls," Ph.D. dissertation, Standford University, Aug. 1978.

[7] M. Burke and R. Cytron, "Interprocedural dependence analysis and parallelization," *Proc. of the ACM SIGPLAN'86 Symposium on Compiler Construction. ACM SIGPLAN Not.*, Vol. 21, No. 7, pp. 162-175, July 1986.

[8] D. Callahan, K. Cooper, K. Kennedy, and L. Torczan, "Interprocedural constant propagation," *Proc. of the ACM SIGPLAN '86 Symp. on Compiler Construction, ACM SIGPLAN Not.*, Vol. 21, No. 6, June 1986.

[9] R. G. Cytron, "Compile-time scheduling and optimization for multiprocessors," Ph.D. dissertation, University of Illinois at Urbana-Champaign, DCS Rep. UIUCDCS-R-84-1177, 1984.

[10] J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979. Spring-Verlag, Heidelberg, 1976.

[11] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau, "Parallel processing: A smart compiler and a dumb machine," *Proc. of the ACM SIGPLAN '84 Symp. on Compiler Construction, SIGPLAN Notices Vol. 19, No. 6*, June 1984.

[12] Z. Fang, P. Yew, P. Tang, and C. Zhu, "Dynamic processor self-scheduling for general parallel nested loops," *Proc. 1987 Int'l. Conf. on Parallel Processing* (August, 1987), pp. 1-10.

[13] D. Kuck, R. Kuhn, B. Leasure, and M. Wolfe, "The structure of an advanced vectorizer for pipelined processors", *Proceedings of COMPSAC 80, The 4th International Computer Software and Applications Conference*, October 1980, pp. 709-715.

[14] D. Kuck, R. Kuhn, D. Padua, B. Leasure and M. Wolfe, "Dependence graphs and compiler organizations", *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, Williamsburgh, VA, January 1981, pp. 207-218.

[15] D. Kuck, A. Sameh, R. Cytron, A. Veidenbaum, et al. "The effects of program restructuring, algorithm change, and architecture choice on program performance," *Proc. 1984 Int'l. Conf. on Parallel Processing*, pp. 129-138, August 1984.

[16] D. Kuck, *The Structure of Computers and Computations*, Vol. 1, John Wiley and Sons, New York, 1978.

[17] D. J. Kuck, Y. Muraoka, and S.-C. Chen, "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup," *IEEE Trans. Compt.*, vol. c-21, No. 12, pp. 1293-1310, Dec. 1972.

[18] A. Lichnewsky and F. Thomasset, "Introducing symbolic problem solving techniques in the dependence testing phases of a vectorizer," *Proc. 1988 Int'l. Conf. on Supercomputing*, July, 1988.

[19] Z. Li, P.-C. Yew, and C.-Q. Zhu, "An efficient data dependence analysis for parallelizing compilers," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, No. 1, pp. 26-34, Jan. 1990.

[20] A. Nicolau and J. A. Fisher, "Measuring the parallelism available for very long instruction word architectures," *IEEE Trans. Comput.*, vol. c-33, No. 11, pp. 968-976, Nov. 1984.

[21] D. A. Padua, "Multiprocessors: Discussions of some theoretical and practical problems," Ph.D. dissertation, University of Illinois at Urbana-Champaign, DCS Rep. UIUCDCS-R-79-990, Nov. 1979.

[22] J.-K. Peir, "Program partitioning and synchronization on multiprocessor systems," Ph.D. dissertation, University of Illinois at Urbana-Champaign, DCS Rep. UIUCDCS-R-86-1259, Mar. 1986.

[23] C. D. Polychronopoulos, D. J. Kuck, and D. A. Padua, "Optimal processor allocation of programs on multiprocessor systems," *Proc. 1986 Int'l. Conf. on Parallel Processing*, Aug. 1986.

[24] W. Shang and J. A. B. Fortes, "Independent partitioning of algorithms with uniform dependencies," *Proc. 1988 Int'l. Conf. Parallel Processing*, Aug. 1988, pp. 26-33.

[25] B. J. Smith, "A pipelined, shared resource MIMD computer," in *Proc. 1978 Int'l. Conf. Parallel Processing*, Aug. 1978, pp. 6-8.

[26] P. Tang, P. Yew, and C. Zhu, "Impact of self-scheduling order on performance of multiprocessor systems," *Proc. of ACM 1988 Int'l. Conf. on Supercomputing* (July, 1988)

[27] P. Tang, P. Yew, and C. Zhu, "Algorithms for generating data-level synchronization instructions," Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Rpt. No. 733, Urbana, January, 1988.

[28] M. J. Wolfe, "Optimizing Supercompilers for Supercomputers", Ph.D. dissertation, University of Illinois at Urbana-Champaign, DCS Rpt. No. UIUCDCS-R-82-1105, October 1982.

[29] C. Q. Zhu and P. C. Yew, "A scheme to enforce data dependence on large multiprocessor systems," *IEEE Trans. Software Eng.*, vol. SE-13, pp. 726-739, June 1987.

Package	Description	Subroutines	Lines
LINPACK	Linear system package	60	10175
EISPACK	Eigensystem package	70	11700
ITPACK	Sparse matrix algorithms (iterative methods)	68	5591
MSL	Mathematic science library (CDC)	407	20473
FISHPAK	Separable elliptic partial differential equations package	159	22647
ACM	Random algorithms from ACM	25	2712
OLD	Checon Chebyshev economization program	74	3218
BARO	Shallow water atmospheric model	8	1052
NASA	Program from NASA	4	780
SMPL	Flow analysis program	15	2072
WEATHER	Weather forecasting program	64	3918
SPICE	Circuit simulation program	120	17857
Total		1074	102195

Table 1. Analyzed Fortran packages

Dim.	References		Linear		Partially linear		nonlinear	
1	10421	56%	6282	60%	not	applicable	4139	40%
2	6721	36%	2671	40%	1929	29%	2121	31%
3	1300	7%	800	62%	459	35%	41	3%
4	97	1%	0	0%	97	100%	0	0%
5	10	0.05%	0	0%	10	100%	0	0%
Total	18549	100%	9753	53%	2495	13%	6301	34%

Table 2. Linearity of array subscripts

Dim.	Causes			
	Unknown var.		Array element	
1	3441	83%	698	17%
2	5995	97%	176	3%
3	782	98%	14	2%
4	206	100%	0	0%
5	22	100%	0	0%

Table 3. Causes of nonlinear subscript expressions

1 dimension	Undefined		Defined	
Total references	3061		3061	
w/ nonlinear subscripts	1452	47%	852	28%
Num. of unknown var.	1324		695	

2 dimensions	Undefined		Defined	
Total references	2897		2897	
w/ nonlinear subscripts	1301	45%	426	15%
w/ partially linear subscripts	787	27%	723	25%
Num. of unknown var.	3384		1560	

Table 4. The effects of user assertions for data definition
(from the subset: 6 packages)

1 dimension	Undefined		Defined		Def. & Intp.*	
Total array references	623		623		623	
references w/ nonlinear subscripts	305	49%	236	38%	181	29%
Num. of unknown variables	289		220		165	

2 dimensions	Undefined		Defined		Def. & Intp.*	
Total array references	250		250		250	
references w/ nonlinear subscripts	99	40%	64	26%	56	22%
references w/ partially linear subscripts	67	27%	47	19%	56	22%
Num. of unknown variables	265		175		143	

* After both user assertions and interprocedural analysis

Table 5. The effects of interprocedural analysis
(from Linpack)

	2 dimensions		3 dimensions	
Total array reference pairs	18698		2867	
Pairs w/ linear/partially linear subscripts	9257	50%	2798	98%

Table 6. Reference pairs with linear or partially linear subscripts

	2 dimensions		3 dimensions	
Pairs w/ linear/partially linear subscripts	9257		2798	
Pairs w/ coupled subscripts (linear)	2935	32%	0	0%
Pairs w/ coupled subscripts (partially linear)	1170	13%	60	2%

Table 7. Reference pairs with coupled subscripts

Dim	Coeff=0		Abs(coeff)=1		Abs(coeff)>1		Total	
1	978	15%	4756	76%	548	9%	6282	100%
2	1095	17%	5215	81%	132	2%	6442	100%
3	994	32%	1615	52%	482	16%	3091	100%
4	88	46%	103	54%	0	0%	191	100%
5	8	29%	20	71%	0	0%	28	100%

Table 8. The coefficients of array references with linear or partially linear subscripts

Test method	Usage frequency	Indep. detection rate	
Constant	43935	31229	72%
Real root	75820	4555	6%
Exact	13959	7563	54%
GCD	12832	409	3%
Theta	30605	4311	14%
"All equal"	13552	2558	19%

Table 9. The usage frequency and the independence detection rate of various dependence test methods

Zero Distance	8529	11%
Unity Distance	1097	2%
Constant (>1) Distance	797	1%
Uncertain Distance	65931	86%

Table 10. Dependence distance