Parallelism for Single Loops with Multiple Dependences

Sam Jin Jeong*
Division of Information and Communication Engineering
BaekSeok University, Cheonan, Korea

ABSTRACT

We review some loop partitioning techniques such as loop splitting method by thresholds and Polychronopoulos' loop splitting method for exploiting parallelism from single loop which already developed. We propose improved loop splitting method for maximizing parallelism of single loops with non-constant dependence distances. By using the iteration and distance for the source of the first dependence, and by our defined theorems, we present generalized and optimal algorithms for single loops with non-uniform dependences. The algorithms generalize how to transform general single loops with one dependence as well as with multiple dependences into parallel loops.

Keywords: *Parallelizing Compiler, Loop Splitting, Loop, Multiple Dependences, Non-uniform Dependences*

1. INTRODUCTION

Computationally expensive programs spend most of their time in the execution of DO-loops. Therefore, an efficient approach for exploiting potential parallelism is to concentrate on the parallelism available in loops in ordinary programs and has a considerable effect on the speedup [1-2].

Through some loop transformations using a dependence distance in single loops [3][7], a loop can be splitted into partial loops to be executed in parallel without violating data dependence relations, that is, the size of distance can be used as a reduction factor [3], which is the number of iterations that can be executed in parallel. In the case of non-constant distance such that it varies between different instances of the dependence, it is much more difficult to maximize the degree of parallelism from a loop.

Partitioning of loops requires efficient and exact data dependence analysis. A precise dependence analysis helps in identifying dependent/independent iterations of a loop. And it is important that appropriate dependence analysis be applied to exploit maximum parallelism within loops. We can consider some tests that examine the dependence of one-dimensional subscripted variables – the separability test, the GCD test and the Banerjee test [5][8]. In general, the GCD test is applied first because of its simplicity, even if it is an approximate test. Next, for the case that the gcd test is true, the separability test is attempted again, and through this exact test, it can be obtained additional information such as solution set, and minimum and maximum distances of dependence, as well as whether the existence of dependence or not.

When we consider the approach for single loops, we can review two partitioning techniques proposed in [3] which are fixed partitioning with minimum distance and variable partitioning with ceil(d*(i)*). However, these leave some parallelism unexploited, and the second case has some constraints.

The rest of this paper is organized as follows. Chapter two describes our loop model, and introduces the concept of data-dependence computation in actual programs. In chapter three, we review some partitioning techniques of single loops such as loop splitting method by thresholds and Polychronopoulos' loop splitting method. In chapter four, we propose a generalized and optimal method to make the iteration space of a loop into partitions with variable sizes. In the method, two algorithms can be considered as one for loops with one dependence and the other for loops with multiple dependences. Finally, we conclude in chapter five with the direction to enhance this work.

2. PROGRAM MODEL AND DATA DEPENDENCE ANALYSIS

For data-dependence computation in actual programs, the most common situation occurs when we are comparing two variables in a single loop and those variables are elements of a one-dimensional array, with subscripts linear in the loop index variable. Then this kind of loop has a general form is shown if Fig. 1. Here, l, u, a_1, a_2, b_1 and b_2, are integer constants known at compile time.

DO $I = l, u$
S_1: $A(a_1*I + a_2) = \cdots$
S_2: $\cdots = A(b_1*I + b_2)$
END

Fig. 1 A single loop model.

* *Corresponding author. E-mail : sjjeong@cheonan.ac.kr*
Manuscript received Aug. 27, 2007;accepted Sep. 20, 2007

For dependence between statements S_1 and S_2 to exist, we must have an integer solution (i, j) to equation (1) that is a linear diophantine equation in two variables. The method for solving such equations is well known and is based on the extended Euclid's algorithm [4].

$$a_1 i + a_2 = b_1 j + b_2 \text{ where } l \le i, j \le u \quad (1)$$

This equation may have infinitely many solutions (i, j) given by a formula of the form:

$$(i, j) = ((b_1/g)t + i_1, (a_1/g)t + j_1$$
$$\text{where } (i_1, j_1) = ((b_2-a_2)\, i_0/g, (b_2-a_2)\, j_0/g) \quad (2)$$

i_0, j_0 are any two integers such that $a_1 i_0 - b_1 j_0 = g(\gcd(a_1, b_1))$ and t is an arbitrary integer [5][6]. Acceptable solutions are those for which $l \le i, j \le u$, and in this case, the range for t is given by

$$\max(\min(\alpha, \beta), \min(\gamma, \delta)) \le t \le \min(\max(\alpha, \beta), \max(\gamma, \delta))$$
$$\text{where } \alpha = -(l - i_1)/(b_1/g), \beta = -(u - i_1)/(b_1/g),$$
$$\gamma = -(l - j_1)/(a_1/g), \delta = -(u - i_1)/(a_1/g). \quad (3)$$

3. RELATED WORKS

Now, we review some partitioning techniques of single loops. We can exploit any parallelism available in such a single loop in Fig. 1, by classifying the four possible cases for a_1 and b_1, coefficients of the index variable I, as given by (4).

(a) $a_1 = b_1 = 0$
(b) $a_1 = 0, b_1 \neq 0$ or $a_1 \neq 0, b_1 = 0$
(c) $a_1 = b_1 \neq 0$
(d) $a_1 \neq 0, b_1 \neq 0, a_1 \neq b_1$ (4)

In case 4(a), because there is no cross-iteration dependence, the resulting loop can be directly parallelized. In the following subsections, we briefly review several loop splitting methods for the cases of 4(b) through 4(d).

3.1 Loop splitting by thresholds

A threshold indicates the number of times the loop may be executed without creating the dependence. In case 4(b), for a dependence to exist, there must be an integer value i of index variable I such that $b_1 * i + b_2 = a_2$ (if $a_1 = 0$) or $a_1 * i + a_2 = b_2$ (if $b_1 = 0$) and $l \le i \le u$. If there is no solution, then there is no cross-iteration dependence and the loop can also be parallelized. And if integer exists, then there exist a flow dependence (or anti-dependence) in the range of I, $[l, u]$ and an anti-dependence (or flow dependence) in $[i, u]$. In this case, be breaking the loop at the iteration $I = i$ (called *turning threshold*), the two partial loops can be transformed into parallel loops.
In case 4(c), let (i, j) be an integer solution to (1), then there exists a dependence in the range of I and the dependence distance (d) is $|j - i| = |(a_2 - b_2)/a_1|$. Here, the loop can be transformed into two perfectly nested loops; a serial outer loop with stride d (called constant threshold) and a parallel inner loop [5].

In case 4(d), an existing dependence is non-uniform since there is a non-constant distance, that is, such that it varies between different instances of the dependence. And we can consider exploiting any parallelism for two cases when $a_1 * b_1 < 0$ and $a_1 * b_1 > 0$. Suppose now that $a_1 * b_1 < 0$. If (i, i) is a solution to (1), then there may be all dependence sources in (l, i) and all dependence sinks in $[i, u]$. Therefore, by splitting the loop at the iteration $I = i$ (called *crossing threshold*), the two partial loops can be directly parallelized [5]. Fig. 2(c) shows the general form of loop splitting by the crossing threshold.

```
                                    DOALL I = 1, 4
                                      A(I+5) = ···
DO I = 1, 10                            ··· = A(9)
  A(I+5) = ···          =>          ENDDO
    ··· = A(9)                    DOALL I = 5, 10
ENDDO                                 ···
                                    ENDDO
```
(a) Using turning threshold.

```
                                  DO I' = 1, N, abs((b2-a2)/a1)
DO I = 1, N                         DOALL  I  = I', I'+(abs((b2-a)/a1)-1
  A(a1*I+ a2) = ···                   A(a1*I+ a2) = ···
    ··· = A(b1*I+ b2)     =>              ··· = A(b1*I+ b2)
ENDDO                               ENDDO
                                  ENDDO
```
(b) Using constant threshold.

```
                                  DOALL I = 1, floor((b2 - a2)/ (a1+ b1))+1,N
                                    A(a1*I+ a2) = ···
DO I = 1, N                               ··· = A(b2+ b1*I)
  A(a1*I+ a2) = ···       =>      ENDDO
    ··· = A(b2+ b1*I)             DOALL I = floor((b2 - a2)/ (a1+ b1))+1,N
ENDDO                                 ···
                                  ENDDO
```
(c) Using crossing threshold.

Fig. 2 The loop splitting by thresholds.

3.2 Polychronopoulos' loop splitting

We can also consider exploiting any parallelism for the case 4(d) when $a_1 * b_1 \ge 0$. We will consider three cases whether it exists only flow dependence, anti-dependence, or both in the range of I. First, let (i, j) be an integer solution to (1). If the distance, $d(i)$ depending on i, as given by (5), has a positive value, then there exists a flow dependence, and if $d_a(j)$ depending on j, as given by (6), has a positive value, then there exists an anti-dependence. Next, if (x, x) is a solution to (1) (x may not be an integer.), then $d(x) = d_a(x) = 0$ and there may exist a flow (or anti-) and an anti-dependence (or flow) before and after $I = \text{ceil}(x)$, and if x is an integer, then there exists a loop-independent dependence at $I = x$. Here, suppose that Then for each value of I, the element $A(a_1 * I + a_2)$ defined by that iteration cannot be consumed before ceil($d(i)$) iterations later, and this indicates that ceil($d(i)$) iterations can execute in parallel.

$$d(i) = j - i = D(i)/b_1, \text{ where } D(i) = (a_2 - b_1) * i + (a_2 - b_1) \quad (5)$$
$$d_a(j) = i - j = D_a(j)/a_1, \text{ where } D_a(j) = (b_1 - a_1) * j + (b_2 - a_2) \quad (6)$$

Consider the loop, as given in Fig. 3, in which there exist flow dependences. $d(i) = D(i)/b_1 = (i + 5)/2 > 0$ for each value of I and $d(i)$ have integer values, 3, 4, 5, ... as the value of I is incremented.

```
DO I =1, N
S1 :  A(3I+1) = ...
S2 :    ... = A(2I-4)
ENDDO
```

Fig. 3 An example of a single loop.

Fig. 4 shows the results of applying two transformations using minimum distance and ceil(d(i)) proposed in [3] to the loop in Fig. 3, respectively. However these leave some parallelism unexploited. Moreover, the transformed loop in Fig. 4(b) has some constraints: It must be only a flow dependence in the original loop, the first iteration of the original loop must be the iteration at which a dependence source exists, and d(i) ≥ 1 for each value of I.

```
DO I' = 1, N, 3
  DOALL I = I', min(N, I'+2)
    A(3I+1) = ···
          ··· = A(2I-4)
  ENDDO
ENDDO
```

(a) Using minimum distance.

```
I' = 1
L: inc = min(N-I', ceil((I'+5)/2)-1)
DOALL I = I'+inc
    A(3I+1)= ···
          ··· =A(2I-4)
ENDDO
I' = I'+inc+1
If I' < N then goto L
```

(b) Using ceil(d(i)).

Fig. 4 Polychronopoulos' loop splitting.

4. MAXIMI ING PARALLELISM FOR SINGLE LOOPS

From a single loop with non-constant distance such that it satisfies the case (d) in (4) and $a_1 * b_1 > 0$, we can get the following Lemmas. For convenience' sake, suppose that there is a flow dependence in the loop.

Lemma 1 The number of iterations between a dependence source and the next source, sd is given by $| b_1 | / g$ iterations where $g = gcd(a_1, b_1)$.

And we can know the facts that if we obtain the iteration for the source of the first dependence, then we can compute the others easily, and $i \equiv j \pmod{s_d}$ for i, j are arbitrary iterations for all sources.

Lemma 2 The dependence distance, that is, the number of iterations between the source and the sink of a dependence, is $D(i)/b_1$ where $D(i) = (a_1 - b_1) * i + (a_2 - b_2)$, and the increasing rate of a distance per one iteration, d' is given by $(a_1 - b_1)/ b_1$. And the difference between the distance of a dependence and that of the next dependence, d_{inc} is $| a_1 - b_1 | / g$.

Hence, if we obtain the distance for the source of the first dependence, then we can compute the others easily. Also for the case of anti-dependence, similarly, Lemma 1 and 2 can be represented. Namely, s_d is given by $| a_1 | / g$ iterations where $g = gcd(a_1, b_1)$, the distance is given by (6), and d' is $(b_1-a_1)/a_1$. And $d_{inc} = d' * s_d = (b_1-a_1)/a_1 * | a_1 | / g = | b_1 - a_1 | / g$.
By using the iteration and distance for the source of the first dependence, and concepts defined by Lemma 1 and 2, we obtain the generalized and optimal algorithm to maximize parallelism from single loops with non-uniform dependences.

4.1 Transformation of Loops with One Dependence

Procedure MaxSplit shows the transformation of single loops satisfying the case (d) in (4) and $a_1 * b_1 > 0$ into partial parallel loops [9].
The following Procedure MaxSplit_2 generalizes how to transform general single loops, as shown in Fig. 1, into parallel loops.

Procedure MaxSplit_2
/* A generalized transformation of single loops into parallel loops */
BEGIN
  /* (1) Testing for data dependence. */
Step 1:
  If (gcd test) is not true then
    {Transform a loop into a parallel loop directly; Stop};
  If (separability test) is not true then
    {Transform a loop into a parallel loop directly; Stop};
  /* (2) Data dependence computation and transformation for each of cases in (4). */
Step 2:
  If case (a) is true then
    {Transform a loop into a parallel loop directly; Stop};
  If case (b) or case (c) or $a_1 * b_1 < 0$ is true then
    {Compute a turning threshold or a constant threshold or a crossing threshold and process the loop splitting by using those, respectively; Stop};
Step 3: /* The case satisfying (d) in (4) and $a_1 * b_1 > 0$ */
  If (x, x) is a solution to (1) then
    go to Step 5;
Step 4: /* The case that there exists only a flow dependence (or anti-dependence) */
    /* There exists flow dependence */
  If $d(i) > 0$ for $I = i$ within the range of loop then
    {$s_d = | b_1 | / g$; $d_{inc} = | a_1 - b_1 | / g$};
    /* There exists anti-dependence */
  else {$s_d = | a_1 | / g$; $d_{inc} = | b_1 - a_1 | / g$};
    /* α, β: the iteration and distance for the source of the first dependence in the loop, respectively */
    Compute α, β by the separability test;
Step 4.1: Call MaxSplit(l, u, s_d, d_{inc}, α, β);
  Stop;
Step 5: /* The case that there exist both flow and anti-dependences in the range of loop */
  The same as Step 4 for the range of $l \le I \le$ ceil(x);
Step 5.1: Call MaxSplit(l,ceil(x), s_d , d_{inc}, α, β);
Step 6: The same as Step 4 for the range of ceil(x)+1 $\le I \le u$;
Step 6.1: Call MaxSplit(ceil(x) + 1, u, s_d , d_{inc}, α, β);
Step 7: Merge the last block splitted by Step 5 and the first block splitted by Step 6 together;
  Stop;
END MaxSplit_2

In step 5-6, if there exist a flow (or anti-) and an anti-dependence (or flow) before and after I = ceil(x), as defined in step 3, then dividing the loop two parts, and each of parts can be transformed by Procedure MaxSplit. No dependences are violated by the transformed block in step 7, since there may be different type of dependences before and after I = ceil(x) and a loop-independent dependence may exist at $I = x$, even if it exists. An example of such case is shown in Fig. 5(a). In the loop of Fig. 5(a), there exist both anti-and flow dependences before and after i = 7. The unrolled version of this loop is shown in Fig. 5(b). The results of this loop transformed by step 5 and 6 are shown in Fig. 5(c), and the result of those merged by step 7 is the loop as in Fig. 5(d).

```
DO I = 1, 12
   A(3*I-2) = ···
      ··· = A(2*I+5)
END
```

(a) A serial loop

I	A(3*I-2)	A(2*I+5)
1	A(01)	A(07)
2	A(04)	A(09)
3	A(07)	A(11)
4	A(10)	A(13)
5	A(13)	A(15)
6	A(16)	A(17)
7	A(19)	A(19)
8	A(22)	A(21)
9	A(25)	A(23)
10	A(28)	A(25)
11	A(31)	A(27)
12	A(34)	A(29)

(b) The unrolled version of transformed loop

```
DOALL I = 1, 2          DOALL I = 1, 2
   ···                     ···
DOALL I = 3, 4          DOALL I = 3, 4
   ···                     ···
DOALL I = 5, 7     =>
                        DOALL I = 5, 9
DOALL I = 8, 9             ···
   ···
DOALL I = 10, 12        DOALL I = 10, 12
   ···                     ···
```

(c) By step 5 and 6. (d) By step 7.

Fig. 5 The results of a loop with both flow and anti-dependences transformed by Procedure MaxSplit_2.

4.2 Transformation of Loops with Multiple Dependences

In the previous section, we considered only the case that one dependence relation exists in loops. In this section, a transformation of single loops with multiple dependences is presented by extending the method of loops with one dependence.

We assume the existence of m non-uniform dependences in a single loop, then the algorithm to exploit any parallelism available in single loops with multiple dependences can be given by Procedure MultiSplit. The steps of Procedure MultiSplit are similar to those of Procedure MaxSplit. Procedure MultiSplit produces the array of the first iteration in each block, St[], and transforms the loop into blocks with variable size, St[i+1] – St[i].

Procedure MultiSplit (l, u, s_d[], α[], β[])
/* Transformation of single loops with multiple dependences */
BEGIN
/* (1) to find the first iteration in the ith block, St[i].
St[k]: the first source iteration in any block of each of m dependences.
$d_k(i)$: the distance at any iteration I for each of m dependences.
S_d[k]: the difference between two adjacent source iterations for each of m dependences.
α[k], β[k]: the iteration and distance of the first source for each of m dependences computed by the separability test, respectively */
Step 1: i = 1; St[1] = l;
  Sr[k] = α[k] and d_k(Sr[k]) = β[k] for $1 \le K \le m$;
Step 2: St[i+1] = min {Sr[k] + d_k(Sr[k]} for $1 \le K \le m$;
  If St[i+1] $\ge u$ then {St[i+1]= u + 1; go to Step 5};
Step 3: Sr[k] = St[I+1] + q[k] for $1 \le K \le m$,
  Where $0 \le q[k] \le s_d[k]$ and q[k] = (α[k] – St[k+1]) mod s_d[k];
Step 4: Compute d_k[k](Sr[k]) for $1 \le K \le m$;
  i = i + 1 ; go to Step 2;
/* (2) to transform the loop into blocks with variable sizes, St[i+1] – St[i]. */
Step 5: i = 1; $I' = l$;
  While $I' \le u$ Do
    Inc = St[i+1] – St[i];
    DOALL $I = I', I'$ + inc – 1
      $A_{11}(a_{11} * I + a_{12}) = \ldots$
        ···
          $\ldots = A_m(b_{m1} * I + b_{m2})$
    ENDDO
    $I' = I'$ + inc; i = i + 1;
  EndWhild
END MultiSplit

```
                                    DOALL I = 1, 2
                                       ···
DO I = 1, N                         DOALL I = 3, 6
   A1(3I + 1) = A3(I –3)               ···
   A2(2I) = A1(2I – 4)        =>    DOALL I = 7, 12
   A3(2I) = A2(I – 1)                  ···
ENDDO                               DOALL I = 13, 21
                                       ···
                                    DOALL I = 22, 36
                                       ···
```

(a) (b)

Fig. 6 An example of loop with multiple dependences transformed by Procedure MultiSplit.

As an example, let's consider the loop given in Fig. 6(a). By Procedure MaxSplit, St[i]s for array A_1, A_2, and A_3 are obtained as {1, 4, 10, 19, 31, 49, ···}, {1, 3, 7, 15, 31, 63, ···} and {1, 5, 13, 29, 61, ···}, respectively. However, Procedure MultiSplit produces {1, 3, 7, 13, 22, 33, 57, ···}. This loop can be splitted as shown in Fig. 6(b).

5. CONCLUSIONS

In this paper, we have studied the parallelization of single loop with non-uniform dependences for maximizing parallelism. For single loops, we can review two partitioning techniques which are fixed partitioning with minimum distance and variable partitioning with ceil(d*(i)*). However, these leave some parallelism unexploited, and the second case has some constraints. Therefore, we propose a generalized and optimal method to make the iteration space of a loop into partitions with variable sizes. In the method, two algorithms are considered as one for loops with one dependence and the other for loops with multiple dependences. First algorithm generalizes how to transform general single loops with one dependence into parallel loops. Second one generalizes how to transform single loops with multiple dependences into parallel loops.

Our future research work is to consider the extension of our proposed method to n-dimensional space.

6. REFERENCES

[1] W. Zhang, G. Chen, M. Kandeemir, and M. Karakoy, "Interprocedural Optimizations for Improving Data Cache Performance of Array-Intensive Embedded Applications," in DAC 2003, Anaheim, California, 2003.

[2] D. S. Park, M. H. Choi, "Interprocedural Transformations for Parallel Computing," in Journal of Korean Multimedia Society, vol. 9, no. 12, pp.1700-1708, Dec., 2006.

[3] C. D. Ploychronopoulos, "Compiler optimizations for enhancing parallelism and their impact on architecture design," in IEEE Trans. computers, vol. 37, no. 8, pp. 991-1004, Aug. 1988

[4] D. E. Knuth, The Art of Computer Programming, vol. 2: Seminumerical Algorithms, Reading, MA: Addison-Wesley, 1981

[5]] H. Zima and B. Chapman, Supercompilers for Parallel and Vector Computers, Addison-Wesley, 1991

[6] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, "Automatic program parallelization," in Proceedings of the IEEE, vol. 81, no. 2, pp.211-243, Feb 1993.

[7] J. R. Allen and K. Kennedy. "Automatic translation of Fortran programs to vector form," in ACM Trans. Programming Languages and Systems, vol. 9, no. 4, pp. 491-542, Oct. 1987.

[8] M. J. Wolfe, Optimizing Supercompilers for Supercomputers, Cambridge, MA: MIT Press, 1989.

[9] S. J. Jeong, "A Loop Transformation for Parallelism from Single Loops," in International Journal of Contents, vol. 2, No. 4, pp.8-11, Dec. 2006

Sam-Jin Jeong

He received the B.S. in polymer science from KyungBuk National university, Korea in 1979, and the M.S. in computer science from Indiana university, USA in 1987, and also received Ph.D. in computer science from ChungNam National university, Korea in 2000. From 1988 to 1991, he was a senior research staff at SamSung Electric Co. From 1992 to 1997, he was an assistant professor at Haecheon University. Since then, he has been with BaekSeok University as a professor. His main research interests include parallelizing compiler, parallel systems, general compiler, and programming languages.