Parallelizing Imperfectly Nested Loops

Ki-Chang Kim

Abstract

Loops are some of the richest program constructs where parallelism is available. Exploiting fine-grain parallelizm out of these constructs is particularly important in light of the growing popularity of superscalar and VLIW machines. This paper explains how the fine-grain parallelization techniques can be generalized to handle nested loops. Our technique integrates nested loop parallelization techniques at the fine-grain level, thus exposing more fine-grain parallelism, and is flexible enough to handle non-perfectly nested loops. Examples and some experimental results are presented to illustrate our approach.

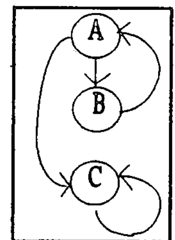
I. Introduction

Loops are some of the richest program constructs where parallelism is available. Especially as the nest depth of the loop increases, the time that the CPU spends in it sharply climbs up. Many vectorization techniques have been developed to exploit the parallelism hidden in this construct[1,2,3,4]. For loops that are not vectorizable, however, the general technique is the Wavefront method [5,6,7].

An elegant way of implementing the Wavefront method through the combination of loop skewing and loop interchange is shown in [3]. [8,9] show recent developments in this direction. But since in the Wavefront method, the unit of scheduling is an iteration, the parallelism inside iterations is not utilized. Each iteration is regarded as an atomic computational unit and executed by a single processor sequentially. This approach is useful to reduce the parallelizing complexity for nested loops, but it also reduces the amount of parallelism exploitable. Of course, the fine grain loop body parallelism may trivially be exposed after the Wavefront method is applied, but this misses some of the fine-grain parallelism that may exist across loop dimensions (not iteration-level parallelism that is already exposed by the Wavefront method)[18]. Integrating fine and coarse grain parallelism is particularly important in light of the growing popularity of superscalar and VLIW machines.

Another limitation of the Wavefront method is that it can be applied only to perfectly nested loops. A perfectly nested loop is one in which all loops are iterating over the same set of statements as can be seen in Figure 2(a). Non-perfectly nested loops represent the general form of loops, where a loop can contain a set of statements or another loop, or both. Thus, a perfectly nested loop is a special case of a non-perfectly nested loop. Figure 4(a) shows an example of a non-perfectly nested loop. Since non-perfectly nested loops are more common in the general code, parallelizing only perfectly nested loops is often not enough.

```
For i = 0 to N - 1
   A: A[i] = f(B[i-1])
   B: B[i] = g(A[i])
   C: C[i] = h(A[i],C[i-1])
Endfor
```



(a) The source code and its dependence graph

```
For i = 0 to N - 1
   A: A[i] = f(B[i-1])
   B: B[i] = g(A[i]);    C: C[i] = h(A[i],C[i-1])
Endfor
```

(b) An optimal schedule for the loop in Figure 1(a). Statement B and C are in the same line to show that they can be executed in parallel.

0	A0			
1	B0 C0			
2		A1		
3		B1 C1		
4			A2	
5			B2 C2	
6			……	……

(c) As-soon-as-possible schedule for the code in Figure 1(a).

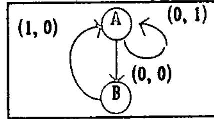
Fig. 1. A parallelization example for 1-dimensional case.

Manuscript received August 1, 1995; accepted October 6, 1995.

The author is with Department of Computer Science, Inha University, Incheon, Korea.

Parallelizing loops at the fine grain level has been pursued by numerous researchers [10,11,12,13,14]. For one dimensional loops, given enough resources, there exists an optimal solution [15].

```
For i1 = 0 to N1 - 1
  For i2 = 0 to N2 - 1
    A: A[i1, i2]=f(A[i1, i2-1], B[i1-1, i2])
    B: B[i1, i2]=g(A[i1, i2])
  Endfor
Endfor
```



(a) The source code and its dependence graph for a 2-dimensional case.

0	A00													
1	B00	A01												
2		B01	A02			A10								
3			B02	A03		B10	A11							
4				B03	A04		B11	A12			A20			
5					B04	A05		B12	A13		B20	A21		
6						B04	···		B13	A14		B21	A22	
7							···			B14	···		B2	···
8											···			···

(b) as-soon-as-possible schedule for the loop in Figure 2(a).

```
For t = 0 to (N1-1)2 + (N2-1) + 1
   Forall i1 = L1 to U1
      Forall i2 = L2 to U2
         Case MAX(0, t - 2i1 - i2) is
         0: A[i1,i2-1]=f(A[i1,i2-1],B[i1-1,i2])
         1: B[i1,i2]=g(A[i1,i2])
         Endcase
      Endforall
   Endforall
Endfor
```

where $L1 = MAX(0, \lceil (t-N_2)/2 \rceil)$, $U1 = MIN(N1-1, \lfloor t/2 \rfloor)$, $L2 = MAX(0, t-2i1-1)$, and $U2 = MIN(N2-1, t-i1)$)

(c) A parallel code for the loop in Figure 2(a). The details of this transformation are in Section 3.3.

Fig. 2. Parallelization example for 2-dimensional case.

Figure 1(a) and (b) show an example loop and its parallelized form at the statement level. The parallelized form can be obtained by the following process. We unwind the loop repeatedly while scheduling each statement instance at the earliest cycle it can be executed until a pattern is detected in the schedule[1]. Figure 1(c) shows this scheduling process, where the pattern is appearing every two cycle. Then we replace the original loop body with this pattern. The schedule obtained this way is known to be optimal, and shown in Figure 1(b). However, previous attempts to expose fine grain parallelism in nested loops have not been totally satisfactory. For example, Loop Quantization [16] computes the amount of unwinding for nested loops, but it is hard to specify the right amount of unwinding for each dimension. [14] shows another way of handling nested loops in a hierarchical manner. The inner-most loops are compacted first. Then, the outer loop surrounding these inner-most loops is compacted regarding the inner loop as a single statement. This process continues until the outer-most loop is compacted. The drawback of this approach is that once compacted, the inner loops can not be exposed to scheduling.

In this paper, we extend the approach in [15] to the n dimensional case to expose fine-grain parallelism for nested loops whether they are perfectly nested or not. Finding an optimal schedule for the n dimensional case is an open problem. For some cases, we can easily find optimal schedules. Figure 2 shows such an example. In the figure, the dependence edges are annotated with the dependence distance vectors.[2] Thus, (0,0) means an in-loop (not loop-carried) dependence, (0,1) means the dependence carried by the i2 EMBED Equation loop, etc. The as-soon-as- possible schedule of this loop after some number of 2-dimensional unwinding is given in Figure 2(b). We observe that the delay of statements A and B along the i1 dimension is always 2, while along the i2 dimension it is always 1. For example, A(i1,i2) can be executed only 1 cycle after A(i1,i2-1) and 2 cycles after A(i1-1,i2). A similar argument applies for statement B. Because of this regularity, we can parallelize the original loop as in Figure 2(c). The parallelized loop is exposing all statement level parallelism in this loop. The readers can verify this by following its execution several steps. At each step, the loop correctly executes all statements that can be executed in parallel.

However, in general, when we schedule all the statement instances as soon as possible, we do not necessarily see a fixed delay pattern emerge as in the previous example. One example is given in Figure 3. The delay pattern is as follows.

$$d_1(i_1, i_2) = \begin{cases} 2 & \text{if } i_1 < i_2 \\ 1 & \text{if } i_1 \geq i_2 \end{cases}$$

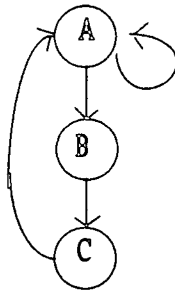
$$d_2(i_1, i_2) = \begin{cases} 2 & \text{if } i_1 > i_2 \\ 1 & \text{if } i_1 \leq i_2 \end{cases}$$

In above, $d_1(i_1, i_2)$ is the delay along the first dimension, I_1 dimension, at iteration (i_1, i_2), and $d_2(i_1, i_2)$ the delay along the second dimension, I_2 dimension, at iteration (i_1, i_2). For example, the delay between $A(i_1, i_2)$ and $A(i_1+1, i_2)$ is represented by $d_1(i_1, i_2)$. Similarly, the delay between $A(i_1, i_2)$ and $A(i_1, i_2+1)$ is represented by $d_2(i_1, i_2)$. For both dimensions, the delays are not constant; they could

1) Throughout this paper, "schedule" means a static reordering of the statements by the compiler. This schedule is then executeed as is, i.e. with the order of the statements in the schedule being preserved.

2) We follow the standard definitions of dependence distance vector as in [21].

be 1 or 2 depending on the values of i_1 and i_2. It is an open problem whether we can optimally parallelize such loops whose delays are not constants but functions of index variables. Furthermore, as will be shown in Section 3.2, an optimal solution would require knowing the exact bound of all the loops at compile time; since this information is not usually available, an optimal solution is, in general, only of theoretical interest. Instead of trying to parallelize loops optimally for all cases of delay patterns, we simply force the delays to be constant, and compute the parallel form based on these delays.



(a) A dependence graph for a 2-dimensional loop for which an optimal schedule is hard to find.

0	A00									
1	B00	A01		A10						
2	C00	B01	A02	B10				A20		
3		C01	B02 ···	C10	A11			B20		
4			C02 ···		B11	A12		C20	A21	
5			···		C11	B12	A13		B21	
6						C12	B13 ···		C21	A22
7							C13 ···			B22
8							···			C22
9										···
10										···

(b) as-soon-as-possible-schedule for the loop in Figure 3(a)

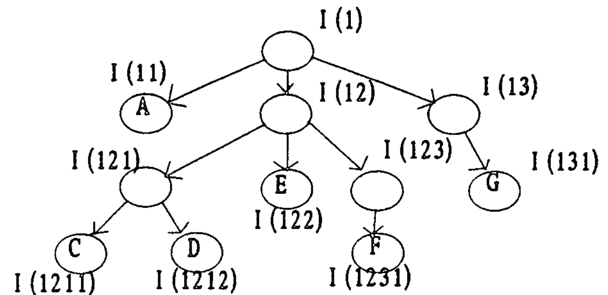
Fig. 3. A 2-dimensional loop for which optimal delays are not constant.

III. Definitions

Before going into the details of our method, we need to define a few terms. We will use a tree, called loop tree, to capture the structure of a nested loop.[3] In this tree, each node corresponds to a loop in the nested loop. The index of each node, then, is represented by the path from the root to the corresponding node. We represent the index of the outermost loop (the root node) by I_1 (we assume there is only one outermost loop); therefore, the i-th child loop of this outermost loop has index I_{1i}, and the j-th child loop of this child loop has index I_{1ij}, and so on. For example, the nested loop in Figure 4(a) will have the loop tree in Figure 4(b). In the figure, the index of each loop is shown next to the corresponding node.

```
For I1I1 = 0 To N1-1
  A
  For I12 = 0 To N12-1
    For I121 = 0 To N121-1
      C
      D
    Endfor
    E
    For I123 = o To N123-1
      F
    Endfor
  Endfor
  For I13 = 0 To N13-1
    G
  Endfor
Endfor
```

(a) An example of a non-perfectly nested loop



(b) A loop tree for the loop in Figure 4(a)

Fig. 4. An example of non-perfectly nested loop and its loop tree.

As can be noted from the above example, we regard a simple statement as a loop with a single iteration. Therefore, the outermost loop, whose index is presented by I_1, has three children whose indices are I_{11}, I_{12} and I_{13}. The first child is a statement; the other two, loops. Also, we assume all loops are normalized such that the lower bounds are always zero's. The upper bounds are represented by $N_{path}-1$, where path shows the position of the corresponding loop in the loop tree, as in loop indices. Note that all leaf nodes in the loop

3) We note that the structure of the loop tree is similar to the control dependence graph [2]. The difference lies in index notation. The change is necessary to accommodate non-perfectly nested loops.

tree are simple statements.

For each node in the loop tree, we define three values: H_{path}, d_{path}, and S_{path}. H_{path} is the number of children of node I_{path}(or loop I_{path}).[4] The second value, d_{path}, represents the delay, or interval, between any two adjacent iterations of loop I_{path}. We will call this value the delay of loop I_{path}. This value corresponds to the skewing factor in loop transformation[9] or delay in DOACROSS[17]. Finally, S_{path} is the size of loop I_{path}, which is defined as,

$$S_{path} = \begin{cases} (N_{path}-1)d_{path}+S_{1,p1,\cdots ps,1}+\cdots+S_{1,p1,\cdots,ps,H_{path}} & \text{for a true loop} \\ 1 & \text{for a statemet} \end{cases}$$

where $path = (1, p1, \cdots, ps)$

Intuitively, S_{path} is the number of parallel statements in loop I_{path}. For example, if all statements in loop I_{path} can be executed in parallel, S_{path} is 1. Note that

$$S_{1,p1,\cdots,ps,1}+\cdots+S_{1,p1,\cdots,ps,H_{path}}$$

is the sum of the sizes of all the children of loop I_{path}. We will use a short-hand representation $S_{1,p1,\cdots,ps,*}$ for this. For example, the size of loop I_{1ij} is

$$S_{1ij} = (N_{1ij}-1)d_{1ij}+S_{1ij*}$$

These values are needed to transform the loop correctly. A node in the loop tree is executed the number of times dictated by the upper bounds of its predecessors. For example, loop I_{121} is repeated $N_1 \times N_{12}$ times. The partial iteration vector shows which copy is active; that is, it shows the index values of the currently active surrounding loops. Therefore, the instance of loop I_{121} at partial iteration vector (iv_1, iv_{12}) is its copy when $I_1 = iv_1$, and $I_{12} = iv_{12}$.

III. Method

Our method consists of three steps: shaping, delay computing, and transformation. In shaping, we reorder the statements to maximize the overall parallelism. In the delay computing step, we compute the delays of all participating loop nests. In the transformation step, we parallelize the loop. Each step will be explained in following sections.

1. Shaping

Shaping is applied to a set of statements within the same loop nest. In perfectly nested loops, the statements in the innermost loop will become the target code. In non-perfectly nested loops, several sets of statements that are surrounded by the same set of loop nests will become the target code. The goal of shaping is to mazimize parallelism in two ways: (1) explicitly exposing parallelism between the statements involved, and (2) reordering statements to minimize overall delays (that have to be introduced to preserve data dependence) across the loop nests.

one simple way to perform shaping is to take the original order of the statements as it is. Let's call this the original-order method. A variation of this method is to take the set of statements and schedule it as soon as possible, ignoring all loop-carried dependences. The resulting ordering will be used. This will be referred to as the compacted-order method. Neither of them is satisfactory. We take an approach derived from Perfect Pipelining and Loop Quantization. We unwind the loop a finite number of times for each dimension, schedule all statement instances in it as soon as possible, and take the order of the statements when a repeating pattern is found. Such a pattern will often occur naturally, and in those cases it in fact expresses near-to-optimal parallelism in the loop; when the pattern does not occur on its own, we simply force it heuristically after a finite number of iterations. This method will be referred to as scheduled-order method.

Figure 5 shows a sample loop and the three different orderings.(a) shows the original-order and the resulting delays, (b) the compacted-order and its delays, and (c) the scheduled-order and its delays. (c) shows the smallest delays. The dependence distance vectors[5] between statements are shown in (a); they should be same for (b) and (c). The sample loop is two-dimensional and has three statements (A, B, and C). In the figure, we calculated the delay of each loop for each case (see Section 2 for the delay computation). d1 is the delay of the outer loop, and d11 that of the inner loop.

The merit of the scheduled-order method is that it rearranges the statements to expose hidden parallelism across dimensions. For example, in Figure 5, we get limited parallelism using the first two methods (the successive iterations of the inner or outer loops cannot be fully overlapped in both cases because of the loop carried dependences), while using the third method we can make the first dimension arbitrarily parallelizable. That is, the successive iterations of the inner loop in this case can be fully overlapped.[6] Since the methods are heuristics, we wanted to test the robustness of their performance under unbiased conditions. Thus we have generated 20 random loops, and calculated three different delay sets for each loop. The three delay sets correspond to the above three cases. The loops we have generated contain 50 nodes and 100 edges with nest depth 3. One half of the edges are in-loop dependences, with the remaining dependences being loop

4) We will use node and loop interchangeably throughout the paper.

5) Again, we follow the standard definition of dependence distance vector as in [21].

6) This fact is represented by d_{11}=0, which means no delay is needed amoong iterations of loop I_{11}. See Section 3.2 for the computation of delay values.

carried. The edges are generated randomly. In the case of the third method (scheduled-order method), the patterns were found within 10 iterations for all 20 loops.

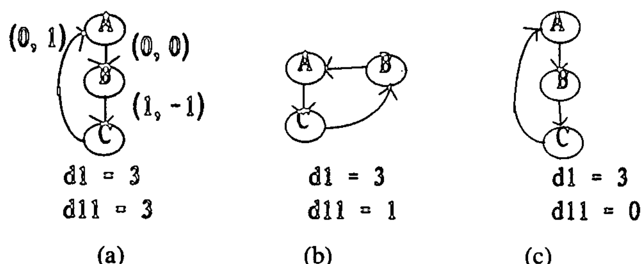


Fig. 5. Three different ordering methods.

The results are presented in Figure 6. For loops 3, 4, 5, 12, and 18, the third method detects arbitrary parallelism in some dimensions, while the other two do not. For example, in loop 3, the scheduled-order method produces d1 = 0 meaning that loop 3 is completely vectorizable for I1 loop, while the other two methods produce non-zero delays for I1 loop. Computing the total sum of delays for each method for each loop, we find the third method produces smaller sums than the other two for all loops except loop 6. For loop 6, the sum of delays obtained by the third method is much smaller than the sum of delays of the first method but slightly larger than that of the second method.[7)]

loop	original			compacted			scheduled		
	d1	d11	d111	d1	d11	d111	d1	d11	d111
1	16	26	33	2	10	4	3	5	4
2	8	22	29	4	10	11	4	10	11
3	38	32	18	8	8	10	0	8	8
4	22	33	16	15	3	19	14	0	13
5	17	9	25	3	1	9	3	0	4
6	16	20	18	1	6	8	4	4	9
7	21	21	12	15	14	14	15	2	10
8	20	9	18	6	3	5	6	2	2
9	19	0	32	7	0	8	3	0	5
10	27	30	9	10	15	9	1	8	6
11	7	25	17	1	7	13	3	6	8
12	29	19	34	15	13	16	0	12	16
13	23	9	17	9	14	12	7	1	12
14	11	13	10	4	9	4	2	9	3
15	20	29	22	4	11	3	3	11	3
16	9	18	9	4	4	5	4	2	5
17	27	25	30	4	10	7	1	10	5
18	15	15	23	1	1	5	0	1	4
19	33	27	34	21	17	23	10	17	22
20	31	20	25	3	7	8	5	1	7

Fig. 6. Comparison of the three different ordering methods.

7) This indicates that the scheduled-order method does not necessarily produce optimal delays even though a pattern occured naturally during the unwinding and scheduling proecss.

2. Delay Computing

This step, as we mentioned before, exactly corresponds to the delay computing process in DOACROSS, or the skewing factor computing process in loop transformation. In general, computing the optimal delay set is an integer programming problem [6]. The problem can be formulated as follows.

Compute delay set $(d_1, d_2, codts, d_y)$ such that it minimizes the total execution time and satisfies the following inequalities.

$$(v_{11}d_1 + v_{12}d_2 + \cdots + v_{1y}d_y) \geq c_1$$
$$(v_{21}d_1 + v_{22}d_2 + \cdots + v_{2y}d_y) \geq c_2$$
$$\cdots\cdots\cdots$$
$$(v_{x1}d_1 + v_{x2}d_2 + \cdots + v_{xy}d_y) \geq c_x$$

where y is the cardinality of the delay set, and x is the number of dependence distance vectors.

In the above, $v_{i1}, v_{i2}, \cdots, v_{iy}$ is the i-th dependence distance vector, and c_i is some constant. Note that to simplify the notations we gave each delay a serial number; so, the subscripts of d here do not represent the path as they did in Section II.

One interesting observation is that if one of the columns (denoted by v), say the j-th column, contains only positive elements (greater than 0), then we can solve the system simply by putting $d_k = 0$ for all $k \neq j$, and putting d_j = "some integer that satisfies the remaining system". Based on this observation, plus the fact that the first non-zero element of a legal dependence distance vector should be positive, we can suggest a delay computing algorithm which is much simpler but still more effective compared to previous approaches.

Basically, the algorithm works in a divide-and-conquer manner. The inequalities are divided into two groups: those whose leading coefficients are positive -- call this the first group; and those whose leading coefficients are zero's -- call this the second group. The second group is solved first. Note that this problem has at least one fewer variables than the original one. The delay corresponding to the leading coefficient is missing here. Assume the second group is solved, which means we know all the values of the delays except the one corresponding to the leading coefficient. We substitute these delay values to the first group of inequalities (whose leading coefficients are positive), and compute the missing delay. The solving process of the second group is a recursive application of the same divide-and-conquer strategy. Please refer to [18] for the details of this algorithm.

3. Transformation

The third step is transforming the loop tree into a parallel loop tree according to the delays computed in the previous section. All nodes in the loop tree, except the root and leaf

nodes, are transformed into parallel form as follows. Suppose we want to transform node I_{1ij} into parallel form. Assume it has three children. Then, the I_{1ij} loop below,

```
.....
   FOR  I1ij = 0 TO  N1ij - 1
      FOR I1ij1 = 0 TO  N1ij1 - 1
         .....
      ENDFOR
      FOR I1ij2 = 0 TO  N1ij2 - 1
         .....
      ENDFOR
      FOR I1ij3 = 0 TO  N1ij3 - 1
         .....
      ENDFOR
   ENDFOR
.....
.....
```

will be transformed into

```
.....
   FORALL  I1ij = L1ij TO  U1ij
      CASE  t1ij - I1ij d1ij IS
      0 TO  S1ij1 - 1:  FOR I1ij1 = 0 To N1ij1 - 1
                   .....
                ENDFOR
       S1ij1 To  S1ij1 + S1ij2 - 1:  For  Iij2 = 0, Nij2 - 1
                      ...
                   Endfor
       S1ij1 +1S1ij2 To  S1ij1 + S1ij2 + Sij3 - 1:  For  Iij3 = 0, Nij3 - 1
                            ...
                          Endfor
      ENDCASE
   ENDFORALL
.....
.....
```

Note that the inner loops of loop I_{1ij}, loop I_{1ij1}, I_{1ij2}, and I_{1ij3}, are not parallelized yet. They can be parallelized by applying the same process recursively. S_{path} is the size of the loop I_{path} as explained in Section II. L_{1ij} and U_{1ij} are the new loop bounds. The value of t_{1ij} is computed by

$$t_{1ij} = t - TB(I_{1ij}, (iv_1, iv_{1i}))$$

where (iv_1, iv_{1i}) is the partial iteration vector of loop I_{1ij} (see Section II for the definition of a partial iteration vector).

TB is the starting time step of its argument loop at the designated partial iteration vector, whose computation will be described shortly, while t is the global time step which represents the time elapsed since the entire loop began execution. t_{1ij} represents the time elapsed since the instance of loop I_{1ij} at partial iteration vector (iv_1, iv_{1i}) began, so we call it the local time step of loop I_{1ij}.

The computation of TB proceeds as follows. Assume we want to compute the TB for loop I_{1ij} at a partial iteration vector (iv_1, iv_{1i}). If we draw the surrounding loops of loop I_{1ij}, we get

```
For I1 = 0, N1 - 1
  For I11 = 0, N11 - 1
    ...
  Endfor
  ...
  For I1i = o, N1i - 1
     For I1i1 = 0, N1i1 - 1
       ...
     Endfor
     ...
     For  I1ij = 0, N1ij - 1
       ...
     Endfor
     ...
  Endfor
  ...
Endfor.
```

We want to calculate the starting time step of the instance of loop I_{1ij} when $I_1 = iv_1$ and $I_{1i} = iv_{1i}$. Since each iterations of I_1 is delayed by d_1, the iv_1-th iteration will start at the $iv_1 d_1$ time step. (Note the iteration count starts from zero.) At the iv_1-th iteration, we have to wait until all previous loops before I_{1i} are executed. Therefore, S_{11} + *codts* + $S_{1,i-1}$ time steps should pass. At this point, we again have to wait for the iv_{1i}-th iteration of I_{1i} loop. This adds $iv_{1i} d_{1i}$ time steps to the delay time accumulated so far. Finally, we have to wait until all the previous loops before I_{1ij} loop at the iv_{1i}-th iteration are executed. So, the starting time step of the desired instance of loop I_{1ij} is

$$TB(I_{1ij}, (iv_1, iv_{1i})) = iv_1 d_1 + S_{11} + \cdots + S_{1,i-1} + iv_{1i} d_{1i} + S_{1i1} + \cdots + S_{1,i_{i-1}}$$

The last variables we need to compute are L_{1ij} and U_{1ij}, the new loop bounds. L_{1ij} is the iteration that spans t_{1ij}, the local time step of the current instance of loop I_{1ij}, for the first time. U_{1ij} is the last iteration that spans t_{1ij}. Therefore, if $L_{1ij} > 0$, the ending time step[8] of the iteration $L_{1ij} - 1$ should be strictly less than t_{1ij}, and the ending time step of the iteration L_{1ij} should be greater than or equal to t_{1ij}. Also if $U_{1ij} < N_{1ij} - 1$, the starting time step of U_{1ij} should be greater than or equal to t_{1ij}, while that of $U_{1ij} + 1$ should be strictly greater than t_{1ij}. Therefore, when $L_{1ij} > 0$ and $U_{1ij} < N_{1ij} - 1$, we get the following inequalities to be satisfied.

---

8) Actually local ending time step. We are looking at only the current instance of loop I_{1ij}. Every time step here, while we are explaining the computation of L_{1ij} and U_{1ij}, refers to the local time step of the current instance of loop I_{1ij}.

$$(L_{1ij}-1)d_{1ij}+S_{1ij*} < t_{1ij} \le L_{1ij}d_{1ij}+S_{ij*}$$

$$U_{1ij}d_{1ij} \le t_{1ij} < (U_{1ij}+1)d_{1ij}$$

Solving these with the constraints that L_{1ij} is an integer greater than or equal to zero, and that U_{1ij} is an integer less than or equal to $N_{1ij}-1$, we get

$$L_{1ij} = MAX(0, \lceil (t_{1ij}+1-S_{1ij*})/d_{1ij} \rceil),$$

$$U_{1ij} = MIN(N_{1ij}-1, \lfloor t_{1ij}/d_{1ij} \rfloor)$$

Now, the same parallelization process can be repeated for all the intermediate nodes. The parallelization of the leaf nodes is simple: just leave them untouched. For the root node (the outermost loop), we parallelize it following the above process, but this time add another loop on top of it. The new outermost loop is a sequential loop, and its index is the global time step. At each global time step, the sequential outermost loop specifies which statements of which loops can be executed in parallel.

The general formula for TB, L, U, and S is in Figure 7. The derivations of the first three are straightforward from the above explanations, and that of the last is borrowed from Section II. Note that TB(I_1) = 0, which is the case when the path $p_1, p_2, codts, p_{x-1}, p_x$ is nil, because by definition, it is the starting time step of the outermost loop. For completeness, we have include the formula for MAX_GLOBAL_TIME_STEP in the figure. MAX_GLOBAL_TIME_STEP is the time step of the last statement executed, or 1 time step less than the size of the root node.

$$TB(I_{1,p1,\cdots,px-1,px}(iv_1, iv_{1,p1}, \cdots, iv_{1,p1,p2,\cdots,px-1})) = \begin{cases} o \text{ for the root node} \\ \alpha \text{ for other nodes,} \end{cases}$$

Where

$$\begin{aligned}\alpha = {} & iv_1 d_1 + S_{11} + S_{12} + \cdots + S_{1,p_1,p_2-1} \\ & iv_{1,p_1} d_{1,p} + S_{1,p_1,1} + S_{1,p_1,2} + codts + S_{1,p_1,\cdots,p_{x-1},p-1} \\ & + \cdots \\ & + iv_{1,p_1,\cdots,p_{x-1}} d_{1,p_1,\cdots,p_{x-1}} + S_{1,p_1,\cdots,p_{x-1},1} + \cdots + S_{1,p_1,codts,p_{x-1},p_x-1}\end{aligned}$$

$$L_{1,p_1,\cdots,p} = \begin{cases} MAX(0, \lceil (t_{1,p_1,\cdots,p_x}+1-S_{1,p_1,\cdots,p})/d1,p_1,\cdots,p_x \rceil) \text{if } d_{1,p_1,\cdots,p} > 0 \\ 0 \text{ if } d_{1,p_1,\cdots,p} = 0 \end{cases}$$

$$U_{1,p_1,codts,p} = \begin{cases} MIN(N_{1,p_1,\cdots,p_x}-1, \lfloor t_{1,p1,\cdots,p}/d_{1,p_1,\cdots,p_x} \rfloor) \text{if } d_{1,p_1,\cdots,p_x} > 0 \\ N_{1,p_1,\cdots,p} \text{ if } d_{1,p_1,\cdots,p_x} \end{cases}$$

$$S_{1,p,\cdots,p_x} = \begin{cases} (N_{1,p,\cdots,p}-1)d_{1,p,\cdots,p}+S_{1,p,\cdots,p,1}+\cdots+S_{1,p,\cdots,p,1} \text{if } I_{1,p_1,\cdots,p} \text{ is a loop} \\ 1 \text{ if } I_{1,p_1,\cdots,p} \text{ is a statement} \end{cases}$$

MAX_GLOBAL_TIME_STEP = S_1-1

Fig. 7. Transformation formula.

IV. Examples

An example is shown in Figure 8. To make the explanation clear, we concentrate on the transformation step. We will take the original order of the statements as it is. Figure 8(a) show the source code and its dependence graph. Statement A has a self dependence with distance 3 at loop I_1, and statement G has another self dependence with distance 0 at loop I_1 and distance 1 at loop I_{13}. From this dependence, we can easily compute that $d_1=3$, $d_{13}=1$, and all other delays are zero's. Figure 8(b) shows the template of the parallel form.

First, let's compute all S_{path} values. Since a single statement has a size of 1, $S_{11}=S_{1211}=S_{1212}=S_{122}=S_{1231}=S_{131}=1$. And at the next level, $S_{121}=(N_{121}-1)d_{121}+2$, $S_{123}=(N_{123}-1)d_{123}+1$, and $S_{13}=(N_{13}-1)d_{13}+1$. Based on these values,

$$S_{12} = (N-12-1)d_{12}+(N_{121}-1)d_{121}+2+1+(N_{123}-1)d_{123}+1$$

and finally

$$\begin{aligned}S_1 &= (N_1-1)d_1+1+(N_{12}-1)d_{12}+(N_{121}-1)d_{121}+2+1 \\ &\quad +(N_{123}-1)d_{123}+1+(N_{13}-1)d_{13}+1 \\ &= (N_1-1)d_1+(N-12-1)d_{12}+(N_{121}-1)d_{121} \\ &\quad +(N_{123}-1)d_{123}+(N_{13}-1)d_{13}+6\end{aligned}$$

Then,

$$MAX_GLOBAL_TIME_STEP = (N_1-1)3+(N_{13}-1)+6-1 = 3N_1+N_{13}+1$$

$$t_1 = t - TB(I_1) = t,$$

$$L_1 = MAX(0, \lceil (t+1-S_{1*})/3 \rceil) = MAX(0, \lceil (t-N_{13}-4)1/13 \rceil),$$

$$U_1 = MIN(N_1-1, \lfloor t/3 \rfloor)m$$

$$t_{12} = t - TB(I_{12},(I_1)) = t-(I_1 d_1+S_{11}) = t-3I_1-1,$$

$$L_{12} = 0, U_{12} = N-12-1,$$

$$t_{121} = t_{TB}(I_{121},(I_1,I_{12})) = t-3I_1-1, L_{121}=0, U_{121}-1,$$

$$t_{123} = t-TB(I_{123},(I_1,I_{12})) = t-(I_1 d_1+S_{11}+I_{12}d_{12}+S_{121}+S_{122}) = t-3I_1-4$$

$$L_{123} = 0, U_{123} = N_{123}-1,$$

$$t_{13} = t-TB(I_{13},(I_1)) = t-(I_1 d_1+S_{11}+S_{12}) = t-(3I_1+1+4) = t-3I_1-5,$$

$$L_{13} = MAX(o, \lceil (t-3I_1-5+1-S_{1,3,*})/1 \rceil) = MAX(0, t-3I_1, -5),$$

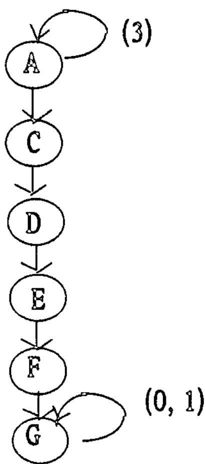
$$U_{13} = MIN(N_1-1, t-3I_1-5).$$

Therefore, the final parallel loop is as shown in Figure 9.

Another example is given in Figure 10(a)-(b). The algorithm is the well known SOR (Successive Over-Relaxation) algorithm as presented in a standard cookbook for numerical applications[19]. The parallel form obtained by our technique is given in Figure 10(b). Note that we have regarded the statements at line 1 through 6 as a single statement and the innermost loop at line 8 to 13 as another single statement. By increasing the granulity in this way, we can adjust our method for coarser-grain parallel machines. In fact, the parallel form was run on the Sequent machine, with an almost linear speed-up over sequential version (see Section 5).

Before we leave this section, we demonstrate the difference of our schedule against others even when the loops are perfectly nested. Figure 11(a) and (b) show an example of a perfectly nested loop and its dependence graph. Figure 11(c) shows a schedule that would be produced by Loop

Transformation [9], while Figure 11(d) shows what our technique would produce. In Loop Transformation, which can be thought as one of the representative parallelization techniques for perfectly nested loops, the innermost loop is skewed against all the outer loops by some computed value, and move all the way to the outermost position through "loop permutation". In our example, the J loop is skewed by 1 against the I loop and interchanged with the I loop to produce the schedule in Figure 11(c). The schedule in Figure 11(d) is produced first by computing the shape of the loop body as shown in the figure, then overlapping it with one cycle delay along I and J dimension. The total number of cycles needed to complete the loop is 5 in our case, while it is 15 in Loop Transformation. Our schedule finishes ealier because it schedules at the statement level, while the Loop Transformation parallelizes at the iteration level as we mentioned before.




```
For I1 = 0, N1 - 1
    A: A[I1 + 3] = A[I1] + 1
    For I12 = 0, N12 - 1
        For I121 = 0, N121 - 1
            C: C[I121] = A[I1 + 3]
            D: D[I121] = C[I121]
        Endfor
        E: E[I12] = D[N121 - 1]
        For I123 = 0, N123 - 1
            F: F[I123] = E[I12]
        Endfor
    Endfor
    For I13 = 0, N13 - 1
        G: G[I13 + 1] = G[I13] + F[0]
    Endfor
Endfor
```

(a) A non-perfectly nested loop and its dependence graph

```
For t = 0, MAX_GLOBAL_TIME_STEP
  Forall I1 = L1, U1
    Case t1 - I1d1 is
    0 :
    1 to S12 : Forall I12 = L12, U12
                 Case t12 - I12d12 is
                 0 to S121 - 1 : Forall I121 = L121, U121
                                   Case t121 - I121d121 is
                                   0 : C
                                   1 : D
                                   Endcase
                                 Endforall
                 S121 S121   : E
                 S121 + 1 S121 + 1 to S121 + S123 : Forall I123 = L123, U123
                                                      F
                                                    Endforall
                 Endcase
               Endforall
    S12 + 1 to S12 + S13   : Forall I13 = L13, U13
                               G
                             Endforall
    Endcase
  Endforall
Endforall
```

(b) The template of parallel code for the loop in Figure 8(a)

Fig. 8. A parallelization example.

```
For t = 0, 3 3N1 + N13 + 1
  Forall I1 = MAX(0, ⌈(t - N13 - 4)/3⌉), MIN(N - 1 - 1, ⌊t/3⌋)
    Case t - 3I1 is
    0:A
    1 to 4:
      Forall I12 = o, N12 - 1
        Case t - 3I1 - 1 is
        0 to 1: Forall I121 = o, N121 - 1
                  Case t121 - I121d121 is
                  0:C
                  1:D
                  Endcase
                Endforall
        2:      E
        3 to 4: Forall I123 = 0, N121 - 1
                  F
                Endforall
        Endcase
      Endforall
    5 to 4 + N13 :
      Forall I13 = MAX(0, t - 3I1 - 5), MIN(N1 - 1, t - 3I1 - 5)
        G
      Endforall
    Endcase
  Endforall
Endfor
```

Fig. 9. The parallel form for the loop in Figure 8(a).

V. Experiments and results

Seven numerical algorithms were selected for the experiment. Five of them were non-perfectly nested; the other two perfectly nested. The first six algorithms were taken from [20]; the last was taken from [19]. 2d_bnd_val computes the transverse deflections of a simply-supported rectangular plate (program 8.4 in [20]); comp_sim_eq solves a system of linear simultaneous equations whose coefficients and constant vector may be complex (program 9.5 in [20]); eig_val computes the largest eigenvalue of a matrix by iteration (program 5.5 in [20]); laplace computes the solution of Laplace's equation by iteration (program 8.1 in [20]); simpson evaluates the integral of a function using Simipson's 1/3 rule (program 9.9-A in [20]); sub_comeqs computes a check vector by substituting the solution vector into the

original equations (program 9.6 in [20]).

The seven algorithms were parallelized using our method and run on a Sequent machine. The results are in Figure 12(a) and (b). The speed-up is computed as the ratio of the runtime of the transformed code divided by the runtime of the original sequential code running on one processor. The speed-up is less than 1 when a single pe (processing element) is used to run the transformed code because of the overhead of the transformed code. However, as the number of pe's increases, the speed-up improves considerably. The average speed-up over the original sequential program as the number of pe's varies from 1 to 5 is 0.8, 1.7, 2.4, 3.1, and 3.8.

```
0:For N=0,MAXITS-1
1:  . anorm[N+1]=0
2:    If N+1 = 1 then
3:        omega[N+2] =   1/(1-(1/2)ρ²)
4:    Else
5:        omega[N+2] =   1/(1-(1/4)ρ²omega[ N+1] )
6:    Endif
7:    For J = 0, JMAX - 3
8:        For L = 0, JMAX - 3
9:            If MOD(J+L+4,2)=MOD(N+1,2) then
10:               anorm[N+1]=anorm[N+1]+ABS(U(J+3,L+2)+U(J+1,L+2)
                      +U(J+2,L+3) +U(J+2,L+1)-4U(J+2,L+2)-F(J+2,L+2))
11:               U(J+2,L+2)=U(J+2,L+2)+omega[N+1]*(U(J+3,L+2)
                      +U(J+1,L+2)+U(J+2,L+3)+U(J+2,L+1)
                      -4U(J+2,L+2)-F(J+2,L+2))/4
12:           Endif
13:       Endfor
14:   Endfor
15:   If (N+1 > 1 and anorm[N+1] < eps * anormf) then RETURN
16:Endfor
```

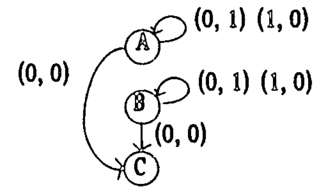
(a) SOR algorithm. The code was taken from [19].

```
For t = 0, 2*MAXITS + JMAX - 3
   Forall I1 = MAX(0,⌈(t+1-JMAX)/2⌉), MIN(MAXITS-1,⌊t/2⌋)
      Case t-2I1 is
      0:anorm[ I1+2]=0
         If I1+1 = 1 then
            omega[ I1+2] = 1/(1-(1/2)ρ²)
         Else
            omega[ I1+2] = 1/(1-(1.4)ρ²omega[ I1+1] )
         Endif
      1 to JMAX - 2:
         I12 = t-2I-1-1
         For L=0, JAMX - 3
            If MOD(I12+L+4,2) = MOD(I1+1,2) then
               anorm[I1+1]=anorm[I1+1]+ABS(U( I12+3,L+2)
               +U( I12+1,L+2) +U( I12+2,L+3)+(U( I12+2,L+1)
               -4U( I12+2,L+2)-F( I12+2,L+2)) U( I12+2,L+2)
               =U( I12+2,L+2)+omega[I1+1](U( I12+3,L+2)
               +U( I12+1,L+2) +U( I12+2,L+3)+U( I12+2,L+1)
               -4U( I12+2,L+2)-F( I12+2,L+2))/4
            Endif
         Endfor
      JMAX -1 :
            If (I1+1>1 and anorm[I1+1]<eps * anormf) then RETURN
      Endcase
   Endforall
Endfor
```

(b) Parallel code for Figure 10(a).

Fig. 10. Parallelization of SOR algorithm.

```
For I = 1, 3
   For J = 1, 3
      A
      B
      C
   EndFor
Endfor
```



(a) Source code (b) Dependence graph

cycle	execution schedule		
1	A11		
2	B11		
3	C11		
4	A12	A21	
5	B12	B21	
6	C12	C21	
7	A13	A22	A31
8	B13	B22	B31
9	C13	C22	C31
10		A23	A32
11		B23	B32
12		C23	C32
13			A33
14			B33
15			C33

(c) Schedule by Loop Transformation

cyc	execution schedule											
1	A11	B11										
2	C11		A12	B12			A21	B21				
3			C12		A13	B13	C1		A22	B22		
4					C13				C22		A23	B23
5											C23	

(d) Schedule by our technique

Fig. 11. Comparison of our technique with Loop Transformation for a perfectly-nested loop.

algorithms	seq. time	parallel time				
		1 pe	2 pe	3 pe	4 pe	5 pe
2d-bnd-val	13.8	15.8	8.1	6.6	5.3	3.9
comp-sim-eq	31.9	33.2	16.6	11.1	8.9	6.9
eig-val	19.8	19.8	9.9	6.7	5.1	4.3
alplace	30.5	47.5	26.3	17.9	13.7	11.3
simpson	52.8	91.0	45.5	33.4	23.6	21.3
sub-comeqs	10.5	10.9	5.5	3.6	2.7	2.2
sor	310.9	331.6	176.4	115.1	92.7	72.3

(a) Comparison of sequential and parallel execution time (in seconds) for the benchmark programs on Sequent machine.

algorithms	speed-ups				
	1 pe	2 pe	3 pe	4 pe	5 pe
2d-bnd-val	0.9	1.7	2.1	2.6	3.5
comp-sim-eq	0.9	1.9	2.9	3.6	4.6
eig-val	1.0	2.0	2.9	3.9	4.6
laplace	0.6	1.2	1.7	2.2	2.7
simpson	0.6	1.2	1.6	2.2	2.5
sub-comeqs	0.9	1.9	2.9	3.8	4.7
sor	0.9	1.8	2.7	3.4	4.3

(b) Speed-ups for the benchmark programs when parallelized by our technique

Fig. 12. Summary of results.

VII. Conclusion

In this paper, we have presented our approach to extend fine-grain parallelization techniques to non-perfectly nested loops. Previously fine-grain approaches were mainly limited to single-nested loops. Our technique parallelizes nested loops at the fine-grain level whether they are perfectly nested or non-perfecly nested. There exist techniques to handle nested loops at iteration level, however our method differs from them in that it works at statement level exposing more fine-grain parallelism and handles non-perfectly nested loops efficiently.

References

[1] Kennedy, K., "Automatic Translation of Fortran Programs to Vector Form," Rice Technical Report 476-029-4, Rice University, Houston, Oct. 1980.

[2] Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure,B., and Wolfe, M., "Dependence Graphs and Compiler Optimization," Proc. of the 8th ACM symp. on Programming Languages, Williamsburg, VA, pp. 207-218, Jan. 1981.

[3] Wolfe, M.J., "Oprimizing Supercompilers for Supercomputers," Ph.D. Thesis, Univ. of Ill. at Urbana-Champaign, Dept. of Comp. Sci. Rpt. No. 82-1105, Oct. 1982.

[4] Allen, J.R. and Kennedy, K., "Automatic Translation of Fortran Programs to Vector Form," ACM Transactions on Programming Languages and Systems, Vol. 9, No. 4, pp. 491-542, Oct. 1987.

[5] Muraoka, Y., "Parallelism Exposure and Exploitation in Programs," Ph.D. Thesis, Univ. of Ill. at Urbana-Champaign, Dept. of Comp. Sci., Rpt. No. 71-424, Feb. 1971.

[6] Lamport, L., "The Parallel Execution of DO Loops," Comm. of the ACM, pp. 83-93, Feb. 1974.

[7] Kuhn, R.H., "Optimization and Interconnection Complexity for: Parallel Processors, Single-Stage networks, and Decision Trees," Ph.D. Thesis, Univ. of Ill. at Urbana-Champaign, Dept. of Comp. Sci. Rpt. No. 80-1009, Feb. 1980.

[8] Banerjee, U., "Unimodular Transformations of Double Loops," 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, California, August 1990.

[9] Lam, M. and Wolf, M., "Maximizing Parallelism Vis Linear Loop Transformations," 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, California, August 1990.

[10] Fisher, J.A., "The Optimization of Horizontal Microcode within and beyond Basic Blocks: an Application of Processor Scheduling with Resources," Ph.D. thesis, New York Univ., 1979.

[11] Nicolau, A., "Uniform Parallelism Exploitation in Ordinary Programs," Proc. International Conf. on Parallel Processing, August 1985.

[12] Lam, M., "A Systolic Array Optimizing Compiler," Ph.D. thesis, Carnegie Mellon Univ., 1987.

[13] Rau, B. and Fisher, J., "Instruction-Level Parallel Processing: History, Overvies, and Perspective," Journal of Supercomputing: Special Issue on Instruction-Level Parallelism, 7(1/2):9-50, 1993

[14] Nakatani, T. and Ebcioglu, K., "Making Compaction Based Parallelization affordable," IEEE transactions on Parallel and distributed Systems, pp. 1014-1529, Sept. 1993.

[15] Aiken, A. and Nicolau, A., "Perfect Pipelining: a new loop parallelization technique," Proc. of the 1988 European Symposium on Programming, pp 221-235, Springer Verlag Lecture Notes in Computer Science, No. 300, March 1988.

[16] Nicolau, A., "Loop Quantization or Unwinding Done Right," Proc. Supercomputing 1st International Conference, June 1987.

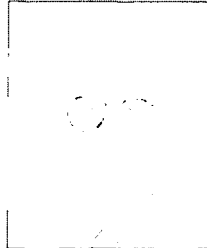
[17] Cytron, R.G., "Doacross: Beyond Vectorization for Multiprocessors," Proc. of the 1986 International Conf. on Parallel Processing, St. Charles, Ill, pp. 836-844, August 1986.

[18] Kim, K.C., "Vertical and Horizontal Program Parallelization Techniques," Ph.D. Thesis, Comp. Sci. Dept., Univ. of California, Irvine, 1992.

[19] Press, W.H., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T., "Numerical Recipes," pp. 647-659, Cambridge Univ. Press, 1986.

[20] McCormick, J.M. and Salvadori M.G., "Numerical

Methods in Fortran," Prentice Hall Inc., 1964
[21] Padua, D.A.H., "Multiprocessors: Discussions of some Theoretical and Practical Problems," Ph.D. thesis, Univ. of Ill. at Urbana-Champaign, Urbana, Ill., 1979.



Ki-chang Kim received the B.S. degree from the California State Polytechnic University at Pomona in 1986, and the M.S. and Ph.D. degrees from the University of California at Irvine in 1989 and 1992, respectively. All degrees are in computer science. Currently he is an assistant professor in the Inha University at Incheon, Korea. His research interests include parallel architectures, parallelizing compilers, and distributed processing.