JOURNAL OF KOREA MULTIMEDIA SOCIETY, VOL. 9, NO. 12, DECEMBER 2006(pp. 1700-1708)

Interprocedural Transformations for Parallel Computing

Doo-Soon Park†, Min-Hyung Choi††

ABSTRACT

Since the most program execution time is consumed in a loop structure, extracting parallelism from loop programs is critical for the faster program execution. In this paper, we proposed data dependency removal method for a single loop. The data dependency removal method can be applied to uniform and non-uniform data dependency distance in the single loop. Procedure calls parallelisms with only a single loop structure or procedure call most of other methods are concerned with the uniform code within the uniform data dependency distance. We also propose an algorithm, which can be applied to uniform, non-uniform, and complex data dependency distance among the multiple procedures. We compared our method with conventional methods using CRAY-T3E for the performance evaluation. The results show that the proposed algorithm is effective.

Keywords: Parallel Compiler, Parallel Computing, Interprocedural Transformation, Data Elimination

1. INTRODUCTION

There has been a move to parallel processing systems in order to build the faster computers. However, simply adding more processors is not sufficient enough. Many researchers have been suggested that new parallel programming environments for parallel computers. These environments analyze the dependency relationships of the variables being used in the program. When the source code is sequential, the parallelizing compilers for parallel computers detect the implicit parallelism and translate the sequential programs into the parallel programs. Examples of such parallelizing compilers are PARAFRASE II[1] from University of Illinois, PTRAN from IBM, and SUIF from Stanford University, among many others.

The most fundamental and usable part of the parallel compiler is the restructuring module, which extracts parallelisms from sequential loops. This method is fairly good for speeding up parallel processing system. We can classify existing loop transformation methods[2,3] into two categories: when the data dependency distance is uniform and when it is non-uniform. The uniform data dependency distance case includes interchanging[2], tiling[2], unimodular[4], selective cycle shrinking[5], Hollander[5], and Chen&Wang[6] methods. The non-uniform case includes DCH[7] and IDCH[8] methods. All of these methods analyze the data dependency, and divide them into pieces to schedule, but they have a limitation to achieve better performance. For this reason, we propose an algorithm, which can efficiently remove data dependency and be implemented in both uniform and non-uniform fashion in the single loop.

We expanded a loop procedure call into multiple loops. In uniform situations, there are many proposed methods. Those are loop extraction[9], loop embedding[10] and procedure cloning[11].

※ Corresponding Author : Doo-Soon Park, Address : (336-745) Sinchang-Myun, Asan-Si, Choongchungnam-Do, Korea, TEL : +82-41-530-1317, FAX : +82-41-530-1548, E-mail : parkds@sch.ac.kr
Receipt date : Aug. 9, 2006, Approval date : Oct. 16, 2006
† Division of Computer Science and Computer Engineering, SoonChunHyang University
†† Computer Science and Engineering, University of Colorado at Denver
(E-mail : Min.Choi@cudenver.edu.

However, no such a proposal has been made yet for the non-uniform However, no such a proposal has been made yet for the non-uniform situation case.

Also a method of extracting parallelism from a loop, which contains procedure calls, is proposed. We applied the proposed data dependency removal method to achieve the goal, and the method can be implemented in both uniform and non-uniform way. The proposed method is summarized in Figure 1.

To show that data dependency removal method is the most efficient in a one loop, we evaluated the performance on the data dependency distance for the case of uniform and non-uniform. The performance analysis on using data dependency removal method in the inter-procedure transformation method on proposed algorithm are performed using CRAY-T3E.

The rest of this paper is organized as follows: the data dependency removal algorithm is described in Section 2. We then propose an expanded data dependency removal algorithm in Section 3. Performance analysis is in Section 4, and conclusion in Section 5.

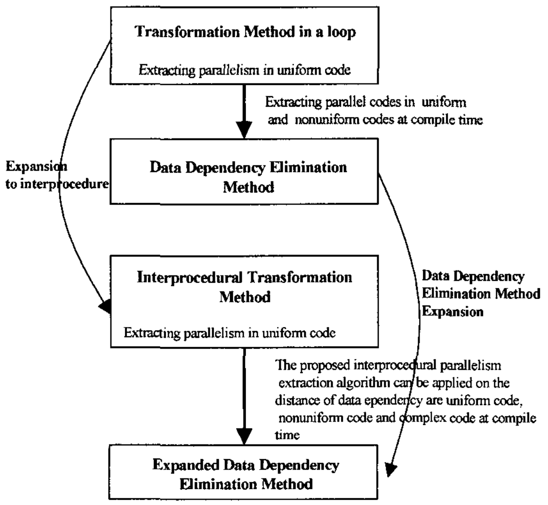


Fig. 1. Extended data dependency removal method.

2. THE DATA DEPENDENCY REMOVAL ALGORITHM

In this section, definitions and theorems for the data dependency removal algorithm are presented.

2.1 Data Dependency

The data dependency can be divided in the form of flow dependence, anti dependence, output dependence. Between sentence S_i and S_j, in S_i the variable X is defined and S_j uses X. In this situation the flow dependence exists while S_i runs before S_j. Two sentences S_i, and S_j define the same variable and S_i performs before S_j, and the output dependence exists. Sentence S_i uses X and S_j defines X. If S_i performs before S_j then there exists anti dependence. There are many approaches to analyze the data dependency and the easiest method of them is the separability test. This method can be implemented when only one common loop variable exists between two sentences. And GCD test has nothing to do with loop boundary and only subordinating equation tells whether there exist a fixed solution or not. Power test, I test, λ test can be used for data dependency analysis. But these methods can only be applied when the data dependency distance is uniforms and cannot be used for the non-uniform case.

According to the study on the added expression in array variable and data dependency, only 13.65% is uniform type and 86.35% is non-uniform. For that reason,to perform parallel process more efficiently not only uniform type but also non-uniform type loops must be implemented. If a loop consists of only one added variable as in $a \times i+b$, $c \times i+d$(a, b, c, d are fixed number and i is loop variable), and then in added sequence if a=c then, dependency distance is $|(a \times i \pm b)-(c \times i \pm d)|$, and there exists an uniform type. If $a \neq c$, a non-uniform type exists. If data dependency distance is the uniform type, there exists only one dependent distant. If it is a non-uniform type, there can be many more dependent distances.

[Definition 1]

Dependence Matrix(DM) becomes an array for calculating the data dependent distance by using all of the added markers in nested loops to express data dependency. All elements in DM are pairs and the elements in pairs are expression of dependent sentences in between. DM(k,1,m) means that the number of nested loops is k, the initial value of dependent array matrix count is 1, and m is the number of sentences in the loop.

If the number of nested loops is 2 and the number of sentences in loop is n, then the dependent matrix is DM(2,1,n), and it consists of followings:

$$DM(2,1,n)=\begin{pmatrix}(x_{11};y_{11}),(x_{12};y_{12}),\cdots,(x_{1n};y_{1n})\\(x_{21};y_{21}),(x_{22};y_{22}),\cdots,(x_{2n};y_{2n})\\\vdots\\(x_{n1};y_{n1}),(x_{n2};y_{n2}),\cdots,(x_{nn};y_{nn})\end{pmatrix}$$

Every pair consists of $x_{ij}=(a_{ij}\oplus u_{ij},b_{ij}\oplus v_{ij})$, $y_{ij}=(c_{ij}\oplus w_{ij},d_{ij}\ominus x_{ij})$ with arbitrary i, j($1\le i, j\le n$). This shows that S_i has a data dependency from S_j. And when it is 0, it shows there exist no data dependency at all. But, $\oplus$ is used for constant s and t which can be used at the same time in added number equation $s\times i+t$. S_i has data dependency from S_j, so it is expressed in the form of (a_{ij},b_{ij}) and becomes the first solution of diophantine equation $x_{ij}\times J+g' = w_{ij}\times Q+h'$. @$((a_{ij}\ominus u_{ij},b_{ij}\ominus v_{ij}),(c_{ij}\oplus w_{ij},d_{ij}\oplus x_{ij}))$ stands for the fixed dependent distance, and #$((a_{ij}\oplus u_{ij},b_{ij}\oplus v_{ij}),(c_{ij}\oplus w_{ij},d_{ij}\ominus x_{ij}))$ is the upper bound on the modification of added numbers.

[Definition 2]

The product($\ominus$) of dependent matrix DM(m,n,p) and DM(m,l,p) on operand produces dependent matrix DM(m,n+1,p).

Using predefined DM expression

$$\begin{pmatrix}(x_{11};y_{11}),(x_{12};y_{12}),\cdots,(x_{1n};y_{1n})\\(x_{21};y_{21}),(x_{22};y_{22}),\cdots,(x_{2n};y_{2n})\\\vdots\\(x_{n1};y_{n1}),(x_{n2};y_{n2}),\cdots,(x_{nn};y_{nn})\end{pmatrix}\ominus\begin{pmatrix}(x_{11};y_{11}),(x_{12};y_{12}),\cdots,(x_{1n};y_{1n})\\(x_{21};y_{21}),(x_{22};y_{22}),\cdots,(x_{2n};y_{2n})\\\vdots\\(x_{n1};y_{n1}),(x_{n2};y_{n2}),\cdots,(x_{nn};y_{nn})\end{pmatrix}$$

$$=\begin{pmatrix}A\cdots B\\\vdots\\C\cdots D\end{pmatrix}$$

$$A=\begin{pmatrix}\begin{pmatrix}((a_{11}\oplus u_{11};b_{11}\oplus v_{11});(c_{11}\oplus w_{11};d_{11}\oplus x_{11}))\\((a_{11}\oplus u_{11};b_{11}\oplus v_{11});(c_{11}\oplus w_{11};d_{11}\oplus x_{11}))\end{pmatrix}\\\vdots\\\begin{pmatrix}((a_{1n}\oplus u_{1n};b_{1n}\oplus v_{1n});(c_{1n}\oplus w_{1n};d_{1n}\oplus x_{1n}))\\((a_{1n}\oplus u_{1n};b_{1n}\oplus v_{1n});(c_{1n}\oplus w_{1n};d_{1n}\oplus x_{1n}))\end{pmatrix}\end{pmatrix}$$

$$D=\begin{pmatrix}\begin{pmatrix}((a_{n1}\oplus u_{n1};b_{n1}\oplus v_{n1});(c_{n1}\oplus w_{n1};d_{n1}\oplus x_{n1}))\\((a_{n1}\oplus u_{n1};b_{n1}\oplus v_{n1});(c_{n1}\oplus w_{n1};d_{n1}\oplus x_{n1}))\end{pmatrix}\\\vdots\\\begin{pmatrix}((a_{nn}\oplus u_{nn};b_{nn}\oplus v_{nn});(c_{nn}\oplus w_{nn};d_{nn}\oplus x_{nn}))\\((a_{nn}\oplus u_{nn};b_{nn}\oplus v_{nn});(c_{nn}\oplus w_{nn};d_{nn}\oplus x_{nn}))\end{pmatrix}\end{pmatrix}$$

Element $A=\left(\begin{pmatrix}((a_{11}\oplus u_{11};b_{11}\oplus v_{11}),(c_{11}\oplus w_{11};d_{11}\oplus x_{11}))\\((a_{11}\oplus u_{11};b_{11}\oplus v_{11}),(c_{11}\oplus w_{11};d_{11}\oplus x_{11}))\end{pmatrix}\right)$ of the matrix is for calculating the data dependency. Operator $\ominus$ in dependent matrix stands for one element consists of n elements, and tells that each element is used for calculating the date dependency. If $((a_{ij}\oplus u_{ij},b_{ij}\oplus v_{ij}),(c_{ij}\ominus w_{ij},d_{ij}\oplus x_{ij}))$ shows S_i is dependent on S_j, and $((a_{kl}\oplus u_{kl},b_{kl}\oplus v_{kl}),(c_{kl}\ominus w_{kl},d_{kl}\oplus x_{kl}))$ shows S_j is dependent on S_k. Also, $\left(\begin{pmatrix}((a_{ij}\oplus u_{ij};b_{ij}\oplus v_{ij}),(c_{ij}\oplus w_{ij};d_{ij}\oplus x_{ij}))\\((a_{kl}\oplus u_{kl},b_{kl}\oplus v_{kl}),(c_{kl}\oplus w_{kl},d_{kl}\oplus x_{kl}))\end{pmatrix}\right)$ shows $S_i\rightarrow S_j\rightarrow S_k$ are dependent in the following order. The transitive relationship between them is shown in Lemma 1.

[Lemma 1]

One element in dependent matrix DM(m,2,p) that is $\left(\begin{pmatrix}((a_{ij}\oplus u_{ij},b_{ij}\oplus v_{ij}),(c_{ij}\oplus w_{ij},d_{ij}\oplus x_{ij}))\\((a_{kl}\oplus u_{kl},b_{kl}\oplus v_{kl});(c_{kl}\oplus w_{kl},d_{kl}\oplus x_{kl}))\end{pmatrix}\right)$ show transitive dependent relationships whose path length is 2, and it shows a transitive relationships in $\left(\begin{pmatrix}((a_{ij}\oplus u_{ij},b_{ij}\oplus v_{ij}),(c_{ij}\oplus w_{ij},d_{ij}\oplus x_{ij}))\\((a_{kl}\oplus u_{kl},b_{kl}\oplus v_{kl});(c_{kl}\oplus w_{kl},d_{kl}\oplus x_{kl}))\end{pmatrix}\right)=$

$$*((a_{mn}\oplus u_{mn},b_{mn}\oplus v_{mn}),(c_{mn}\oplus w_{mn};d_{mn}\oplus x_{mn}))$$

a_{mn}, u_{mn}, b_{mn}, v_{mn}, c_{mn}, w_{mn}, d_{mn}, x_{mn} are LCM(Least Common Multiple) of added number variable a_{ij}, u_{ij}, b_{ij}, v_{ij}, c_{ij}, w_{ij}, d_{ij}, x_{ij}, a_{kl}, u_{kl}, b_{kl}, v_{kl}, c_{kl}, w_{kl}, d_{kl}, x_{kl}.

(Proof)

$((a_{ij}\oplus u_{ij},b_{ij}\oplus v_{ij}),(c_{ij}\oplus w_{ij},d_{ij}\oplus x_{ij}))$ shows S_i is dependent on S_j. $((a_{kl}\ominus u_{kl},b_{kl}\oplus v_{kl}),(c_{kl}\oplus w_{kl},d_{kl}\oplus x_{kl}))$

shows dependency from S_j to S_k. From S_i through S_j and to S_k the path length is 2 and a transitive dependent relationship can be derived from LCM. a_{ij}, u_{ij}, b_{ij}, v_{ij} show first added number and its increased first solution's. In the same sense, from S_j to S_k, a_{kl}, u_{kl}, b_{kl}, v_{kl} the dependent relationships are showing added number and first solution c_{kl}, w_{kl}, d_{kl}, x_{kl} shows second added number and its first solution. The transitive dependent relationships $S_i \rightarrow S_j \rightarrow S_k$ are from $((a_{mn} \oplus u_{mn}, b_{mn} \oplus v_{mn}), (c_{mn} \oplus w_{mn}, d_{mn} \oplus x_{mn}))$. Initial increase a_{mn} and u_{mn} are from initial increase a_{mn} and u_{mn}. S_i to S_j is the first added numbers. The first increase is in v_{ij} and S_j to S_k $a_{mn}=u_{mn}=LCM(v_{ij},u_{kl})$ are fair.

$c_{mn} = w_{mn} = LCM(x_{ij},w_{kl})$

$b_{mn} = v_{ij} + b_{kl}(\text{if } b_{kl} = 0 \text{ then } b_{kl} = v_{kl}) - 1$

$d_{mn} = x_{ij} + d_{kl}(\text{if } d_{kl} = 0 \text{ then } d_{kl} = x_{kl}) - 1$

$v_{mn} = LCM(x_{ij},v_{kl})$

$x_{mn} = LCM(x_{ij},x_{kl})$

The transitive relationship can be determined through increased numbers. If one relation is increased by 3 and another one is increased by 4, the LCM of two increases is (3,4), (6,8), (9,12),….

[*Theorem 1*]

The transitive dependent relationship can be derived from LCM of two increased numbers disappears in the finite number of counts.

(*Proof*)

By lemma 1, transitive dependency relationship can be calculated through the increment of dependency relationships between two LCMs. Let be two dependent relationships increase m_1, m_2, and $m_3 = LCM(m_1,m_2)$. If m_3 is no bigger than loop added number of final n, proceed with LCM recursively. For that let the set of common multiplier's element value be $m_1, m_2, m_3, \cdots, m_k$. Then this values are fixed numbers and are bigger than 1. And finally $1 \le m_1 \le m_2 \le m_3 \le \cdots \le m_k$. At the end, from the count m_1 to m_k, let the number of count be k, and let there exists k that always satisfies $m_k \ge n$. If and only if k=n, it must be performed in sequential which means recursively n times. Finally, the data dependency will disappear. If k>n, the dependency will be eliminated with k-th count. The data dependency can be derived from LCM and then can be eliminated within limited numbers.

2.2 Data dependency removal algorithm

For loops regardless of whether the data dependence distance is uniform or non-uniform, we apply the parallelism extraction algorithm. In general, the loop consists of fixed dependent distance or variable dependent distance, as follows:

$S_1 : a_1(a_1 I+b_1,\ e_1 J+f_1) = \cdots$

$\vdots \qquad\qquad \vdots$

$S_i : a_i(a_i I+b_i,\ e_i J+f_i) = a_{i-1}(c_{i-1} I+d_{i-1},\ g_{i-1} J+h_{i-1}) + \cdots$

$S_{i+1}: a_{i+1}(a_{i+1} I+b_{i+1},\ e_{i+1} J+f_{i+1}) = a_i(c_i I+d_i,\ g_i J+h_i) + \cdots$

$\vdots \qquad\qquad \vdots$

$S_n : a_n(a_n I+b_n,\ e_n J+f_n) = a_{n-1}(c_{n-1} I+d_{n-1},\ g_{n-1} J+h_{n-1}) + \cdots$

$a_i(1 \le i \le n)$ may be different for each i.

For arbitrary $i(1 \le i \le n)$ between S_i and S_{i+1}, if and only if there exists data dependency, perform sentence S_i first and then S_{i+1}. Then sentence S_{i+1}'s value $a \times (i+1)$ must not be correct because of a_i. But if we perform S_{i+1} again, the value a_{i+1} of sentence S_{i+1} might be correct. In this situation, if we perform sentence S_{i+1} again, the value $a \times (i+1)$ in S_{i+1} would be correct. Finally, in order to remove the data dependency totally, we must know the length of path, that is, how many times the transitive relation is formed. These procedures are shown in <algorithm 1>, <algorithm 2>, and <algorithm 3>. In order to know what kind of data dependency exists, we may perform <algorithm 3>, and the result will be transformed into parallel codes.

<algorithm 1>

```
FOR all aij, cij <> 0
 IF the number of computation = 1 THEN
    IF uij and wij = 1 THEN
        DOALL k=1, m-bij +1
```

```
            DOALL l=1, n-dij +1
                  Si
            ENDDOALL
          ENDDOALL
       ELSE
          DOALL k=uij, m, uij
            DOALL k=wij, n, wij
                  Si
            ENDDOALL
          ENDDOALL
   ELSE IF the number of computation = 2 THEN
       IF ukl and wkl = 1 THEN
          DOALL k=bkl, m
            DOALL l=dkl, n
                  Sj
            ENDDOALL
          ENDDOALL
       ELSE
          DOALL k=u'kl, m, u'kl
            DOALL k=w'kl, n, wkl
                  Sj
            ENDDOALL
          ENDDOALL
   ELSE IF the number of computation = 3 THEN
       IF uuv and wuv = 1 THEN
          DOALL k=bkl + buv - 1, m
            DOALL l=dkl + duv - 1, n
                  Sv
            ENDDOALL
          ENDDOALL
       ELSE
          DOALL k=u'uv, m, wuv'
            DOALL k=u'uv, n, w'uv
                  Sv
            ENDDOALL
          ENDDOALL
```

<algorithm 2>

1. IF path length is 2,

$$DM=\begin{pmatrix} ((a_{ij}\oplus u_{ij}, b_{ij}\oplus v_{ij}), (c_{ij}\oplus w_{ij}, d_{ij}\oplus x_{ij})) \\ ((a_{kl}\oplus u_{kl}, b_{kl}\oplus v_{kl}), (c_{kl}\oplus w_{kl}, d_{kl}\oplus x_{kl})) \end{pmatrix}$$

   THEN

   $U_{kl} = LCM(b_{ij}v_{ij}, a_{kl}u_{kl})$

   $W'_{kl} = LCM(d_{ij}x_{ij}, c_{kl}w_{kl})$

2. IF path length is 3,

$$DM=\begin{pmatrix} ((a_{ij}\oplus u_{ij}, b_{ij}\oplus v_{ij}), (c_{ij}\oplus w_{ij}, d_{ij}\oplus x_{ij})) \\ ((a_{kl}\oplus u_{kl}, b_{kl}\oplus v_{kl}), (c_{kl}\oplus w_{kl}, d_{kl}\oplus x_{kl})) \\ ((a_{uv}\oplus u_{uv}, b_{uv}\oplus v_{uv}), (c_{uv}\oplus w_{uv}, d_{uv}\oplus x_{uv})) \end{pmatrix}$$

   THEN

   $u'_{uv} = LCM(b_{kl}v_{kl}, a_{uv}u_{uv})$

   $w'_{uv} = LCM(d_{kl}x_{kl}, c_{uv}u_{uv})$

<algorithm 3>

1. Make a DM of path length 1.
2. IF there are DMs with identical subscript THEN change to other name.
3. For all i, j
   IF $(a_{ij}\ u_{ij}) > m$ *or* $(c_{ij}\ w_{ij}) > n$
   THEN change the element to 0
4. For all non-zero elements, apply <algorithm 1>.
5. IF $b_{ij} > m$ or $d_{ij} > n$ THEN change the element to 0.
6. IF all element =0 THEN goto 12
   ELSE increase path length by 1.
7. IF there is identical elements
   THEN remove all elements except one.
8. In DM, apply <algorithm 2> and look for data dependency and mutate DM.
9. IF the selected LCM value is bigger than that of the final DO loop
   THEN change that element to 0.
10. Apply <algorithm 1> to every non-zero element.
11. Goto 6
12. IF there is added number THEN apply <algorithm 1>.
13. Except for the mutated sentences, perform doall S_i to every sentence.

3. EXTRACTION OF PARALLELISM FROM THE LOOP WITH PROCEDURE CALLS

In this chapter, we describe inter-procedure transformation and the extraction of parallelism from the loop with procedure calls using the data dependency removal method.

3.1 Inter-Procedure Transformation

Among many transformation methods applied for procedures, expanded inlining transformation method replaces every procedure calling sentences with called procedure codes. Loop extraction transformation method, a loop of the called procedure is replaced at the outer part of the caller's calling point. In Loop embedding, the loop based which includes procedure calls is replaced at the called procedure. In Procedure cloning, if a procedure is called many times, we prepare an optimal copy of the procedure, and let the callers called the copy[9,10].

3.2 Extraction of Parallelism from the Loop with Procedure Calls

In order to extract the parallelism from the loop with procedure calls, we used the data dependency removal method. The method transforms procedures used in the loop using inlining and then applies the data dependency removal algorithm.

<algorithm 4>
1. Draw procedure call multi-graph
2. Expand it into augmented call graph
3. Calculation of information between procedures
4. Dependency analysis
5. IF (one procedure is called and the caller related variables are not changed)
   THEN goto <algorithm 6>
   ELSE goto <algorithm 7>
6. Insert the data dependency removal algorithm <algorithm 5> and make parallel code

<algorithm 5>
1. Initialization
   S is number of sentences, DMA, DMB, DMC are array.
   s_w= 0
2. Use GCD arithmetic function call for diophantine method calculation, and derive pass=1 and S*S size 2 dimensional dependent matrix DMB.
   IF (The index variable is same) THEN rename
3. Set loop index variables i, j. They are N_1, N_2, at best.
   For all DMB matrix i,j
   IF $((a_{ij}u_{ij}) > N_1 \mid\mid (c_{ij}w_{ij}) > N_2)$
   THEN change that element to 0
   IF s_w== 0 THEN DMA = DMB, s_w = 1
4. Using DMB matix on all nonzero element
   Forall i, j
   IF $(u_{ij}$ & $w_{ij} == 1)$ THEN uniform
   ELSE IF $(u_{ij}$ & w_{ij} != 1) THEN non-uniform
   ELSE complex type to doall S_i change
5. IF the same element exists, remove all except one
6. IF all element = 0 THEN go to 8
   ELSE
      to increase pass matrix production $S{*}S^{pass+1}$, derivesized dependent matrix
      DMC = DMA * DMB
      pass++
      DMB = DMC
      free DMC
   ENDIF
7. Go to 3
8. Except changed sentences, change all sentences into DOALL S_i sentence.

<algorithm 6>
1. repeat(until all the procedure be optimized)
2. repeat(until the inter procedure be optimized)
3. apply inlining in a reverse topological order

<algorithm 7>
1. apply inlining in a reverse topological order
2. repeat(until all the procedure be optimized)

4. PERFORMANCE ANALYSIS

To show the proposed method is superior to the conventional ones, we evaluated data distance using two of the most widely used sample codes[8,9] for the uniform and non-uniform type code.

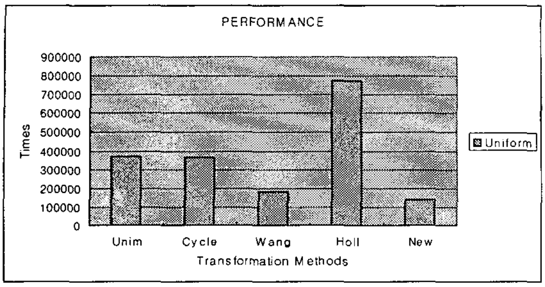
The performance analysis in Figure 2(a) for uniform code is compared with the data dependency removal method and linear transformation methods such as unimodular, selective cycle shrinking, Höllander, and Chen&Wang.

Figure 3(a) shows the performance of unimodular, selective cycle shrinking, Hollander, Chen&Wang and our new method when the distance is uniform. We performed the comparison on CRAY-T3E machine with the fixed values of N_1=20, N_2=100, and the number of processes is 4. Unimodular method and cycle shrinking method are similar in the performance because they divide the blocks and process sequentially, and then calculate the value of dependent distance. Hollander method isthe worst method, because it processes the white node and black node in serial form. The best performance is achieved when the data dependency removal method is processed until there is no more data dependency. The number of parallel code used is $\lambda=2N_1-4$ in tiling, interchanging, selective shrinking. $\lambda=(N_2-4)/4$ is used for skewing, unimodular and Chen&Wang method. It is $\lambda=(N_1 \times N_2)/2$ in Hollander method. Finally, the data dependency removal method takes place only $\lambda=2+N_1/10$ times.

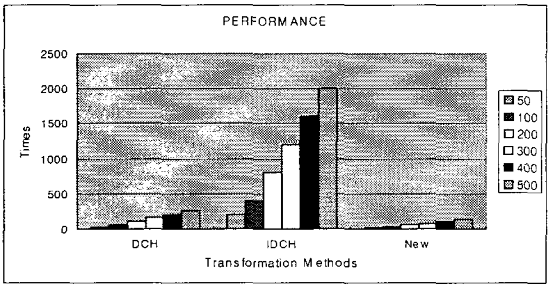
```
DO  I = 3, N1
  DO  J = 5, N2
    A(I, J) = B(I-3, J-5)
    B(I, J) = A(I-2, J-4)
```
(a) uniform code

```
DO I = 1, N1
DO J = 1, N2
  A(2*I+J+1, I+J+3)= …
           = A(2*J+3, I+1)
```
(b) non-uniform code

Fig. 2. Example Code 1.



(a) uniform



(b) non-uniform

Fig. 3. Performance results of Example code 1

Figure 3(b) shows the performance of DCH method, IDCH method, and proposed method when the distance is non-uniform. For the experimental result measurement, we increased the number of iterations from 50 to 500 and the number of processes is 4. With the DCH method, $\lambda = N_1/2$. With the IDCH method, $\lambda = N_1 \times N_2 / T_n$($T_n$: number of tile). For the data dependency method $\lambda = 1 + Min(N_1, N_2)/4$. This shows that the proposed method is effective for both uniform and non-uniform code.

We expand a loop procedure call to multiple loops. The comparison and analysis of data dependency removal method is performed on the CRAY T3E system. We gradually increased the number of processors to 2, 4, 8, 16, 32 and applied data dependency distance method for uniform, non-uniform, and complex code. Using the example in Figure 4, we compared loop extraction transformation method, loop embedding transformation method, and procedure cloning transformation method.

The data dependency removal method for transformation between procedures performs inline expansion to remove the data dependency until there is no more parallel data dependency. The same process is applied to loop extraction and loop embedding, which can reduce the overhead in procedure calls. Procedure cloning is divided into the sequential process part and parallel part, and it produces the best parallelism. For the case of non-uniform and complex data dependency distance, parallelization is possible for the expanded data dependency removal method only. Therefore, we apply parallelization for data dependency method, and apply sequence for all other methods.

```
SUBROUTINE P
 real a(n, n)
 integer i
 do i = 1, 10
  call Q(a, i)
  call Q(a, i+1)
 enddo

SUBROUTIN Q(f, i)
 real f(n, n)
 integer i, j
 do j = 1, 100
   f(i,j)=f(i,j)+…

 endd
```

```
SUBROUTINE P
 real a(n, n)
 integer i
 do i = 1, 10
   call Q(a, i*3)
   call Q(a, i*5)
 enddo

SUBROUTIN Q(f, i)
 real f(n, n)
 integer i, j
 do j = 1, 100
 f(i,j*5)=f(i,j*4)
        + ...
 enddo
```

```
SUBROUTINE P
 real a(n, n)
 integer i
 do i = 1, 10
   call Q(a, i)
   call Q(a, i*5)
 enddo

SUBROUTIN Q(f, i)
 real f(n, n)
 integer i, j
 do j = 1, 100
   f(i,j)=f(i,j)+...

 enddo
```

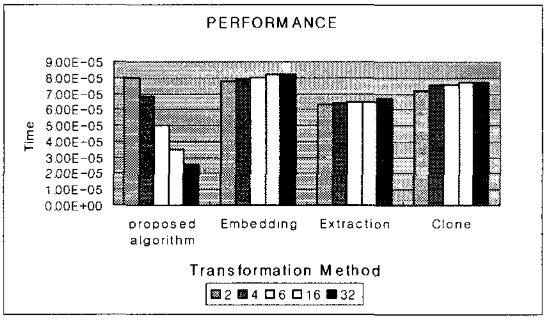
(a)uniform code (b)non-uniform code (c)complex code

Fig. 4. Example code II.

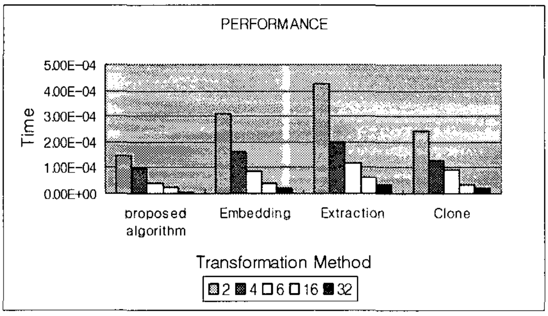
The summary of the performance analysis is in Figure 5. The expanded data dependency removal method in data dependent distance for uniform, non-uniform, complex code are all becoming better with more processors. In the situation where the distance is uniform, the procedure cloning transformation method is better than loop embedding and loop extraction method. For data dependent distance in non-uniform and complex code, only the data dependent removal method can be parallelized. In that sense, this method is the best.

5. CONCLUSION

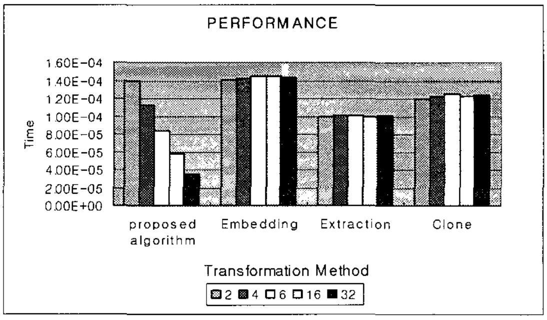
Most programs spend their execution time in the loop structure. For this reason, there are many on-going studies on transforming sequential programs into parallel programs. Most of the studies



(a) uniform code



(b) non-uniform code



(c) complex code

Fig. 5. Performance of Example code II.

are focused on extracting the parallelism, and then transforming it into inter-procedural parallelism. However, these methods can only be applied to uniform code. This paper proposed an algorithm that is applicable for both uniform and non-uniform dependency distance code. To prove this, we used an applicable data dependent removal method for a single loop. The experimental result shows that the execution time of our proposed method is superior to other methods. We also applied this method to the inter-procedure algorithm and showed that our method is significantly efficient as well.

Since the proposed method requires some times for analysis, we will try to reduce the analysis time as a future work.

6. REFERENCE

[1] Allen, F., M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An Overview of the PTRAN analysis System for Multiprocessing," *Journal of Parallel and Distributed Computing,* Vol. 5, No. 5, 1998.

[2] Wolfe, M. J., "High Performance compiler for Parallel Computing," *Oregon Graduate Institute of Science & Technology*, 1996.

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

[4] Banerjee, U., Loop Transformations for Restructuring Compilers : The Foundations, *Kluwer Academic Publishers*, 1993.

[5] D'Hollander, E. H., "Partitioning and Labeling of Loops by Unimodular Transformations," *IEEE Trans. on Parallel and Distributed Systems,* Vol. 3, No. 4, July 1992.

[6] Chen, Y-S and S-D Wang, "A Parallelizing Compilation Approach to Maximizing Parallelism within Uniform Dependence Nested Loops," *Dept. of Electrical Engineering, National Taiwan University*, 1993.

[7] Tzen, T. H. and L. M. Ni, "Dependence Uniformization : A Loop Parallelization Technique," *IEEE Transactions on Parallel and Distributed Systems*, May 1993.

[8] Punyamurtula,S., and V. Chaudhary, Compile-Time Partitioning of Nested Loop Iteration Spaces with Non-uniform Dependences, *In Journal of Parallel Algorithm and Architecture,* 1996.

[9] Hall, M. W., K. Kennedy, and K. S. Mckinley.

Interprocedural Transformations for Parallel Code Generation, Technical Report 1149-s, *Dept. of Computer Science*, Rice University, 1991.

[10] Hall, M. W., "Managing Interprocedural Optimization, Ph.D thesis, *Dept. of Computer Science*, Rice University, 1991.

[11] Mckinley, K. S., "A Compiler Optimization Algorithm for Shared-Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*. 9(8): 769-787, August 1998.

Doo-Soon Park

1983 Computer Science, Chungnam National University (M.S.)

1988 Computer Science, Korea University(Ph. D.)

2004~2005 Visting Professor, University of Colorado at Denver

2002~2003 Dean, Engineering College, Soonchunhyang University

2000~present Director, Korea Multimedia Society

2006~present Director, u-Healthcare Research Center, Soonchunhyang University

1985~present professor, Division of Computer Science and Engineering, Soonchunhyang

Research Areas : Parallel Processing, Data Mining, mutimedia Information processing

Min-Hyung Choi

Min-Hyung Choi is the Director of Computer Graphics and Virtual Environments Laboratory and an Associate Professor of Computer Science Department at Univ. of Colorado at Denver and Health Sciences Center. He received his M.S. and Ph.D. from University of Iowa in 1996 and 1999 respectively. His research interests are in Computer Graphics, Scientific Visualization and Human Computer Interaction with an emphasis on physically-based modeling and simulation for medical and bioinformatics applications. Contact him at min.choi@cudenver.edu.